



2. Gestion des processus

2024-2025

Université Cheikh Anta Diop de Dakar (UCAD)

Faculté des Sciences et Techniques (FST) / Département Mathématiques-Informatique

Licence 2 Informatique

Dr Ousmane SADIO

□ Processus

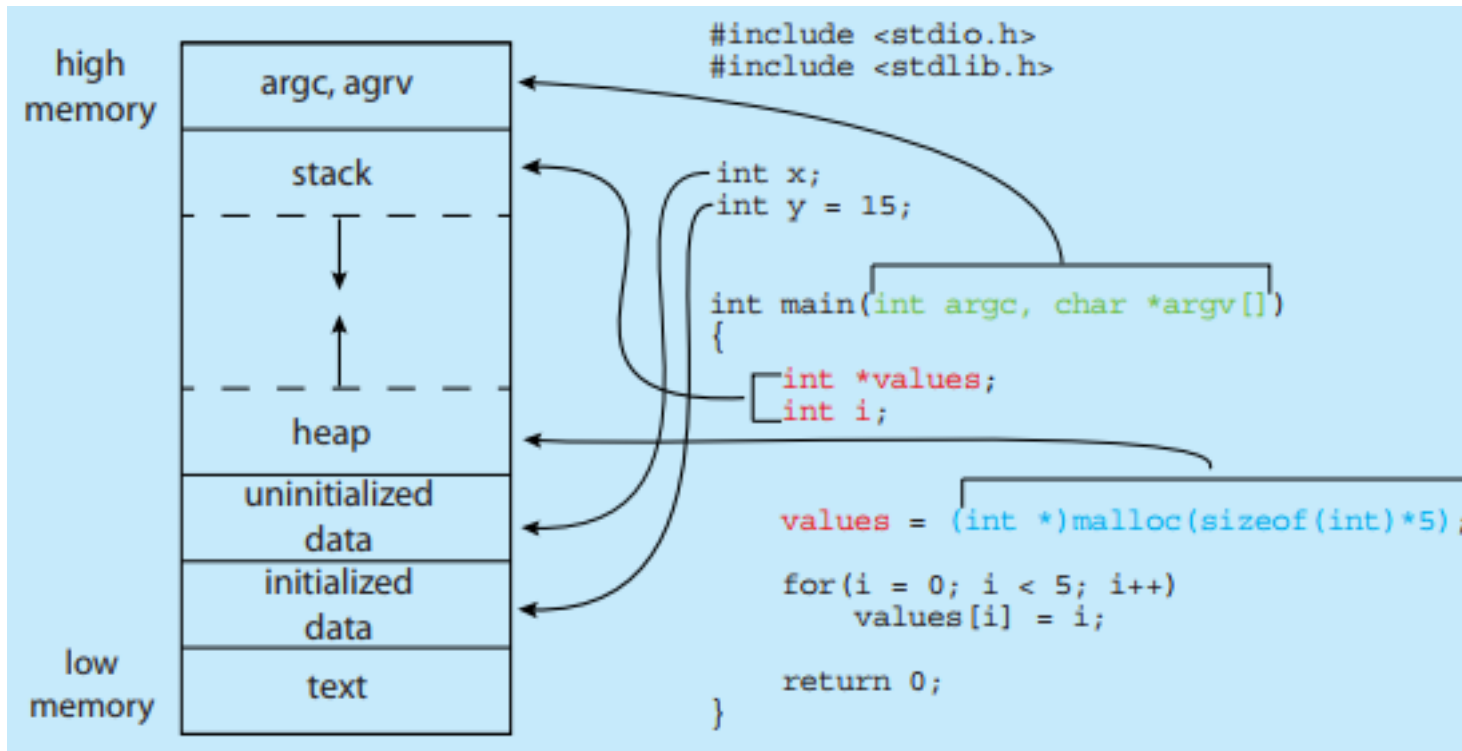
Un processus est un programme en cours d'exécution. Chaque processus possède **ses propres ressources**, telles que la mémoire, les fichiers ouverts, et l'état du processeur, ce qui permet à plusieurs programmes de s'exécuter de manière concurrente sur un même système.

- *Espace d'adressage* : chaque processus a son propre espace mémoire virtuel, isolé des autres processus.
- *Etat du processus* : Un processus peut être dans différents états pendant son exécution.
- *Contexte d'exécution* : garde l'état du processeur (registres, compteur ordinal, etc.) et d'autres informations nécessaires pour reprendre l'exécution du processus après une interruption.
- *Identifiant de processus (PID)* : Chaque processus est identifié par un numéro unique appelé PID (Process Identifier), qui permet au système d'exploitation de le gérer et de le suivre.
- *Hiérarchie de processus* : Les processus sont souvent organisés en une hiérarchie. Un processus parent peut créer des processus enfants, qui peuvent à leur tour créer d'autres processus.



□ Espace d'adressage

Chaque processus possède un espace d'adressage, c'est-à-dire un ensemble d'adresses mémoires dans lesquelles il peut lire et écrire. Cet espace est généralement divisé en plusieurs sections ou segments :



- *Segment de texte (text)* : mémoire pour le code exécutable.
- *Segment de données (data)* : mémoire pour les variables globales.
- *Tas (heap)* : mémoire qui est allouée de manière dynamique pendant la durée d'exécution du programme.
- *Pile (stack)* : stockage temporaire de données (paramètres, variables locales) lors de l'appel de fonctions.



❑ Bloc de contrôle du processus (PCB)

Un PCB (Process Control Block) contient de nombreux éléments d'information associés à un processus spécifique. Le PCB sert simplement de dépôt pour toutes les données nécessaires au démarrage ou au redémarrage d'un processus, ainsi que pour certaines données comptables.

- *Etat du processus* : contient l'état actuel du processus.
- *Compteur de programme* : indique l'adresse de la prochaine instruction à exécuter pour ce processus
- *Registres CPU* : différents mémoires de registre de la CPU.
- *Pile (stack)* : stockage temporaire de données (paramètres, variables locales) lors de l'appel de fonctions.

De plus, le noyau alloue pour chaque processus une structure appelée **zone u**, qui contient des données privées du processus uniquement manipulables par le noyau.



Contexte

Le contexte d'un processus est l'ensemble des données qui permettent de reprendre l'exécution d'un processus qui a été interrompu. Parmi ces données on citera notamment:

- L'état du processus
- Les valeurs des variables globales statiques ou dynamiques
- Son entrée dans la table des processus
- Sa zone u
- Sa pile et les zones de code et de données

L'exécution d'un processus se fait dans son contexte. Donc quand il y a changement de processus courant, il y a changement de contexte.

Le noyau et ses variables ne font partie du contexte d'aucun processus.



□ Etats d'un processus

Les états **prêt**, **élu** et **bloqué** d'un processus permettent au système d'exploitation de gérer efficacement l'exécution des programmes et l'allocation des ressources.

■ Processus Prêt (Ready)

Un processus est dit prêt lorsqu'il est prêt à être exécuté par le processeur, mais **il attend** que le système d'exploitation **lui alloue du temps CPU**.

- Le processus a toutes les ressources nécessaires pour s'exécuter, sauf le CPU.
- Il est placé dans la file d'attente des processus prêts (ready queue), en attendant d'être sélectionné par l'ordonnanceur (scheduler).
- Le passage à l'état prêt peut se produire après la création d'un nouveau processus, ou lorsqu'un processus bloqué redevient disponible.



□ Etats d'un processus

■ Processus Élu (Running)

Un processus est élu lorsqu'il **est en cours d'exécution** sur le CPU. C'est l'état actif du processus. Voici ce qui caractérise cet état :

- Le processus utilise activement le processeur pour exécuter ses instructions.
- Il peut quitter cet état pour plusieurs raisons :
 1. Il termine son exécution.
 2. Il est interrompu par le système d'exploitation (par exemple, à cause d'un quantum de temps écoulé).
 3. Il demande une ressource indisponible et passe à l'état bloqué.



