

## 0. Installation du sous-système linux sous Windows 10

<https://docs.microsoft.com/fr-fr/windows/wsl/install-win10#manual-installation-steps>

<https://korben.info/linux-wsl-gui-interface-graphique-windows-10.html>

terminal linux en ligne : <https://bellard.org/jslinux/>

## 1. Les processus

Toute machine est dotée d'un **système d'exploitation** (Operating **System OS**) qui a pour fonction de charger les programmes depuis la mémoire de masse (ROM, DD, SSD ) et de lancer leur exécution en leur créant des processus (proces) en anglais dans la ( RAM ), de gérer l'ensemble des ressources, de traiter les interruptions ainsi que les entrées-sorties et enfin d'assurer la sécurité globale du système.

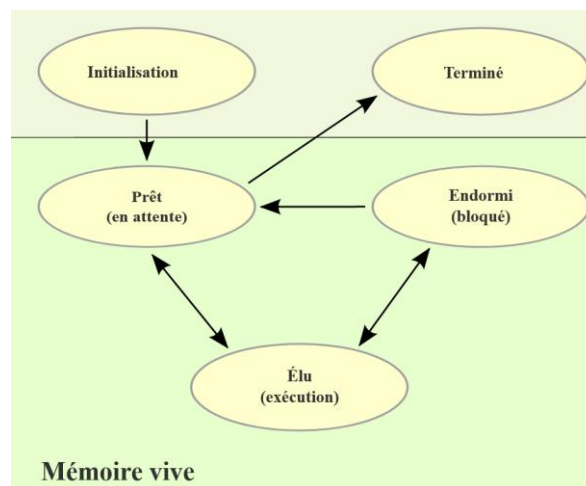
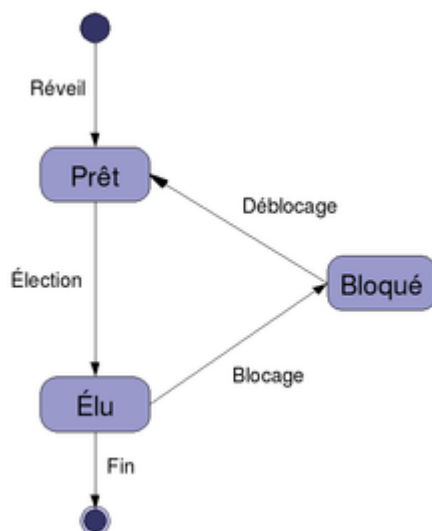
Les OS modernes sont **multitâches** (capable d'exécuter plusieurs processus de façon quasi simultanée). Ils sont capables d'allouer les ressources (mémoires, temps processeur, entrées/sorties) nécessaires aux processus et d'assurer que le fonctionnement d'un processus n'interfère pas avec celui des autres (isolation).

**Un Processus** (en [anglais](#), *process*), en [informatique](#), est un [programme](#) en cours d'[exécution](#) par un [ordinateur](#). De façon plus précise, il peut être défini comme :

- un ensemble d'instructions à exécuter le plus souvent chargé depuis la mémoire de masse vers la mémoire vive RAM ordinateur .
- un [espace d'adressage](#) en mémoire vive pour stocker la [pile](#), les données de travail, etc.
- la valeur des registres (généraux et spécialisés) de la CPU lors de l'exécution ;
- les ressources qui lui ont été allouées par le système d'exploitation (mémoire principale, fichiers ouverts, périphériques utilisés, etc.)

Il faut bien faire la différence entre un programme qui est un fichier inerte regroupant des instructions de la CPU et un processus qui est un élément actif.

Diagramme d'états d'un processus



Un processus peut être vu comme quelque chose qui prend un certain temps, donc qui a un début et (parfois) une fin. Un processus peut se rencontrer sous différents états.

**Etat "élu"** : le processus est en train de s'exécuter, c'est-à-dire qu'il utilise ressources du microprocesseur.

Un processus qui se trouve dans l'état élu peut demander à accéder à une ressource pas forcément disponible instantanément (typiquement lire une donnée sur le disque dur). Le processus ne peut pas poursuivre son exécution tant qu'il n'a pas obtenu cette ressource.

En attendant de recevoir cette ressource, il passe de l'état "élu" à l'**état "bloqué"**.

