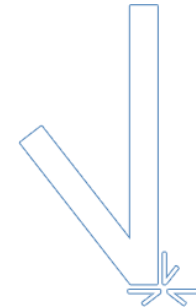


Systemes d'Exploitation



Objectif :

Comprendre comment un système
d'exploitation fonctionne et
comment l'utiliser

Systemes d'Exploitation

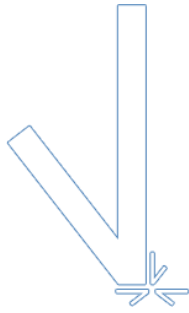


Programmes et Processus

(cours précédent : Systemes d'Exploitation – Gestion de la Mémoire)

I. Systèmes d'Exploitation

I.9. Programmes et Processus



I. Système d'Exploitation

1. Sommaire
2. Introduction
3. Noyau et Pilotes
4. Utilisateurs et Sessions
5. Système de Fichier
6. Permissions et Droits
7. Shell et Utilitaires
8. Gestion de la Mémoire
9. Programmes et Processus
 - a. Processus
 - b. Threads
 - c. Processus Enfant
 - d. IPC
 - e. Ordonnancement
10. Variables d'Environnement
11. Scripts Shell
12. Gestion des Paquets

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus

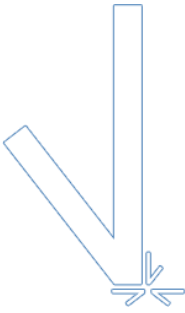


Processus

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



- Un **programme** est un **ensemble d'instructions inactives** stockées dans une mémoire volatile ou non volatile.
- ▶ Un **processus** est un programme chargé dans la mémoire [virtuelle] qui est **en cours d'exécution**.
- ♥ Le **noyau du système d'exploitation** est responsable du **chargement des programmes** dans la mémoire [virtuelle] et de leur exécution.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus

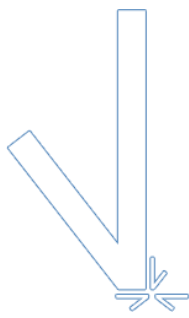


- ⚙️ Un processus unique peut engendrer (**spawn**) d'autres processus :
 - 🏃 En demandant **au noyau de charger et d'exécuter d'autres programmes** ;
 - ⚙️ En se **dupliquant en mémoire**.
- 👤 Les processus créés grâce à d'autres processus sont appelés **processus enfants (child processes)**.
- 👤 Lorsqu'un processus effectue un **appel système** pour se dupliquer, le nouveau processus est appelé un **fork**.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



📦 L'espace mémoire alloué à un **processus** est divisé en plusieurs parties : les **segments de mémoire**. Ces segments de mémoire sont :

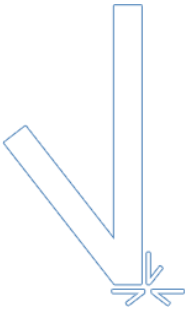
📖 Le **code** : C'est un espace mémoire **statique**. Il contient le code du programme. Le système d'exploitation accède à cet espace en tant qu'espace mémoire en **lecture seule**.

📖 Les **segments de données** : il s'agit d'un espace mémoire **statique**. Il contient des variables globales initialisées (les **données**) et des variables globales non initialisées (le **bss** – **block started by symbol** –).

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus

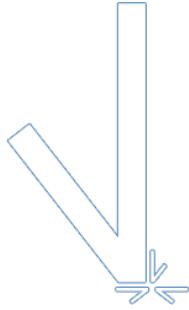


- ☞ La **heap** : c'est un espace mémoire **dynamique** qui peut être consulté à tout moment à partir de n'importe quelle instruction du processus (**accessible globalement**). Il contient des variables avec une **taille variable** qui n'est limitée que par des limitations matérielles.
- ☞ La **pile (stack)**: c'est un espace mémoire **dynamique** qui n'est accessible que par la procédure ou la fonction en cours d'exécution (**accessible localement**). Il contient des variables avec **une taille variable** qui n'est limitée que par les limitations du système d'exploitation.