❏ Etats d'un processus

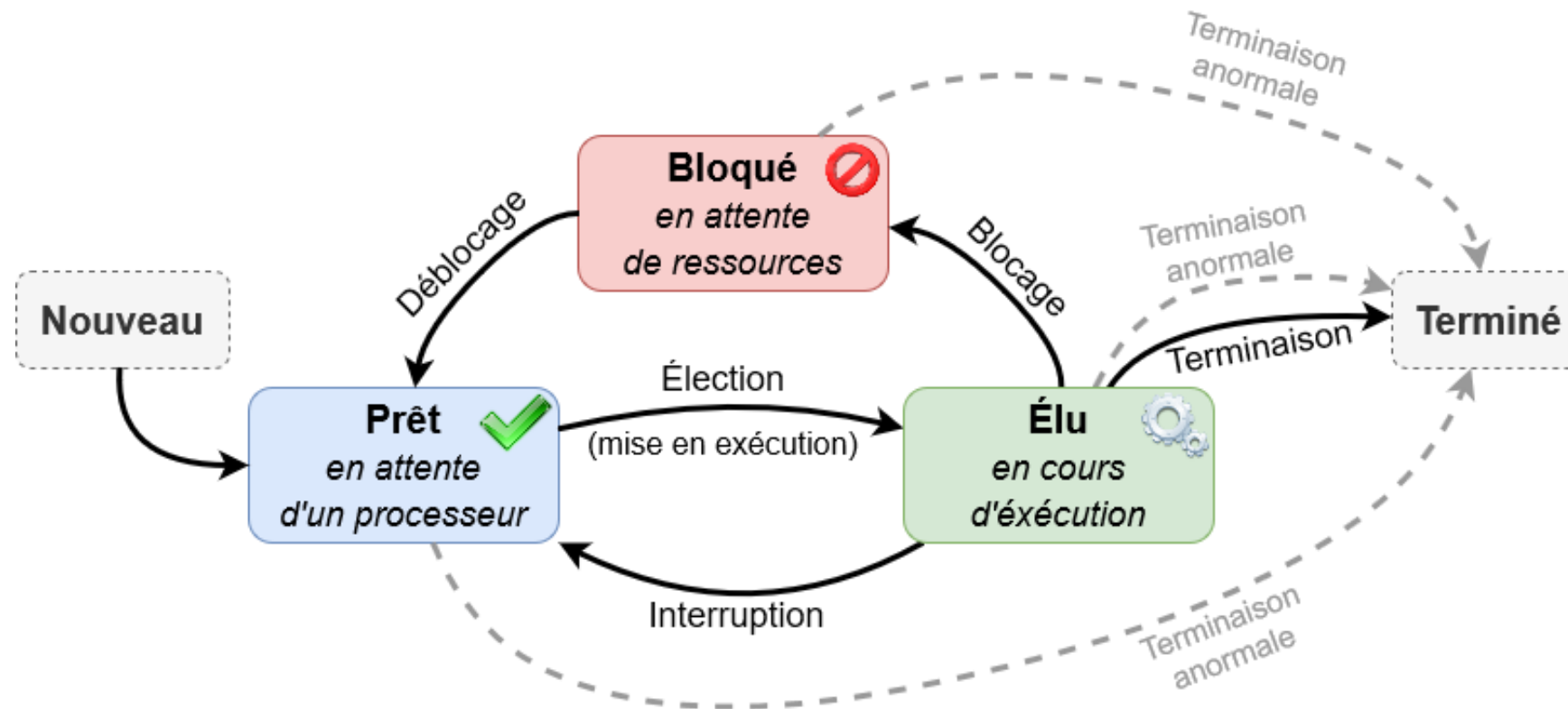
■ Processus Bloqué (Blocked/Waiting)

Un processus est bloqué lorsqu'il ne peut pas continuer son exécution car **il attend une ressource** ou un événement externe.

- Le processus est mis en attente jusqu'à ce que la ressource demandée soit disponible ou que l'événement attendu se produise.
- Il est retiré de la file des processus prêts et placé dans une file d'attente spécifique (par exemple, la file d'attente des E/S).
- Une fois la ressource disponible ou l'événement déclenché, le processus revient à l'état prêt.

❏ Etats : transition

Les processus passent d'un état à un autre en fonction des événements et des décisions du système d'exploitation. Voici les transitions possibles :





□ Ordonnancement

La fonction d'ordonnancement (en anglais *scheduling*) gère le partage du processeur entre les différents processus en attente pour s'exécuter. Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont :

- Point de vue Système
 - Maximiser l'utilisation de la CPU
 - Maximiser le débit (nombre de tâches complétées par unité de temps)
- Point de vue Utilisateur
 - Minimiser la latence
 - Temps de complétion d'une tâche (durée entre l'arrivée d'une tâche et sa complétion)
 - Temps de réponse (durée entre l'arrivée d'une tâche et le début de son exécution)
 - Minimiser l'attente (durée passée à attendre)



□ Ordonnancement

L'opération d'élection consiste à allouer le processeur à un processus. Selon si l'opération de réquisition du processeur est autorisée ou non, l'ordonnancement sera qualifié d'ordonnancement préemptif ou non préemptif :

- **Ordonnancement non préemptif** : la transition de l'état élu vers l'état prêt est **interdite** : un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque.
- **Ordonnancement préemptif** : la transition de l'état élu vers l'état prêt est **autorisée** : un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

❑ Ordonnancement non préemptif

Dans un système d'ordonnancement non préemptif, une fois qu'un processus a été alloué au CPU, il **conserve** le contrôle du processeur jusqu'à ce **qu'il termine son exécution** ou qu'il **passe à l'état bloqué**.

- Le système d'exploitation ne peut pas interrompre le processus en cours d'exécution pour en donner le contrôle à un autre processus.
- L'ordonnancement non préemptif est plus simple à implémenter car il n'y a pas besoin de gérer les interruptions fréquentes des processus.
- Utilisé dans des systèmes temps réel et certaines applications critiques où l'interruption d'un processus peut causer des problèmes.

Inconvénients :

- *Temps de réponse élevé* : si un processus long est en cours d'exécution, les autres processus doivent attendre longtemps, ce qui peut entraîner un temps de réponse élevé pour les processus interactifs.
- *Manque de réactivité* : le système peut sembler peu réactif, surtout dans les environnements où les processus ont des priorités différentes.

□ Ordonnancement préemptif

Dans un système d'ordonnancement préemptif, le système d'exploitation **peut interrompre** un processus en cours d'exécution pour donner le CPU à **un autre processus**, généralement en fonction de critères tels que la priorité, le temps d'exécution écoulé, ou d'autres politiques d'ordonnancement.

- Le système d'exploitation peut interrompre un processus à tout moment pour en exécuter un autre.
- Les systèmes préemptifs sont plus réactifs, car ils permettent de donner rapidement le CPU à des processus plus prioritaires ou interactifs.
- Les processus de haute priorité peuvent être exécutés immédiatement, même si un processus de priorité inférieure est en cours d'exécution.

Inconvénients :

- Les interruptions fréquentes entraînent des changements de contexte, ce qui peut augmenter la surcharge du système.
- L'implémentation de l'ordonnancement préemptif est plus complexe, car il faut gérer les interruptions et les changements de contexte de manière efficace.



❑ Ordonnancement non préemptif vs préemptif

| Critère | Ordonnancement Non Préemptif | Ordonnancement Préemptif |
|--------------|---|---|
| Interruption | Impossible | Possible |
| Réactivité | Faible | Élevée |
| Complexité | Plus simple à implémenter | Plus complexe à gérer |
| Utilisation | Systèmes temps réel, applications critiques | Systèmes d'exploitation multitâches, interactifs |
| Famine | Possible si un processus long bloque les autres | Peut être évité avec des algorithmes adaptés (ex: vieillissement des priorités) |
| Surcharge | Faible, car peu de changements de contexte | Élevée, en raison des interruptions et changements de contexte fréquents |

❑ Politiques d'ordonnancement

La politique d'ordonnancement est un des mécanismes utilisés par les systèmes d'exploitation pour décider de l'ordre d'exécution des processus sur le CPU. Chaque politique a ses propres caractéristiques et est adapté à des scénarios spécifiques.

- **FIFO** : *First-Come First-Serve (FCFS)* ou Premier arrivé, premier servi : c'est une politique à l'ancienneté, sans réquisition.
- **SJF** : *Shortest Job First* ou Plus court d'abord : l'unité centrale est allouée au processus de plus petit temps d'exécution.
- **SRTF** : *Shortest Remaining Time First* ou Temps restant le plus court d'abord : variante du SJF; on choisit le processus dont le temps d'exécution restant est le plus court.
- **RR** : *Round Robin* ou Politique d'ordonnancement par tourniquet : définit une tranche de temps appelée quantum.
- **PS** : *Priority Scheduling* ou Politique d'ordonnancement par priorités constantes : niveau de priorité constant est affecté à chaque processus et à un instant donné, l'élue est toujours celui de plus forte priorité.



❑ Politiques d'ordonnancement : First-Come First-Serve (FCFS)

L'ordonnancement First-Come First-Serve (FCFS) est le plus simple. Les processus sont exécutés dans l'ordre de leur arrivée dans la file d'attente. C'est un algorithme **non préemptif**.

Avantages :

- Simple à implémenter.
- Aucune surcharge due à la préemption.

Inconvénients :

- Peut entraîner un temps d'attente élevé pour les processus courts si un processus long est en tête de la file.
- Pas optimal pour minimiser le temps d'attente moyen.