Lorsque le processus finit par obtenir la ressource attendue, celui-ci peut potentiellement reprendre son exécution. Cependant, bien que les systèmes d'exploitation permettent de gérer plusieurs processus "en même temps", il n'en demeure pas moins qu'un seul processus peut se trouver dans un **état "élu"** (le microprocesseur ne peut "s'occuper" que d'un seul processus à la fois). Quand un processus passe d'un **état "élu" à un état "bloqué"**, un autre processus peut alors "prendre sa place" et passer dans l'état "élu". Le processus qui vient de recevoir la ressource attendue ne va donc pas forcément pouvoir reprendre son exécution tout de suite, car pendant qu'il était dans à état "bloqué" un autre processus a "pris sa place". Un processus qui quitte l'état bloqué ne repasse pas forcément à l'état "élu", il peut, en attendant que "la place se libère" passer dans l'**état "prêt"** ("j'ai obtenu ce que j'attendais, je suis prêt à reprendre mon exécution dès que la "place sera libérée").

Le passage de l'état "prêt" vers l'état "élu" constitue l'opération "d'élection". Le passage de l'état élu vers l'état bloqué est l'opération de "blocage". Un processus est toujours créé dans l'état "prêt". Pour se terminer, un processus doit obligatoirement se trouver dans l'état "élu".

**Le système d'exploitation (OS – Operating System) est le "chef d'orchestre" qui attribue aux processus leur état "élu", "bloqué" ou "prêt". On dit que le système d'exploitation gère l'ordonnancement des processus (un processus sera prioritaire sur un autre...).**

**Remarque** : un processus qui utilise une ressource doit la "libérer" une fois qu'il a fini de l'utiliser afin de la rendre disponible pour les autres processus. Pour libérer une ressource, un processus doit obligatoirement être dans un état "élu".

Un processus peut être démarré par un utilisateur par l'intermédiaire d'un périphérique ou bien par un autre processus : les *applications* des utilisateurs sont des ensembles de processus plus ou moins complexes.

Comme nous venons de le voir, Le système d'exploitation est chargé d'allouer les ressources (mémoires, temps processeur, entrées/sorties) nécessaires aux processus et d'assurer que le fonctionnement d'un processus n'interfère pas avec celui des autres (isolation). Il peut aussi fournir une API (Application Programming Interface) pour permettre la communication inter-processus (IPC).

Outre le multiplexage des ressources matérielles, le système peut contrôler l'accès des processus aux ressources selon une matrice de droits (permissions d'accès) et également associer les processus aux utilisateurs, qui sont les récipiendaires d'un ensemble de droits d'accès : un processus a les droits de l'utilisateur qui l'a démarré.

Un processus peut s'arrêter de plusieurs manières :

1. Arrêt normal (volontaire).
2. Arrêt pour erreur (volontaire).
3. Arrêt pour erreur fatale (involontaire).
4. Le processus est arrêté par un autre processus (involontaire).

La plupart des systèmes offrent la distinction entre processus *lourd* (tels que nous les avons décrits), qui sont a priori complètement isolés les uns des autres, et *processus légers* (*Threads* en anglais), qui ont un espace mémoire (et d'autres ressources) en commun.

Dans le cas de processus comportant plusieurs processus légers (ou suivant l'expression souvent utilisée multi-thread) il existe un état du processeur (un contexte d'exécution) distinct pour chaque processus léger.

Le **CPU ( Central Processing Unit)** processeur en français exécute le programme en allant chercher dans la mémoire vive la prochaine instruction à effectuer et en y organisant la disposition des données. Les CPU modernes sont multicœurs ( 4 , 6 , 12 ... 48 et plus ) , ils possèdent plusieurs blocs constitués d'un processeur complet qui peuvent travailler en « parallèle ». Chacun de ces cœurs pouvant effectuer un processus.

On constate néanmoins que le nombre d'applications simultanées sur un ordinateur dépasse souvent les 4 ou 8 pourtant celles-ci ne semblent pas bloquer l'ordinateur. Il y a plusieurs raisons :

- **Les interruptions**

Une interruption est une suspension temporaire de l'exécution d'un programme informatique par le microprocesseur afin d'exécuter un programme prioritaire. Cette interruption provient :

- du signal de l'horloge
- d'un signal externe : click souris, de la gestion de l'interface graphique, port de communication, USB, disque dur, exploitation mémoire ....