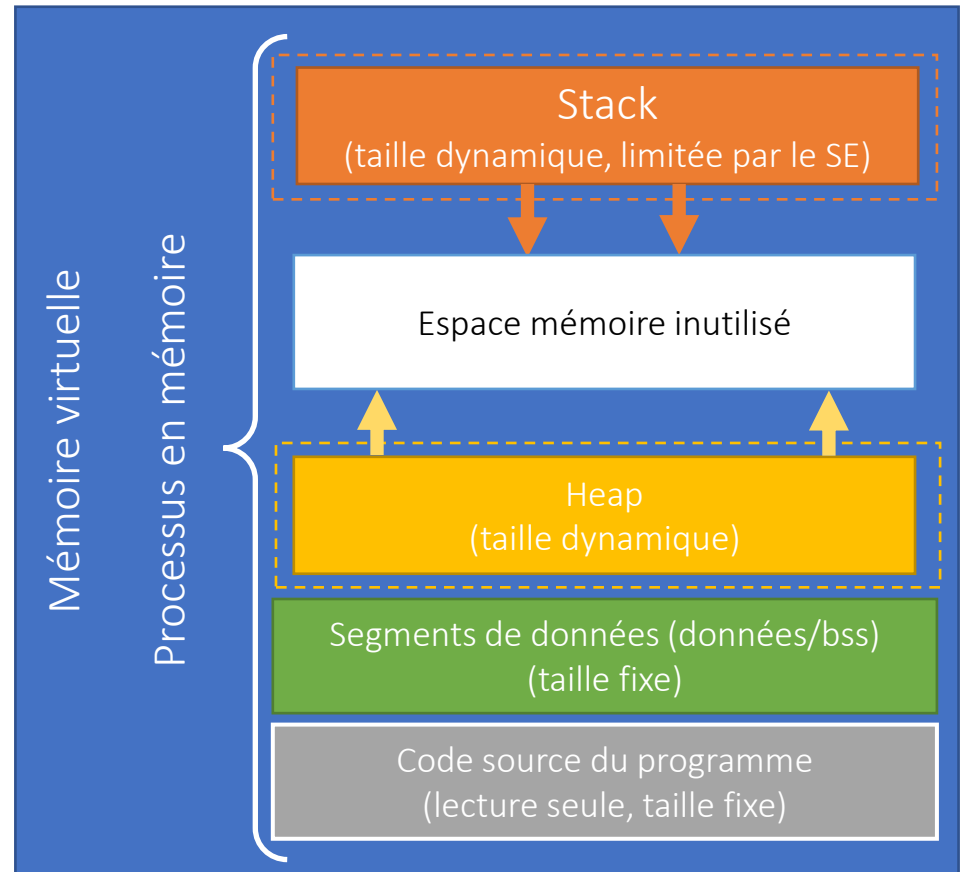
I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



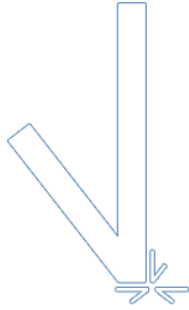
- ❏ Une structure de processus en mémoire (1/3) :
- ⚠ Un processus n'a pas accès aux espaces mémoire alloués à d'autres processus.
- + Les variables déclarées lors de l'exécution d'une procédure ou d'une fonction sont créées à l'intérieur de la **pile (stack)**.
- ✗ À la sortie d'une procédure ou d'une fonction, toutes les variables créées dans la pile sont automatiquement supprimées.