❑ Politiques d'ordonnancement : First-Come First-Serve (FCFS)

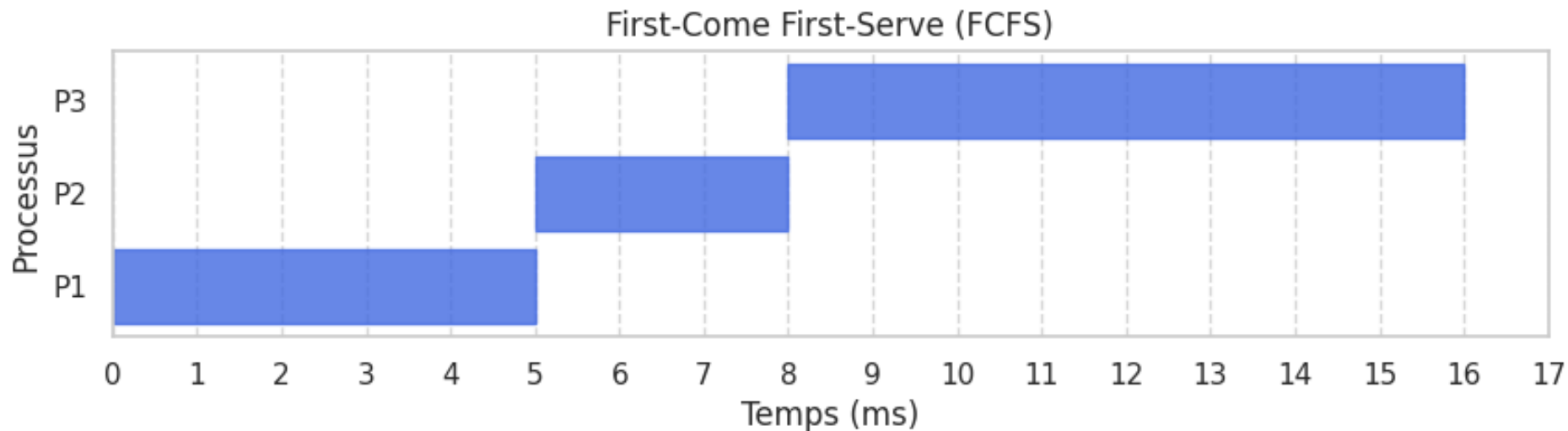
Supposons trois processus avec les temps d'exécution suivants :

P1 : 5 ms

P2 : 3 ms

P3 : 8 ms

L'ordre d'arrivée est P1, P2, P3.





❑ Politiques d'ordonnancement : Shortest Job First (SJF)

L'algorithme Shortest Job First (SJF) exécute en premier le processus ayant le plus court temps d'exécution. Cet algorithme peut être **préemptif** ou **non préemptif**.

Avantages :

- Minimise le temps d'attente moyen.
- Efficace pour les systèmes où les temps d'exécution sont connus à l'avance.

Inconvénients :

- Difficile à implémenter car les temps d'exécution ne sont pas toujours connus.
- Risque de famine pour les processus longs.

❑ Politiques d'ordonnancement : Shortest Job First (SJF)

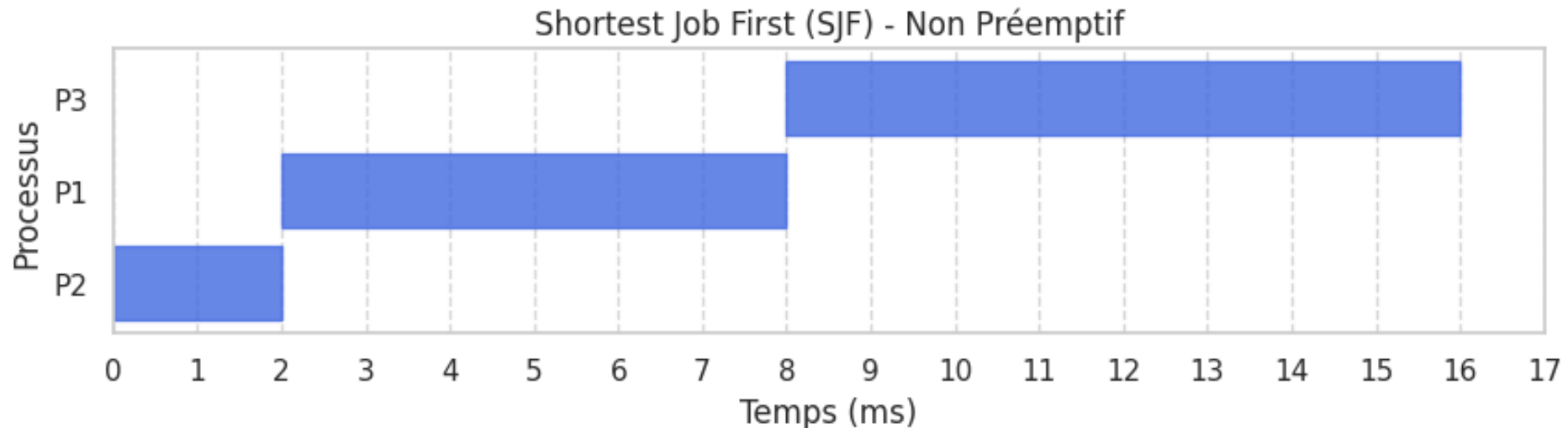
Processus et temps d'exécution :

P1 : 6 ms

P2 : 2 ms

P3 : 8 ms

En SJF non préemptif, l'ordre d'exécution sera P2 (2 ms), P1 (6 ms), P3 (8 ms).





❑ Politiques d'ordonnancement : Shortest Remaining Time First (SRTF)

L'algorithme Shortest Remaining Time First (SRTF) est la version **préemptive** du SJF. Le processus avec le temps restant le plus court est toujours sélectionné. Si un nouveau processus arrive avec un temps d'exécution plus court que le temps restant du processus en cours, ce dernier est interrompu.

Avantages :

- Minimise encore plus le temps d'attente moyen que SJF.
- Meilleur temps de réponse pour les courts processus.

Inconvénients :

- Complexité accrue due à la préemption.
- Nécessite de connaître les temps d'exécution à l'avance



❑ Politiques d'ordonnancement : Shortest Remaining Time First (SRTF)

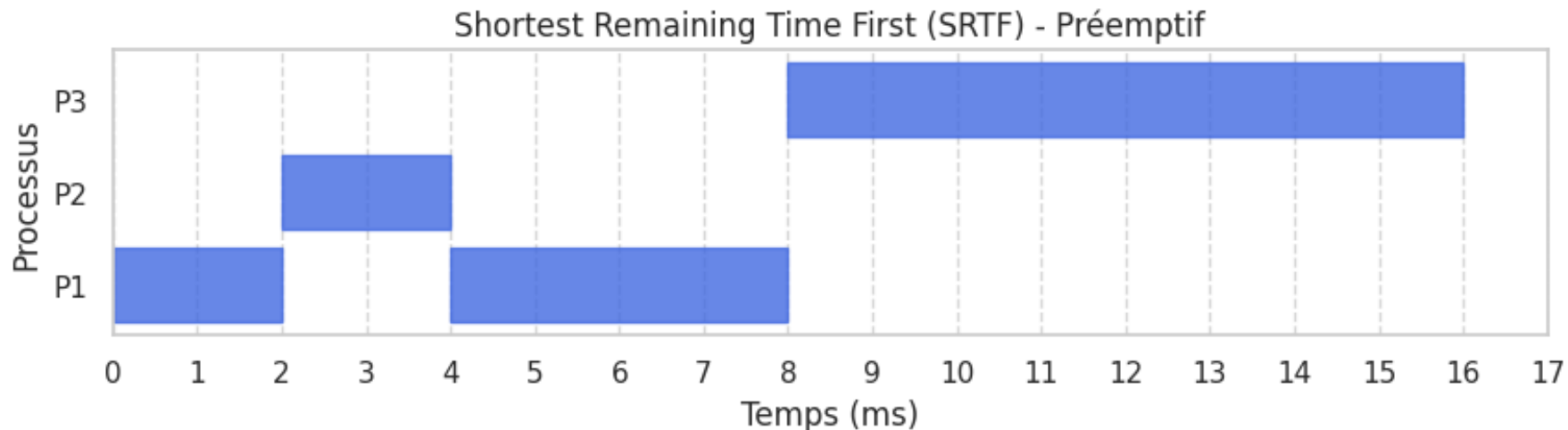
Processus et temps d'exécution :

P1 : 6 ms (arrive à $t = 0$)

P2 : 2 ms (arrive à $t = 2$)

P3 : 8 ms (arrive à $t = 4$)

À $t = 0$, P1 commence. À $t = 2$, P2 arrive avec un temps d'exécution plus court (2 ms) que le temps restant de P1 (4 ms). P1 est interrompu et P2 est exécuté. À $t = 4$, P2 se termine, et P1 reprend. À $t = 4$, P3 arrive, mais P1 a moins de temps restant (4 ms) que P3 (8 ms), donc P1 continue.





❑ Politiques d'ordonnancement : Round Robin (RR)

Pour l'algorithme de Round Robin (RR), chaque processus se voit attribuer un **quantum** de temps (time slice) pour s'exécuter. Si le processus ne se termine pas dans ce laps de temps, il est remis en fin de file d'attente, et le CPU passe au processus suivant. C'est un algorithme qui est donc **préemptif**.

Avantages :

- Equitable, car chaque processus reçoit une part égale du CPU, donc évite la famine.
- Convient aux systèmes interactifs.

Inconvénients :

- Performance dépend fortement du choix du quantum.
- Augmente les changements de contexte.
- Mauvais pour les processus longs.