Lors d'une interruption, le processeur enregistre tout ou une partie de son état interne, généralement dans la pile système, et exécute ensuite une routine d'interruption, généralement en suivant les directives d'une table indiquant pour chaque type d'interruption, le sous-programme à exécuter.

Une fois le traitement de l'interruption terminé, la routine se finit normalement par une instruction de retour d'interruption, qui restaure l'état enregistré et fait repartir le processeur de l'endroit où il avait été interrompu.

- L'emploi de « processus légers » les Threads

On peut pour un processus partager ses données et ses ressources dans plusieurs processus légers ou Threads concurrents qui gérés correctement donne l'illusion de s'exécuter simultanément. Pour une interface graphique l'utilisation des Threads permet d'éviter les phases d'attente par exemple.

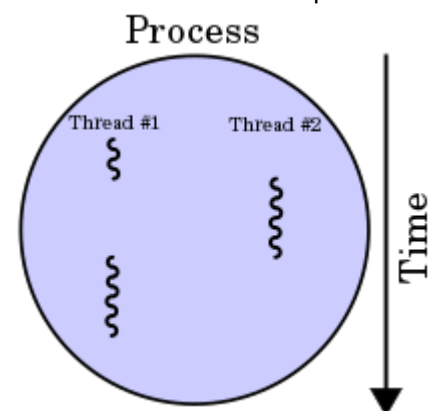
\*

- Le multiprocessing

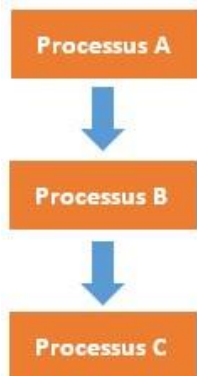
Le nombre de cœurs des CPU qui autorise aussi le fonctionnement simultané qui peut être employé de concert avec le *multithreading* qui augmente l'utilisation d'un seul cœur en tirant profit des propriétés des *threads* et du [parallélisme au niveau des instructions](#).

- La gestion multitâche des OS préemptif

Dans les OS moderne un [ordonnanceur](#) distribue le temps du [processeur](#) entre les différents [processus](#). Dans un système préemptif, à l'inverse d'un système collaboratif, l'ordonnanceur peut interrompre à tout moment une tâche en cours d'exécution pour permettre à une autre tâche de s'exécuter.



## 2. Création des processus et leur identification

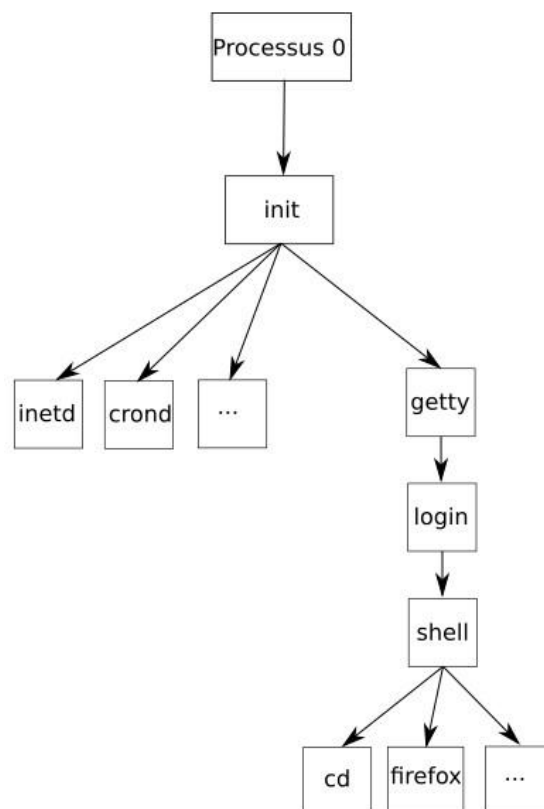


Un processus peut créer un ou plusieurs processus à l'aide d'une commande système ("fork" sous les systèmes de type Unix). Imaginons un processus A qui crée un processus B. On dira que A est le père de B et que B est le fils de A. B peut, à son tour créer un processus C (B sera le père de C et C le fils de B). On peut modéliser ces relations père/fils par une structure arborescente (voir le schéma ci-contre).

Si un processus est créé à partir d'un autre processus, comment est créé le tout premier processus ?