I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus

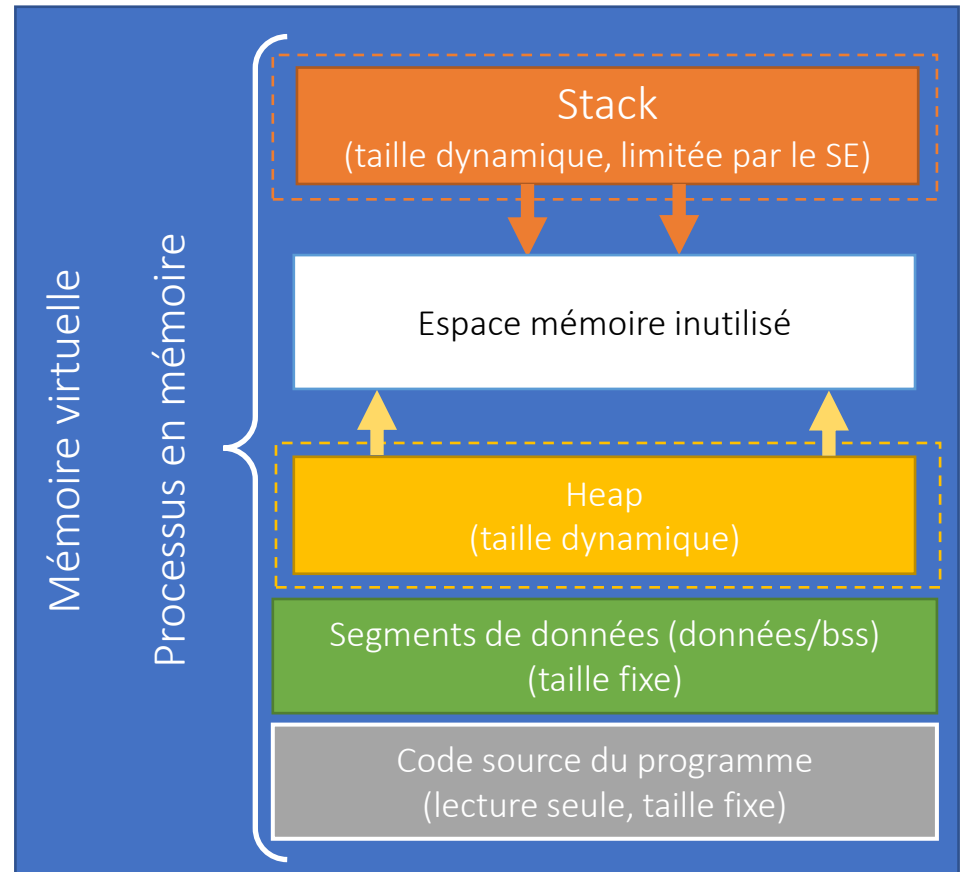


■ Une structure de processus en mémoire (2/3) :

⚠ Les procédures ou fonctions de processus n'ont pas accès aux variables créées par d'autres procédures ou fonctions dans le même processus.

👁 Les variables créées dans la **heap** ou dans les **segments de données** sont disponibles à tout moment par n'importe quelle procédure ou fonction

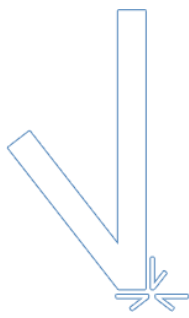
💎 Les variables de la **heap** doivent être supprimées manuellement.



I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



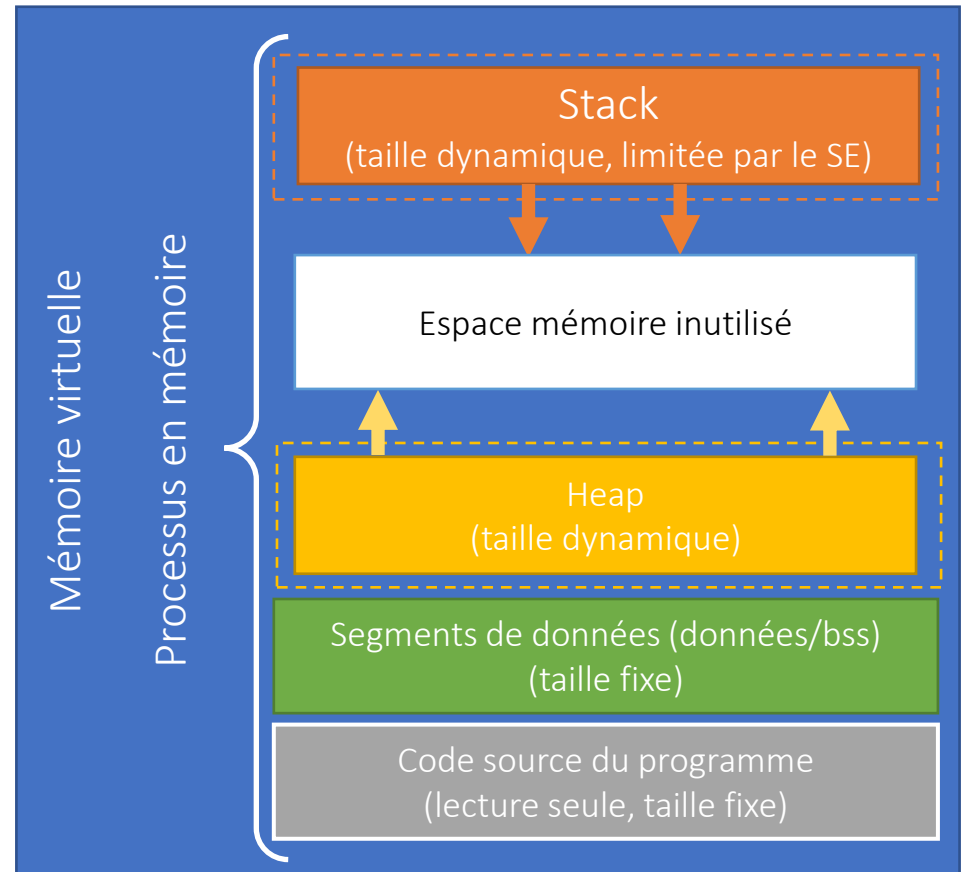
Une structure de processus en mémoire (3/3) :

La taille totale de la **pile** est limitée par le système d'exploitation. La taille de pile par défaut sous Linux est de 8192 kilo-octets. Cette limite peut être vérifiée à l'aide de :

ulimit -s

(ulimit suivi de l'argument -s)

La taille totale disponible pour les autres segments de mémoire du processus est limitée par la couche matérielle.