❑ Politiques d'ordonnancement : Round Robin (RR)

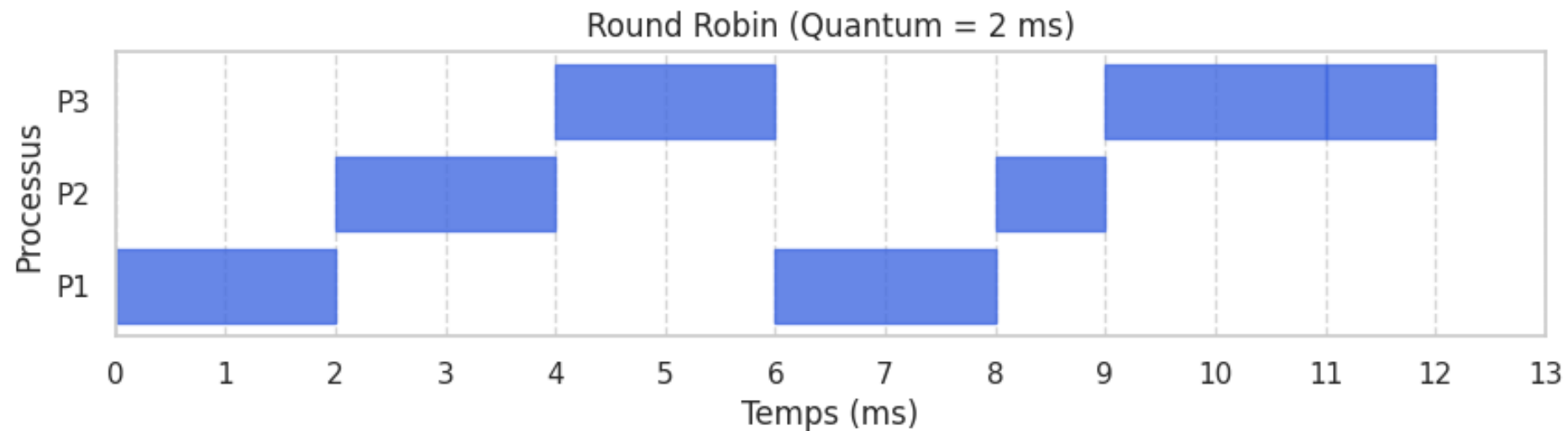
Processus et temps d'exécution :

P1 : 4 ms

P2 : 3 ms

P3 : 5 ms

Avec un quantum de 2 ms.





❑ Politiques d'ordonnancement : Priority Scheduling (PS)

Pour l'algorithme Priority Scheduling (PS), chaque processus se voit attribuer une **priorité**. Le processus avec la priorité la plus élevée est exécuté en premier. Cet algorithme peut être **préemptif** ou **non préemptif**.

Avantages :

- Permet de donner la priorité à des processus critiques.

Inconvénients :

- Risque de famine pour les processus de faible priorité.
- Nécessite un mécanisme complexe pour gérer les priorités dynamiques.



❑ Politiques d'ordonnancement : Priority Scheduling (PS)

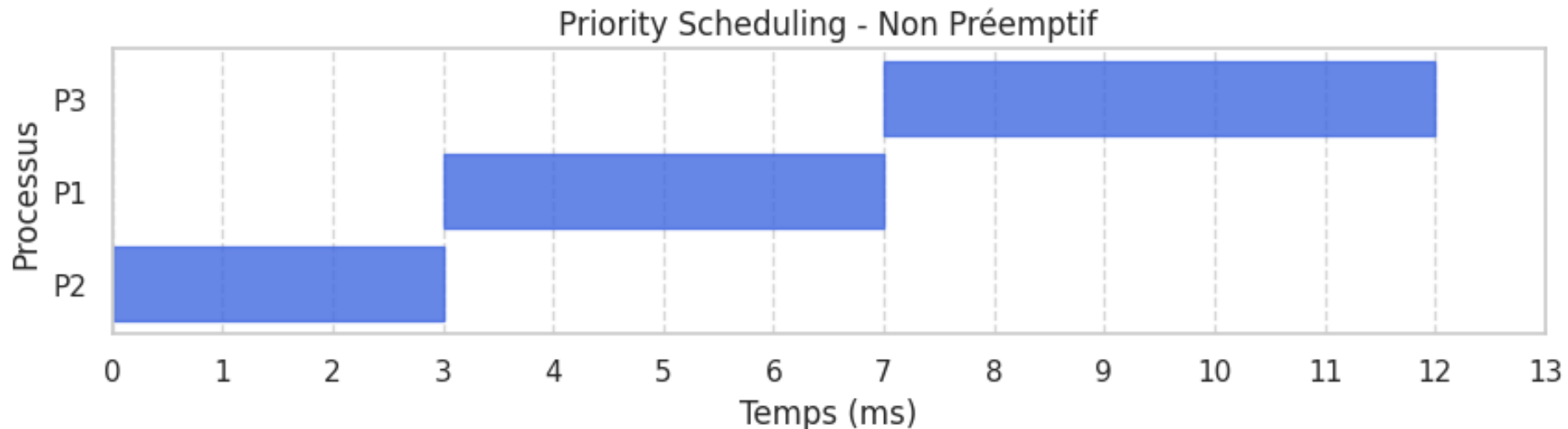
Processus, temps d'exécution et priorité :

P1 : 4 ms, priorité 2

P2 : 3 ms, priorité 1 (le plus prioritaire)

P3 : 5 ms, priorité 3

En mode non préemptif, l'ordre d'exécution sera P2, P1, P3.



□ Threads

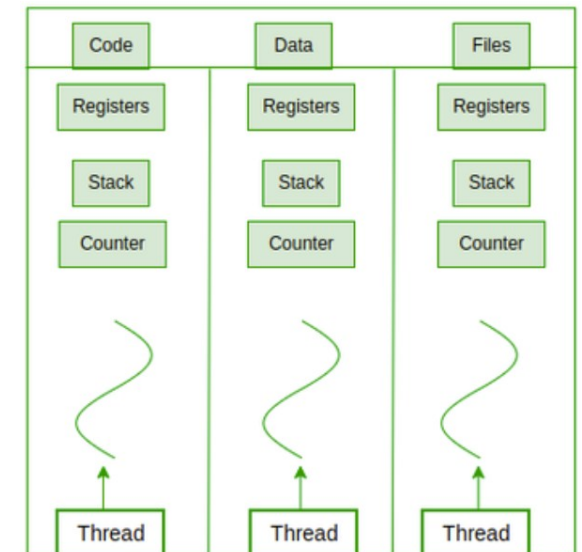
Un **thread** (ou fil d'exécution) est une séquence d'instructions qui s'exécute à l'intérieur d'un processus. Il est exécuté de manière indépendante.

Chaque thread présent dans un processus à son propre contexte d'exécution qui se compose de :

- Compteur ordinal : contient le numéro de l'instruction suivante à exécuter.
- Registres : contient les valeurs intermédiaires du CPU lors de ses opérations.
- Pile (stack) : contient les variables locales, les appels de fonction et les adresses de retour du thread.

Cependant, tous les threads dans un même processus se partagent ces éléments :

- Mémoire
- Fichiers ouverts
- Variables globales





❏ Threads

Les threads dans le système d'exploitation offrent de multiples avantages et améliorent les performances globales du système.

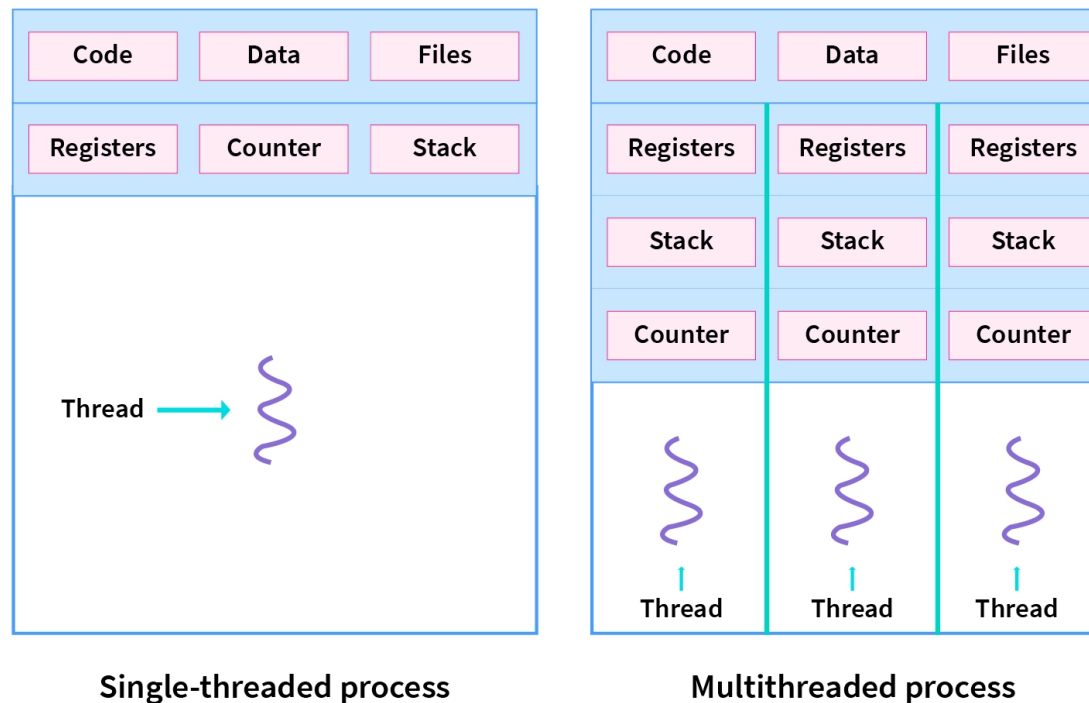
- Comme les threads utilisent les mêmes données et le même code, le coût opérationnel entre les threads est faible.
- La création et l'arrêt d'un thread sont plus rapides que la création ou l'arrêt d'un processus.
- Le changement de contexte est plus rapide dans les threads que dans les processus.

| Caractéristique | Processus | Thread |
|------------------------|---|--|
| Espace mémoire | Indépendant | Partagé avec les autres threads du processus |
| Création | Coûteuse en ressources (duplication de mémoire, allocation de structures) | Plus légère et rapide |
| Communication | Via IPC (Inter Process Communication) | Accès direct aux mêmes données |
| Changement de contexte | Coût élevé | Moins coûteux |

❏ Multithreading

Les **processus traditionnels** ont un **seul fil de contrôle**, il y a un compteur de programme et une seule séquence d'instructions qui peut être exécutée à tout moment.