Sous un système d'exploitation comme Linux, au moment du démarrage de l'ordinateur un tout premier processus (appelé processus 0 ou encore Swapper) est créé à partir de "rien" (il n'est le fils d'aucun processus). Ensuite, ce processus 0 crée un processus souvent appelé "init" ("init" est donc le fils du processus 0).

À partir de "init", les processus nécessaires au bon fonctionnement du système d'exploitation Linux sont créés (par exemple les processus "crond", "inetd", "getty",...). Puis d'autres processus sont créés à partir des fils de "init"...



**Schéma de la création des processus de base sous Linux lors du lancement du système**

Chaque processus possède un identifiant appelé **PID (Process Identification)**, ce PID est un nombre entier. Le premier processus créé au démarrage du système a pour PID 0, le second 1, le troisième 2... Le système d'exploitation utilise un compteur qui est incrémenté de 1 à chaque création de processus, le système utilise ce compteur pour attribuer les PID aux processus.

Chaque processus possède aussi un **PPID (Parent Process Identification)**. Ce PPID permet de connaître le processus parent d'un processus (par exemple le processus "init" vu ci-dessus à un PID de 1 et un PPID de 0). À noter que le processus 0 ne possède pas de PPID (c'est le seul dans cette situation).

### 3. Ordonnancement des processus

Dans un système multi-utilisateur à temps partagé, plusieurs processus peuvent être présents en mémoire centrale en attente d'exécution. Si plusieurs processus sont prêts, le système d'exploitation doit gérer l'allocation du processeur aux différents processus à exécuter. C'est l'ordonnanceur qui s'acquitte de cette tâche.

#### 3.1. Notion d'ordonnancement

Le système d'exploitation d'un ordinateur peut être vu comme un ensemble de processus dont l'exécution est gérée par un processus particulier : l'ordonnanceur (scheduler en anglais).

Un ordonnanceur fait face à deux problèmes principaux :

- le choix du processus à exécuter ;
- le temps d'allocation du processeur au processus choisi.

Un système d'exploitation multitâche est préemptif lorsque celui-ci peut arrêter (réquisition) à tout moment n'importe quelle application pour passer la main à la suivante. Dans les systèmes d'exploitation préemptifs on peut lancer plusieurs applications à la fois et passer de l'une à l'autre, voire lancer une application pendant qu'une autre effectue un travail. Il y a aussi des systèmes d'exploitation dits multitâches, qui sont en fait des « multi-tâches coopératifs ». Quelle est la différence ? Un multitâche coopératif permet à plusieurs applications de fonctionner et d'occuper des plages mémoire, laissant le soin à ces applications de gérer cette occupation, au risque de bloquer tout le système. Par contre, avec un « multi-tâche préemptif », le noyau garde toujours le contrôle (qui fait quoi, quand et comment), et se réserve le droit de fermer les applications qui monopolisent les ressources du système. Ainsi les blocages du système sont inexistant.

#### 3.2. Objectifs de l'ordonnanceur d'un système multi-utilisateurs

Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont, entre autres :

- s'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur ;
- minimiser le temps de réponse ;
- utiliser le processeur à 100% ;
- utiliser d'une manière équilibrée les ressources ; • prendre en compte les priorités ; • être prédictible.

Ces objectifs sont parfois complémentaires, parfois contradictoires : augmenter la performance par rapport à l'un d'entre eux peut se faire au détriment d'un autre. Il est impossible de créer un algorithme qui optimise tous les critères de façon simultanée.

#### 3.3. Ordonnanceurs non préemptifs : First-Come First-Served (FCFS) et Short Job First (SJF)

Dans un système à ordonnancement non préemptif ou sans réquisition, le système d'exploitation choisit le prochain processus à exécuter, en général, le **Premier Arrivé est le Premier Servi PAPS (ou First-Come First-Served FCFS)** ou le **plus court d'abord (Short Job First SJF)**. Il lui alloue le processeur jusqu'à ce qu'il se termine ou qu'il se bloque (en attente d'un événement). Il n'y a pas de réquisition. Si l'ordonnanceur fonctionne selon la stratégie SJF, il choisit, parmi le lot de processus à exécuter, le plus court (plus petit temps d'exécution). Cette stratégie est bien adaptée au traitement par lots de processus dont les temps maximaux d'exécution sont connus ou fixés par les utilisateurs car elle offre un meilleur temps moyen de séjour. Le temps de séjour d'un processus (temps de rotation ou de virement) est l'intervalle de temps entre la soumission du processus et son achèvement.

Considérons par exemple un lot de quatre processus notés A, B, C, D dont les temps respectifs d'exécution sont,  $t_A$ ,  $t_B$ ,  $t_C$  et  $t_D$ . Le premier processus se termine au bout du temps  $t_A$ ; le deuxième processus se termine au bout du temps  $t_A + t_B$ ; le troisième processus se termine au bout du temps  $t_A + t_B + t_C$ ; le quatrième processus se termine au bout du temps  $t_A + t_B + t_C + t_D$ .

Le temps moyen de séjour noté  $\langle t \rangle$  est :  $\langle t \rangle = (4 t_A + 3 t_B + 2 t_C + t_D) / 4$

On obtient le meilleur temps de séjour pour  $t_A \leq t_B \leq t_C \leq t_D$ .

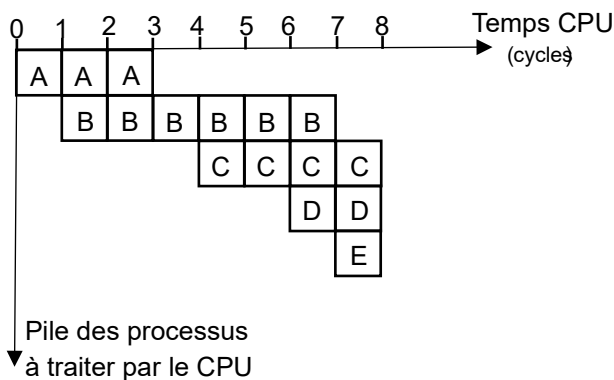
Toutefois, l'ordonnancement du plus court d'abord est optimal que si les travaux sont disponibles simultanément.

**Exemple :**

Considérons cinq processus notés A, B, C, D et E, dont les temps d'exécution et leurs arrivages respectifs sont donnés dans le tableau ci-contre

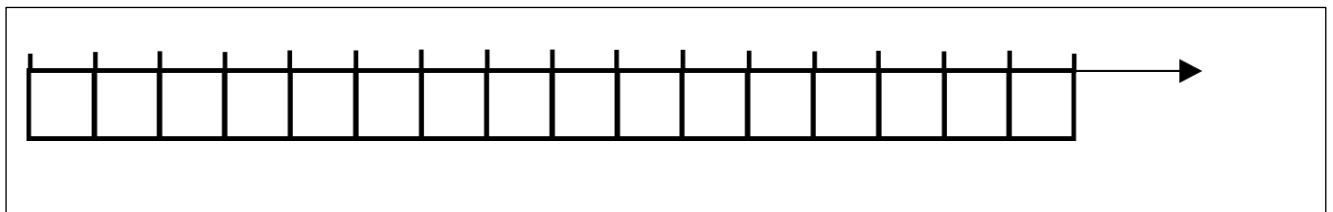
Processus	Temps d'exécution	Temps d'arrivée
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

Ou encore



**Exercice 1 :** Ordonner ces processus suivant l'algorithme "Premier arrivé premier servi" : **First-Come First-Served (FCFS)**, en supposant qu'il n'y pas de blocage.

**Schéma d'exécution**



- ✓ Le temps de séjour pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison.

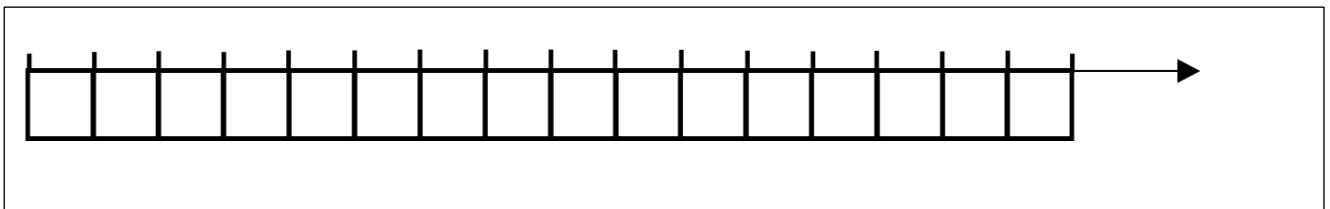
Processus	Temps de séjour


- ✓ Le **temps moyen de séjour** est :
- ✓ Le **temps d'attente** est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente

- ✓ Le temps moyen d'attente est :

□ **Exercice 2** Ordonancer ces processus suivant l'algorithme "**Le plus court d'abord**" : **Short Job First (SJF)**  
**Schéma d'exécution**



- ✓ Le **temps de séjour** pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison.