Dans le **multithreading**, l'idée est de diviser un processus unique en **plusieurs threads** au lieu de créer un tout nouveau processus. Le multithreading est utilisé pour **réaliser le parallélisme** et améliorer les performances et la réactivité des applications, en particulier sur les **systèmes multicœurs**.





❑ Multithreading : motivation

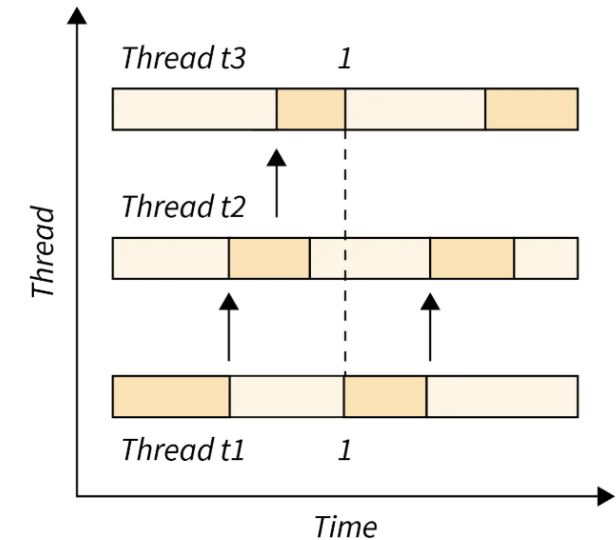
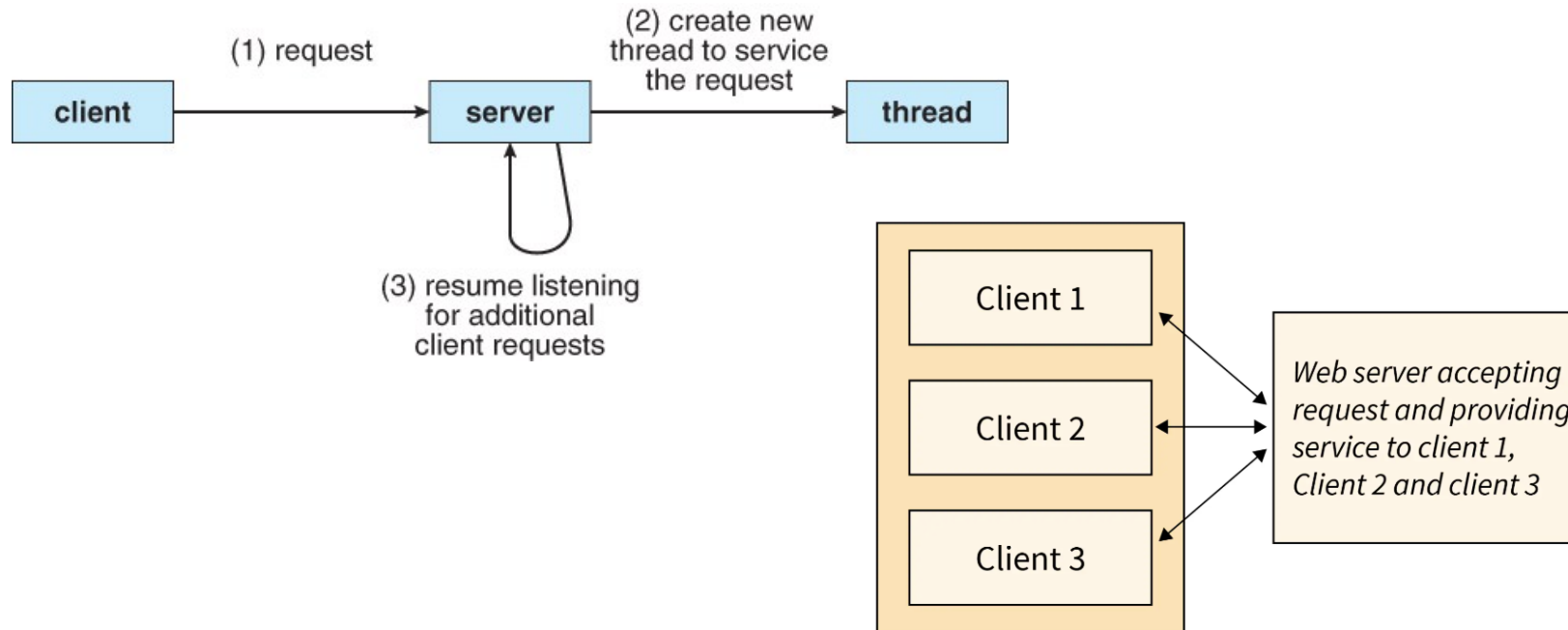
Les threads sont très utiles dans la programmation moderne lorsqu'un processus doit effectuer plusieurs tâches indépendamment les unes des autres.



Par exemple dans un **traitement de texte** :

1. Un processus d'arrière-plan peut vérifier l'orthographe et la grammaire,
2. Tandis qu'un processus d'avant-plan traite les entrées de l'utilisateur (frappes au clavier),
3. Tandis qu'un troisième processus charge des images à partir du disque dur,
4. Et qu'un quatrième effectue des sauvegardes automatiques périodiques du fichier en cours d'édition.

❑ Multithreading : motivation

Un autre exemple est celui d'un **serveur web**, les fils d'exécution multiples permettent de répondre simultanément à plusieurs demandes, sans avoir à traiter les demandes de manière séquentielle ou à lancer des processus distincts pour chaque demande entrante.



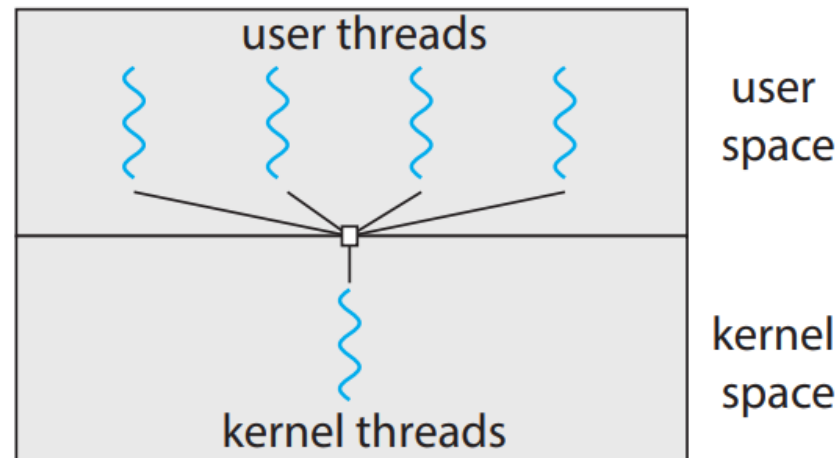
 Thread allotted CPU
 Thread sleeping

❑ Modèles de multithreading

Le support des threads peut être fourni soit au niveau de l'utilisateur, pour les threads utilisateur, soit par le noyau, pour les threads noyau.

■ Modèle Many-to-One

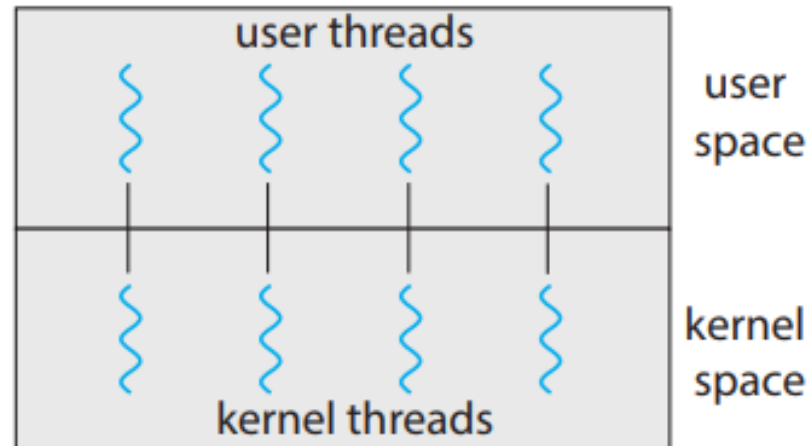
Fait correspondre plusieurs threads utilisateur à un unique thread noyau. Cependant lorsqu'un thread utilisateur fait un appel système bloquant (read, write...), tous les autres threads sont également bloqués. En outre, les threads sont incapables de s'exécuter en parallèle sur des systèmes multicœurs du fait que l'accès au noyau se fait par un seul thread à la fois.



❑ Modèles de multithreading

■ Modèle One-to-One

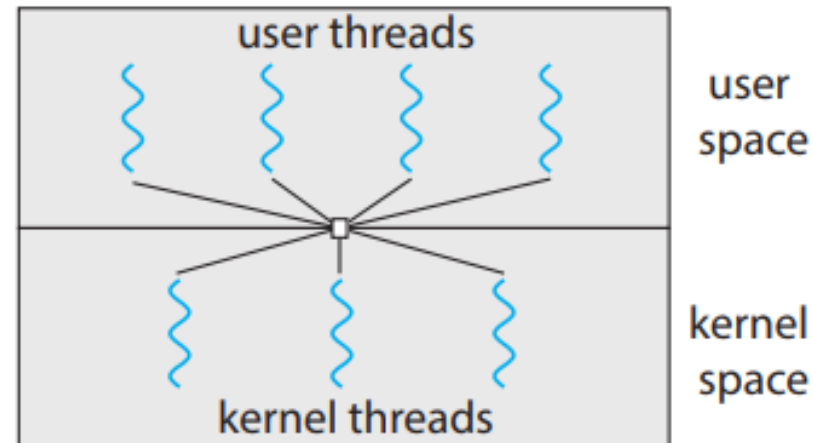
Fait correspondre chaque thread utilisateur à un thread noyau. Ce modèle permet à plusieurs threads de s'exécuter, sans bloquer les autres, en parallèle sur des systèmes multicœurs. Cependant un grand nombre de threads du noyau peut dégrader les performances d'un système. Néanmoins c'est ce modèle qui est implémenté dans les systèmes Linux et Windows.



❑ Modèles de multithreading

■ Modèle Many-to-Many

Multiplexe plusieurs threads utilisateur en un nombre inférieur ou égal de threads noyau. Ce nombre dépend de l'application ou de l'architecture du processeur. Lorsqu'un thread est bloqué (via un appel système bloquant), le noyau crée un autre thread pour prendre le relais. Cependant, dans la pratique, il est difficile à mettre en œuvre ce modèle.



❑ Avantages et inconvénients du Multithreading

Avantages du Multithreading

- Performance : exploite les capacités des processeurs multi-cœurs en exécutant plusieurs threads simultanément.
- Réactivité : permet à une application de rester réactive (par exemple, une interface graphique) tout en effectuant des tâches en arrière-plan.
- Efficacité : les threads partagent les ressources du processus, ce qui réduit la surcharge liée à la création et à la commutation de contexte.

Inconvénients du Multithreading

- Complexité : la gestion des threads peut être complexe, en particulier pour éviter les problèmes de concurrence.
- Problèmes de concurrence : les threads partagent des ressources, ce qui peut entraîner des conditions de course (race conditions), des interblocages (deadlocks), ou des incohérences de données.

❑ Synchronisation entre processus

La synchronisation est nécessaire lorsque plusieurs processus accèdent à des ressources partagées (mémoire, fichiers, périphériques, etc.) pour éviter des problèmes comme les conditions de concurrence (race conditions) ou les incohérences de données.

■ Mutex

Un mutex (ou verrou d'exclusion mutuelle) est un mécanisme de synchronisation utilisé pour protéger l'accès à une ressource partagée en garantissant qu'un seul processus (ou thread) à la fois peut accéder à cette ressource.

- *Verrouillage* (Lock) : un processus tente de verrouiller le mutex. Si le mutex est déjà verrouillé par un autre processus, le processus appelant est mis en attente jusqu'à ce que le mutex soit libéré.
- *Déverrouillage* (Unlock) : un processus libère le mutex, permettant à un autre processus de le verrouiller.

Les mutex sont souvent utilisés dans les programmes multithreadés pour éviter les conditions de concurrence en garantissant que seulement un thread à la fois peut exécuter une section critique de code.



□ Synchronisation entre processus

■ Sémaphores

Un sémaphore est un mécanisme de synchronisation utilisé pour contrôler l'accès à une ressource partagée entre plusieurs processus. Il maintient un compteur qui représente le nombre de ressources disponibles. Les opérations principales sur un sémaphore sont :

- *P (ou wait)* : décrémente le compteur du sémaphore. Si le compteur est déjà à zéro, le processus est bloqué jusqu'à ce qu'une ressource soit libérée.
- *V (ou signal)* : incrémente le compteur du sémaphore, libérant ainsi une ressource. Si des processus étaient en attente, l'un d'eux est réveillé.

Les sémaphores peuvent être utilisés pour résoudre des problèmes classiques de synchronisation.



❑ Communication inter-processus (IPC)

La communication interprocessus (IPC – Inter Process Communication) est un mécanisme qui permet à plusieurs processus d'un système d'exploitation de **s'échanger des informations** et de **se synchroniser**. Ces processus peuvent s'exécuter sur le même ordinateur ou sur des machines différentes connectées via un réseau.

Les processus fonctionnent généralement de manière isolée dans un système d'exploitation, chacun disposant de son propre espace mémoire. Cependant, dans de nombreuses applications, il est essentiel que les processus coopèrent en partageant des données ou en coordonnant leurs actions. L'IPC permet donc de :

- Echanger des données entre processus.
- Synchroniser l'exécution des processus.
- Optimiser l'utilisation des ressources en évitant la duplication inutile de données.

❑ Communication inter-processus (IPC)

■ Tubes (pipes)

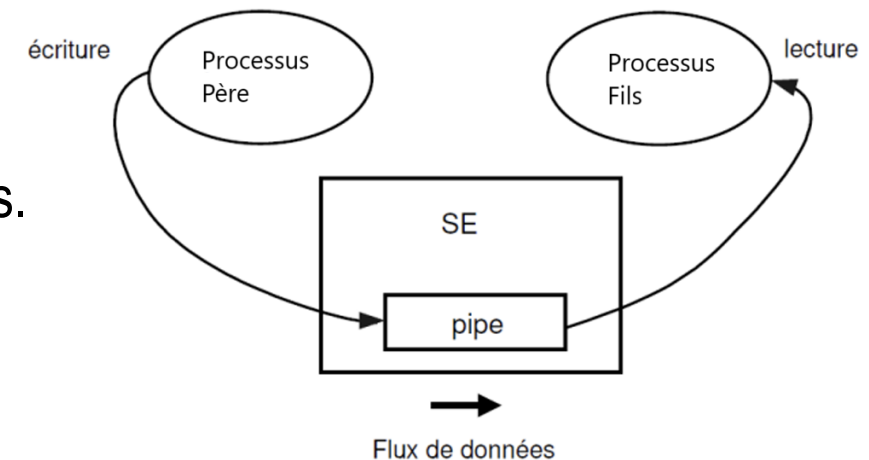
Un tube est un **canal de communication unidirectionnel** entre deux processus. Un tube a une extrémité de lecture et une extrémité d'écriture **créé en mémoire**. Un processus peut alors écrire sur l'extrémité d'écriture ; ces données seront mises en mémoire tampon jusqu'à ce qu'elles soient lues par un autre processus depuis l'extrémité de lecture du tube.

- Un processus écrit dans le tube, et un autre lit à partir du tube.
- Les tubes sont souvent utilisés pour la communication entre processus parent et enfant.

Limitations

- Communication unidirectionnelle.
- Seuls les **processus apparentés** peuvent utiliser des tubes anonymes.

Cas d'utilisation : Redirection de sortie entre commandes shell (par exemple : `ls | grep file`).