Processus	Temps de séjour

- ✓ Le **temps moyen de séjour** est :
- ✓ Le **temps d'attente** est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente


✓ **Le temps moyen d'attente est :**

Conclusion

**Remarque :** Les ordonnanceurs non préemptifs ne sont généralement pas intéressants pour les systèmes multi-utilisateurs car les temps de réponse ne sont pas toujours acceptables.

### 3.4. Ordonnanceurs préemptifs : Shortest Remaining Time (SRT) et Round Robin (RR)

Dans un schéma d'ordonnanceur préemptif, ou avec réquisition, pour s'assurer qu'aucun processus ne **s'exécute pendant trop de temps**, les **ordinateurs ont une horloge électronique** qui génère périodiquement une interruption. A chaque interruption d'horloge, le système d'exploitation reprend la main et décide si le processus courant doit poursuivre son exécution ou s'il doit être suspendu pour laisser place à un autre. S'il décide de suspendre son exécution au profit d'un autre, il doit d'abord sauvegarder l'état des registres du processeur avant de charger dans les registres les données du processus à lancer. C'est qu'on appelle la commutation de contexte ou le changement de contexte. Cette sauvegarde est nécessaire pour pouvoir poursuivre ultérieurement l'exécution du processus suspendu. Le processeur passe donc d'un processus à un autre en exécutant chaque processus pendant quelques dizaines ou centaines de millisecondes. Le temps d'allocation du processeur au processus est appelé quantum. Cette commutation entre processus doit être rapide, c'est-à-dire, exiger un temps nettement inférieur au quantum. Le processeur, à un instant donné, n'exécute réellement qu'un seul processus, mais pendant une seconde, le processeur peut exécuter plusieurs processus et donne ainsi l'impression de parallélisme (pseudo-parallélisme).

#### Problèmes soulevés :

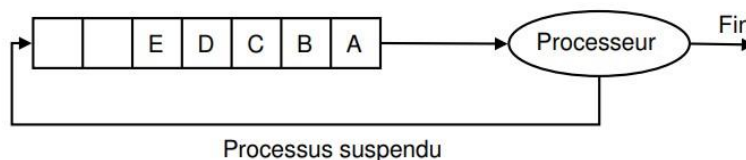
- Choix de la valeur du quantum.
- Choix du prochain processus à exécuter dans chacune des situations suivantes :
  1. Le processus en cours se bloque (passe à l'état Attente).
  2. Le processus en cours passe à l'état Prêt (fin du quantum...).
  3. Un processus passe de l'état Attente à l'état Prêt (fin d'une E/S).
  4. Le processus en cours se termine.

#### □ **Algorithme du plus petit temps de séjour : Shortest Remaining Time (SRT)**

L'ordonnancement du plus petit temps de séjour ou **Shortest Remaining Time (SRT)** est la version préemptive de l'algorithme **SJF**. Un processus arrive dans la file de processus, l'ordonnanceur compare la valeur espérée pour ce processus contre la valeur du processus actuellement en exécution. Si le temps du nouveau processus est plus petit, il rentre en exécution immédiatement.

#### □ **Algorithme du tourniquet circulaire : Round Robin (RR)**

L'algorithme du tourniquet, circulaire ou **Round Robin (RR)** représenté sur la figure ci-dessous est un algorithme ancien, simple, fiable et très utilisé. Il mémorise dans une file du type **FIFO (First In First Out)** la liste des processus prêts, c'est-à-dire en attente d'exécution.





## Ordonnancement circulaire

- **Choix du processus à exécuter**

Il alloue le processeur au processus en tête de file, pendant un quantum de temps. Si le processus se bloque ou se termine avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus (celui en tête de file). Si le processus ne se termine pas au bout de son quantum, son exécution est suspendue. Le processeur est alloué à un autre processus (celui en tête de file). Le processus suspendu est inséré en queue de file. Les processus qui arrivent ou qui passent de l'état bloqué à l'état prêt sont insérés en queue de file.

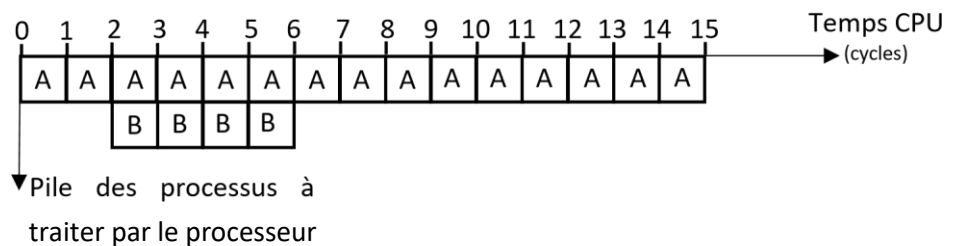
- **Choix de la valeur du quantum**

Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur. Un quantum trop élevé augmente le temps de réponse des courtes commandes en mode interactif. Un quantum entre 20 et 50 ms est souvent un compromis raisonnable.

**Remarque :** le quantum était de 1 s dans les premières versions d'UNIX.

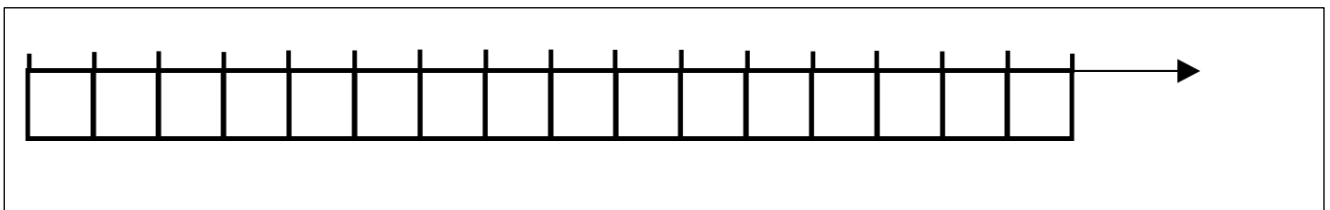
### Exemple :

Soient deux processus A et B prêts tels que A est arrivé en premier suivi de B, 2 unités de temps après. Les temps CPU nécessaires pour l'exécution des processus A et B sont respectivement 15 et 4 unités de temps. Le temps de commutation est supposé nul.



**Exercice 1 :** Ordonner ces processus avec **Algorithme du plus petit temps de séjour : Shortest Remaining Time (SRT)**

### Schéma d'exécution



- ✓ Le **temps de séjour** pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison.

Processus	Temps de séjour

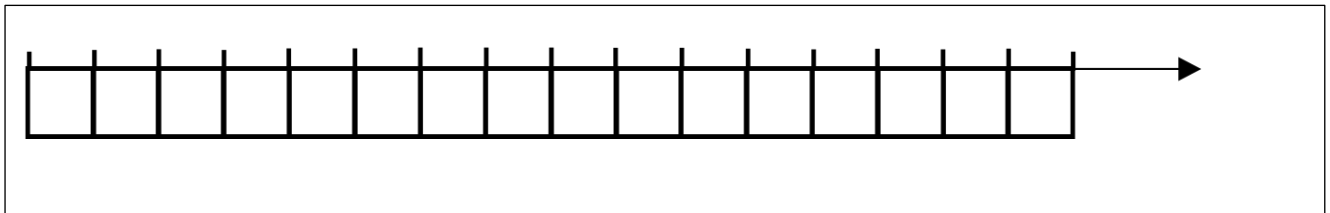
- ✓ Le **temps moyen de séjour** est :
- ✓ Le **temps d'attente** est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente

✓ Le **temps moyen d'attente** est :

**Exercice 2** : Ordonnancer ces processus avec l'**Algorithme du tourniquet circulaire : Round Robin** (quantum = 10 unités de temps)

**Schéma d'exécution**



Processus	Temps de séjour

✓ Le **temps moyen de séjour** est :