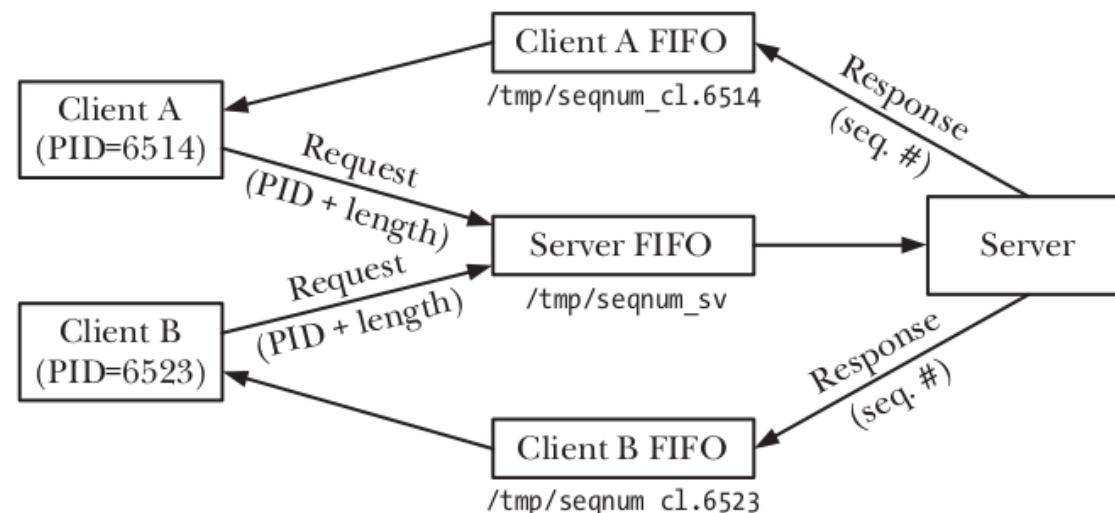
❑ Communication inter-processus (IPC)

▪ Tubes nommés (FIFO)

Un tube nommé est un **fichier spécial** dans le système de fichiers qui agit comme un tube.

- Les processus ouvrent le tube nommé pour lire ou écrire des données.
- Contrairement aux tubes anonymes, les tubes nommés permettent la **communication** entre **processus non apparentés**.

Cas d'utilisation : Communication entre processus indépendants.



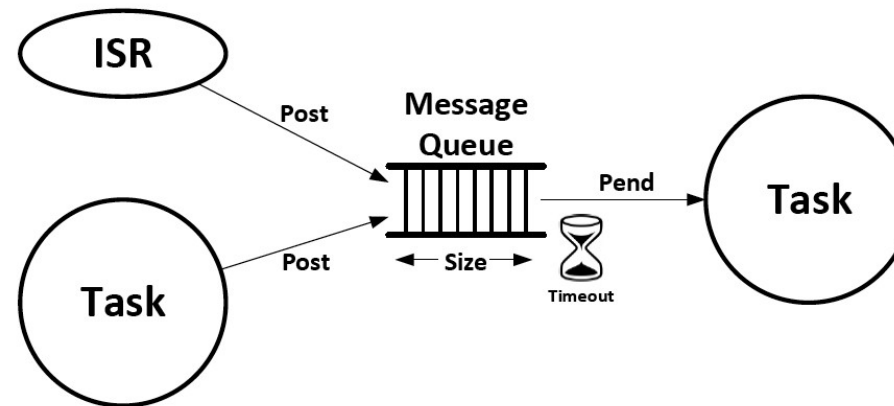
❑ Communication inter-processus (IPC)

▪ Files de messages (Message Queues)

Les files de messages permettent aux processus d'échanger **des messages** sous forme de **blocs de données**.

- Chaque message a un type associé, ce qui permet aux processus de lire sélectivement les messages.
- Les messages sont stockés dans une file gérée par le noyau.

Cas d'utilisation : Communication asynchrone entre processus.



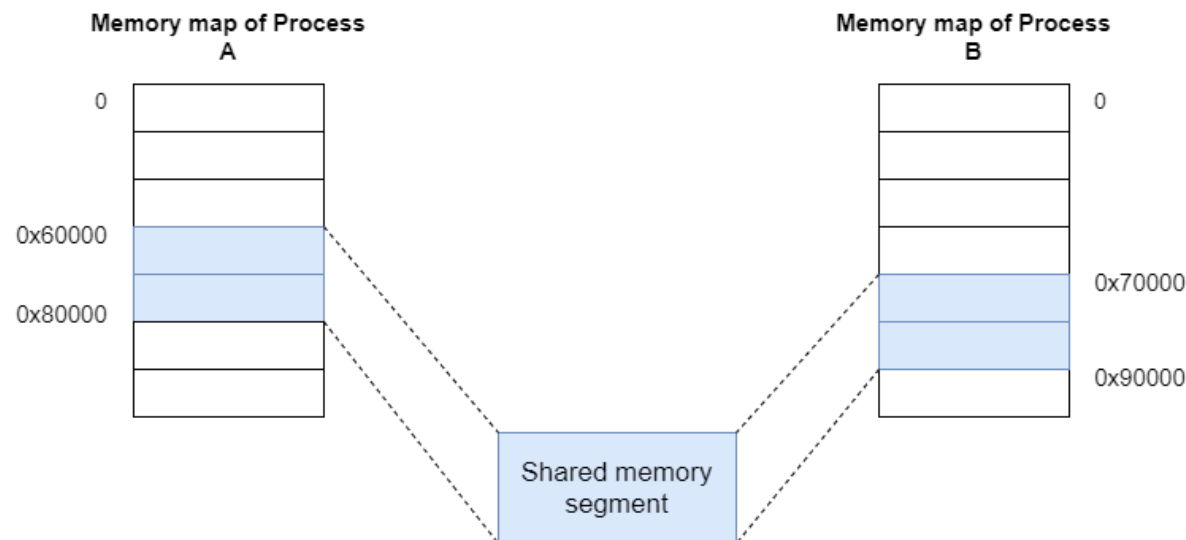
❑ Communication inter-processus (IPC)

■ Mémoire partagée (Shared Memory)

La mémoire partagée permet à plusieurs processus de **partager une région de mémoire**.

- Un processus crée une région de mémoire partagée, et d'autres processus s'y attachent.
- Les processus peuvent lire et écrire directement dans cette mémoire.
- Nécessite une synchronisation explicite (par exemple, avec des sémaphores ou des mutex).

Cas d'utilisation : Partage de données à haute performance entre processus.



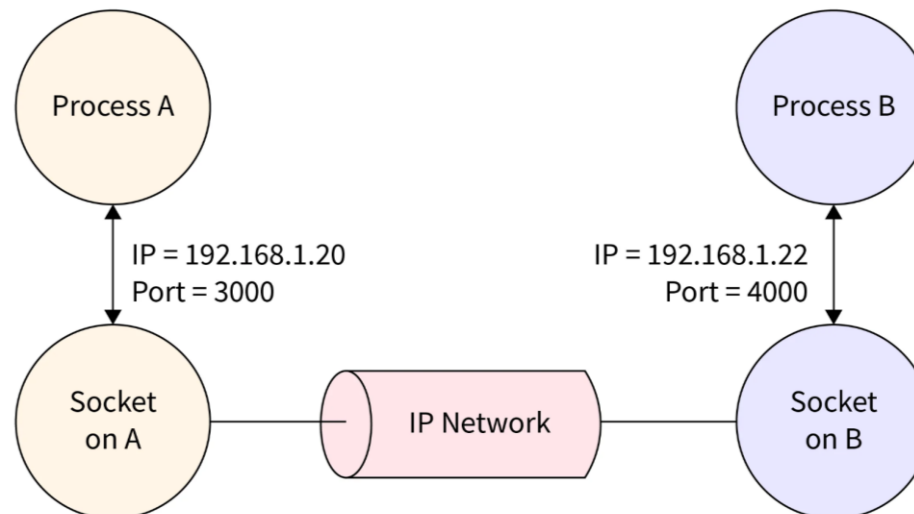
❑ Communication inter-processus (IPC)

▪ Sockets

Les sockets permettent la communication entre processus sur la même machine (sockets Unix) ou sur des machines différentes (sockets réseau).

- Un processus crée un socket et écoute les connexions.
- Un autre processus se connecte au socket pour échanger des données.

Cas d'utilisation : Communication réseau ou IPC local.



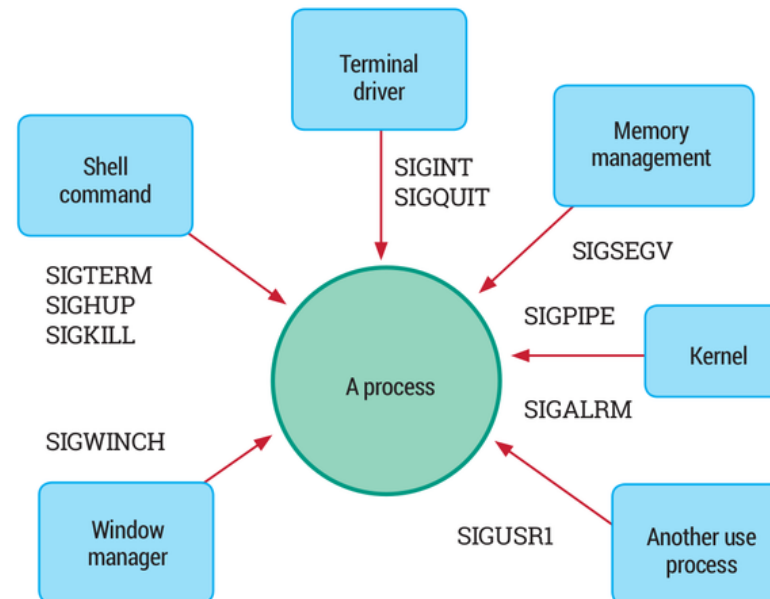
❑ Communication inter-processus (IPC)

▪ Signaux (Signals)

Les signaux sont des notifications asynchrones envoyées à un processus pour l'informer d'un événement.