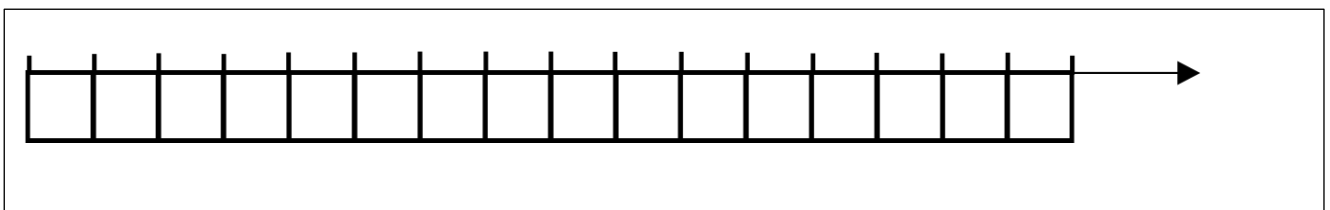
✓ Le **temps d'attente** est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente

✓ Le **temps moyen d'attente** est :

**Exercice 3** : Ordonnancer ces processus avec l' **Algorithme du tourniquet circulaire : Round Robin** (quantum = 10 unités de temps)

**Schéma d'exécution**



✓ Le **temps de séjour** pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison.

Processus	Temps de séjour

✓ Le **temps moyen de séjour** est :

- ✓ Le **temps d'attente** est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente

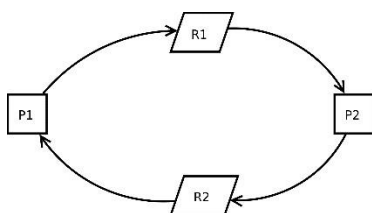
- ✓ Le **temps moyen d'attente** est :

## 4. Interblocage (deadlock)

### Définition

Un **interblocage** (ou **étreinte fatale**, **deadlock** en anglais) est un phénomène qui peut survenir en programmation concurrente. L'interblocage se produit lorsque des processus concurrents s'attendent mutuellement. Un processus peut aussi s'attendre lui-même. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique provoquant le blocage du système.

### 3.2. Exemple



Le processus P1 utilise la ressource R2 qui est attendue par le processus P2 qui utilise la ressource R1, attendue par P1...

Soit 2 processus P1 et P2, soit 2 ressources R1 et R2. Initialement, les 2 ressources sont "libres" (i.e. utilisées par aucun processus). Le processus P1 commence son exécution (état élu), il demande la ressource R1. Il obtient satisfaction puisque R1 est libre, P1 est donc dans l'état "prêt". Pendant ce temps, le système a passé P2 à l'état élu : P2 commence son exécution et demande la ressource R2. Il obtient immédiatement R2 puisque cette ressource était libre. P2 repasse immédiatement à l'état élu et poursuit son exécution (P1 lui est toujours dans l'état prêt). P2 demande la ressource R1, il se retrouve dans un état bloqué puisque la ressource R1 a été attribuée à P1 : P1 est dans l'état prêt, il n'a pas eu l'occasion de libérer la ressource R1 puisqu'il n'a pas eu l'occasion d'utiliser R1 (pour utiliser R1, P1 doit être dans l'état élu). P2 étant bloqué (en attente de R1), le système passe P1 dans l'état élu et avant de libérer R1, il demande à utiliser R2. Problème : R2 n'a pas encore été libéré par P2, R2 n'est donc pas disponible, P1 se retrouve bloqué.

Résumons la situation à cet instant : P1 possède la ressource R1 et se trouve dans l'état bloqué (attente de R2), P2 possède la ressource R2 et se trouve dans l'état bloqué (attente de R1)

Pour que P1 puisse poursuivre son exécution, il faut que P2 libère la ressource R2, mais P2 ne peut pas poursuivre son exécution (et donc libérer R2) puisqu'il est bloqué dans l'attente de R1. Pour que P2 puisse poursuivre son exécution, il faut que P1 libère la ressource R1, mais P1 ne peut pas poursuivre son exécution (et donc libérer R1) puisqu'il est bloqué dans l'attente de R2. Bref, la situation est totalement bloquée !

Cette situation est qualifiée d'interblocage (deadlock en anglais).

Il existe plusieurs solutions permettant soit de mettre fin à un interblocage (cela passe par l'arrêt d'un des 2 processus fautifs) ou d'éviter les interblocage, mais ces solutions ne seront pas étudiées ici.