- Un processus peut envoyer un signal à un autre processus (exemple : SIGKILL pour terminer un processus).
- Les processus peuvent capturer et gérer des signaux pour réagir à des événements.

Cas d'utilisation : Gestion d'événements asynchrones (par exemple, interruption d'un programme).





❑ Implémentation sous Linux avec le langage C

Pour accéder à ces services ou ressources, on utilise des fonctions qui permettent de communiquer avec le noyau. Ces fonctions sont appelées des appels-systèmes. Il existe une centaine d'appels-système sous Unix.

- Généralement, un processus utilisateur entre dans le mode noyau quand il effectue un appel système.
- Le langage C a été créé spécialement pour la programmation système, plus précisément pour le développement de système d'exploitation comme Unix. Il est donc particulièrement adapté à ce type de programmation.
- Le compilateur C utilisé sous Linux est **gcc** (Gnu Compiler Collection), ce dernier est intégré dans toutes les distributions Linux.



❑ Création d'un nouveau processus

Les processus sont organisés en hiérarchie, chaque processus doit être lancé par un autre.

▪ La fonction `fork()`

L'appel système **fork** est utilisé pour créer un nouveau processus, appelé **processus enfant**, qui est une **copie exacte** du **processus parent** (le processus qui appelle fork) mais qui a son propre espace d'adressage.

- Duplication du processus : lorsque fork est appelé, le système d'exploitation crée une copie du processus parent. Cela inclut la duplication de la mémoire, des registres du CPU, des descripteurs de fichiers, etc.
- Valeur de retour : fork retourne une valeur différente dans le processus parent et dans le processus enfant :
 - Dans le **processus parent**, fork retourne le **PID du processus enfant**.
 - Dans le **processus enfant**, fork retourne **0**.
 - En cas **d'erreur**, fork retourne **-1**.



❑ Création d'un nouveau processus

- La fonction `fork()`

Le prototype de la fonction `fork()` est déclarée ainsi :

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

La fonction `fork()` retourne une valeur de type **pid_t** (qu'on peut assimiler à un int), il est déclaré dans `<sys/types.h>`.



❑ Création d'un nouveau processus

- Autres fonctions

La fonction **getpid** retourne le **PID** du processus appelant.

```
pid_t getpid(void);
```

La fonction **getppid** retourne le **PPID** du processus appelant.

```
pid_t getppid(void);
```

La fonction **getuid** retourne l'**UID** du processus appelant.

```
pid_t getuid(void);
```

La fonction **getgid** retourne le **GID** du processus appelant.

```
pid_t getgid(void);
```



```
#include <stdlib.h>           // Pour EXIT_SUCCESS et EXIT_FAILURE
#include <stdio.h>             // Pour fprintf()
#include <unistd.h>            // Pour fork()
#include <errno.h>             // Pour perror() et errno
#include <sys/types.h>         // Pour pid_t

int main (void)
{
    pid_t pid_fils;
    // Creation d'un processus
    pid_fils = fork();

    if (pid_fils == -1) {
        fprintf(stderr, "fork () impossible, errno=%d\n", errno);
        return EXIT_FAILURE;
    }

    if (pid_fils == 0) { // processus fils
        fprintf(stdout, "Fils : PID=%d, PPID=%d\n", (int) getpid(), (int) getppid());
        return EXIT_SUCCESS;
    }
    else { // processus père
        fprintf(stdout, "Pere : PID=%d, PPID=%d, PID fils=%d\n", (int) getpid(), (int) getppid(), (int) pid_fils);
        return EXIT_SUCCESS;
    }
}
```




❑ Création d'un nouveau processus

- La fonction `fork()`

Lors de son exécution, le programme fournit les informations suivantes :

```
ghost@xps:~$ ./exemple_fork
Pere : PID=1767, PPID=402, PID fils=1768
Fils : PID=1768, PPID=401
```



❏ Création d'un nouveau processus

■ Exemple d'usage de la fonction `fork()`

1. Le Shell utilise `fork()` pour exécuter les programmes invoqués depuis la ligne de commande.
2. Le serveur web apache utilisent `fork()` pour créer de multiples processus de serveur, chacun d'entre eux traitant les requêtes dans son propre espace d'adressage. Si l'un d'entre eux meurt ou crée une fuite de mémoire, les autres ne sont pas affectés, il fonctionne donc comme un mécanisme de tolérance aux pannes.
3. Google Chrome utilise `fork()` pour traiter chaque page comme un processus distinct. Cela permet d'éviter que le code d'une page web (ex: javascript) ne fasse tomber tout le navigateur.
4. Fork est utilisé pour dupliquer des processus dans certains programmes parallèles. Notez que c'est différent de l'utilisation des threads, qui eux partagent un même espace d'adressage.
5. Les langages de script utilisent indirectement la fonction `fork()` pour lancer des processus enfants. Une commande comme `subprocess.Popen()` en Python, vous créez un processus fils et lisez sa sortie.
6. Un server FTP se base sur les permissions de l'utilisateur. Une fois l'identité de l'utilisateur connu, le serveur crée un `fork()` et change ses permissions pour prendre les permissions de l'utilisateur.

❑ Exécution des programmes

La famille des fonctions **exec** permet de remplacer l'image du processus actuel par un nouveau programme. Contrairement à `fork`, ces fonctions ne créent pas un nouveau processus, mais remplacent le code, les données, le tas, et la pile du processus actuel par ceux du nouveau programme.

■ La fonction **execve** ()

execve est une fonction puissante qui permet de remplacer l'image du processus actuel par un nouveau programme tout en spécifiant un environnement personnalisé. Son prototype est le suivant :

```
int execve (const char * filename, const char * argv [], const char * envp []);
```

- La variable *filename* doit contenir le chemin d'accès au programme à lancer.
- Le tableau *argv*[] contient des chaînes de caractères correspondant aux arguments.
- La première chaîne doit contenir le nom de l'application à lancer, et la dernière chaîne est égale à NULL.
- Le tableau *envp*[] est un tableau de chaînes déclarant les variables d'environnement personnalisées.



❑ Exécution des programmes

- La fonction `execv()`

L'appel système **execv** est utilisé pour remplacer l'image du processus actuel par un nouveau programme. Elle fonctionne comme `execve()`, mais l'environnement est directement transmis par l'intermédiaire de la variable externe `environ`, sans avoir besoin d'être passé explicitement en argument durant l'appel. La fonction `execv()` dispose du prototype suivant :

```
int execv (const char * filename, const char * argv []);
```



❑ Exécution des programmes

■ La fonction `execv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    // Tableau de char contenant les arguments (ps -e -f)
    char *arguments[] = { "ps", "-e", "-f", NULL };
    // Lancement de la commande
    if (execv("/bin/ps", arguments) == -1) {
        perror("execv");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```



❑ Exécution des programmes

- La fonction `execv()`

Lors de son exécution, ce programme fournit les informations suivantes :

```
ghost@xps:~$ ./exemple_execv
```

| UID | PID | PPID | C | STIME | TTY | TIME | CMD |
|------|-----|------|---|-------|-----|----------|--|
| root | 1 | 0 | 1 | 22:13 | ? | 00:00:00 | /sbin/init |
| root | 2 | 1 | 0 | 22:13 | ? | 00:00:00 | /init |
| root | 7 | 2 | 0 | 22:13 | ? | 00:00:00 | plan9 --control-socket 7 --log-level 4 --server-fd 8 --pipe-fd 10 --log-truncate |
| root | 36 | 1 | 0 | 22:13 | ? | 00:00:00 | /lib/systemd/systemd-journald |
| root | 57 | 1 | 0 | 22:13 | ? | 00:00:00 | /lib/systemd/systemd-udev |
| root | 74 | 1 | 0 | 22:13 | ? | 00:00:00 | snapparse /var/lib/snapparse/snaps/bare_5.snap /snap/bare/5 -o ro,nodev,allow_other,su |



❑ Exécution des programmes

■ Combinaison de `fork` et `execv`

En pratique, `fork` et `execv` sont souvent utilisés ensemble pour créer un nouveau processus et exécuter un programme différent dans ce processus enfant. Ces appels système sont essentiels pour la gestion des processus dans les systèmes d'exploitation de type Unix, et leur compréhension est cruciale pour tout développeur système.

- *Espace d'adressage* : Après un `fork`, le processus enfant a une copie de l'espace d'adressage du parent. Cependant, grâce à la technique de copy-on-write, la mémoire n'est réellement dupliquée que si l'un des processus modifie une page de mémoire.
- *Synchronisation* : Le processus parent peut utiliser `wait` ou `waitpid` pour attendre la fin de l'exécution du processus enfant.
- *Environnement* : Le processus enfant hérite de l'environnement du parent, mais `execv` peut remplacer cet environnement si nécessaire.



```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {    // Processus enfant
        char *args[] = {"/bin/ls", "-l", NULL};
        if (execv("/bin/ls", args) == -1) {
            perror("execv");
        }
    } else if (pid > 0) {    // Processus parent
        wait(NULL);    // Attend que l'enfant termine
        printf("L'enfant a terminé\n");
    } else {
        // Erreur
        perror("fork");
    }
    return 0;
}
```