I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



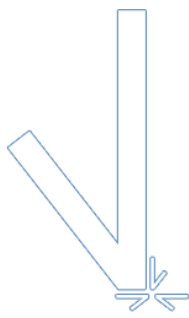
💡 Lorsque la limite de taille de **pile** est atteinte, le noyau de Systèmes d'Exploitation génère une **erreur de dépassement de pile (stack overflow)**. Quand ce type d'erreur se produit, les systèmes d'exploitation modernes déchargent le processus de la mémoire.


🐛 Les **fonctions récursives** avec une **condition de sortie défectueuse** sont une source courante de ce type d'erreur.

I. Systèmes d'Exploitation

I.9. Programmes et Processus


I.9.a Processus













 Les processus actuellement en mémoire peuvent être listés avec:

ps -aux

(ps suivi des arguments -aux)

 Informations sur chaque processus disponible suite à l'exécution de ps :

-  **USER** : Utilisateur qui a démarré le processus;
-  **PID** : L'identifiant unique du processus;
-  **%CPU** et **%MEM** : Utilisation du processeur et de la mémoire;
-  **VSZ – Virtual Memory Size –** : Espace alloué en mémoire;
-  **RSS – Resident Set Size –** : Espace alloué dans la mémoire volatile uniquement;
-  **TTY – TeleTYpewriter –** : Shell à partir duquel le processus a été lancé;
-  **STAT** : [Codes d'état du processus](#);
-  **START** : Date de début du processus;
-  **TIME** : Temps processeur consommé;
-  **COMMAND** : Programme qui a permis de lancer le processus.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



📢 **Communiquer** avec les processus à l'aide de **signaux** :

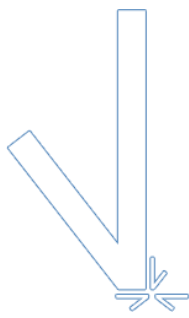
👤 Les signaux sont des **appels système** au noyau.

💡 Les signaux font partie des moyens de communication inter-processus (**IPC**).

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



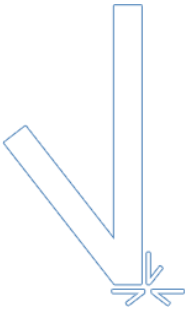
🔊 Communiquer avec les processus à l'aide de **signaux** :

- 💡 Lorsqu'un signal est envoyé au noyau, concernant un processus, **le noyau déclenche un comportement par défaut** sur ce processus (par exemple, le noyau peut suspendre l'exécution d'un processus).
- 🔍 Avant que le comportement par défaut du noyau ne soit déclenché, le noyau vérifie si le processus contient un comportement spécifique concernant ce signal. Si c'est le cas, **le noyau déclenche le comportement spécifique du processus** (par exemple, le processus peut afficher un message).

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.a Processus



☞ Pour envoyer des signaux aux processus, nous pouvons utiliser :

kill -<signal> <pid>

(kill suivi du code du signal à envoyer au processus et du PID du processus)

☞ Pour lister les signaux disponibles :

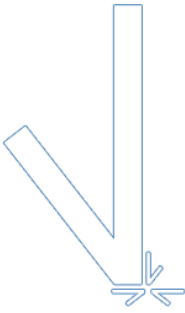
kill -l

(kill suivi de l'argument -l)

☞ Pour arrêter et décharger un processus de la mémoire (« tuer un processus » - fonctionne toujours):

kill -9 <pid>

(kill suivi de -9 pour SIGKILL – tuer le processus – et le PID du processus)



Threads


(fils d'exécution)

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads



- ✂ Un **code en cours d'exécution** à l'intérieur d'un processus peut effectuer un **appel système** au noyau pour demander l'exécution « *parallèle* » **du même code**.
- ✂ Un **code en cours d'exécution** à l'intérieur d'un processus est **appelé thread principal**. Le thread principal peut effectuer des appels système pour créer 1 ou plusieurs **threads secondaires** qui seront exécutés en « *parallèle* ».
-  Le **Dispatcher** du noyau alloue du temps processeur à chaque thread. D'un point de vue matériel, le **nombre de cœurs logiques** du CPU définit le nombre de threads dont le code sera réellement exécuté en « *parallèle* ».

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads



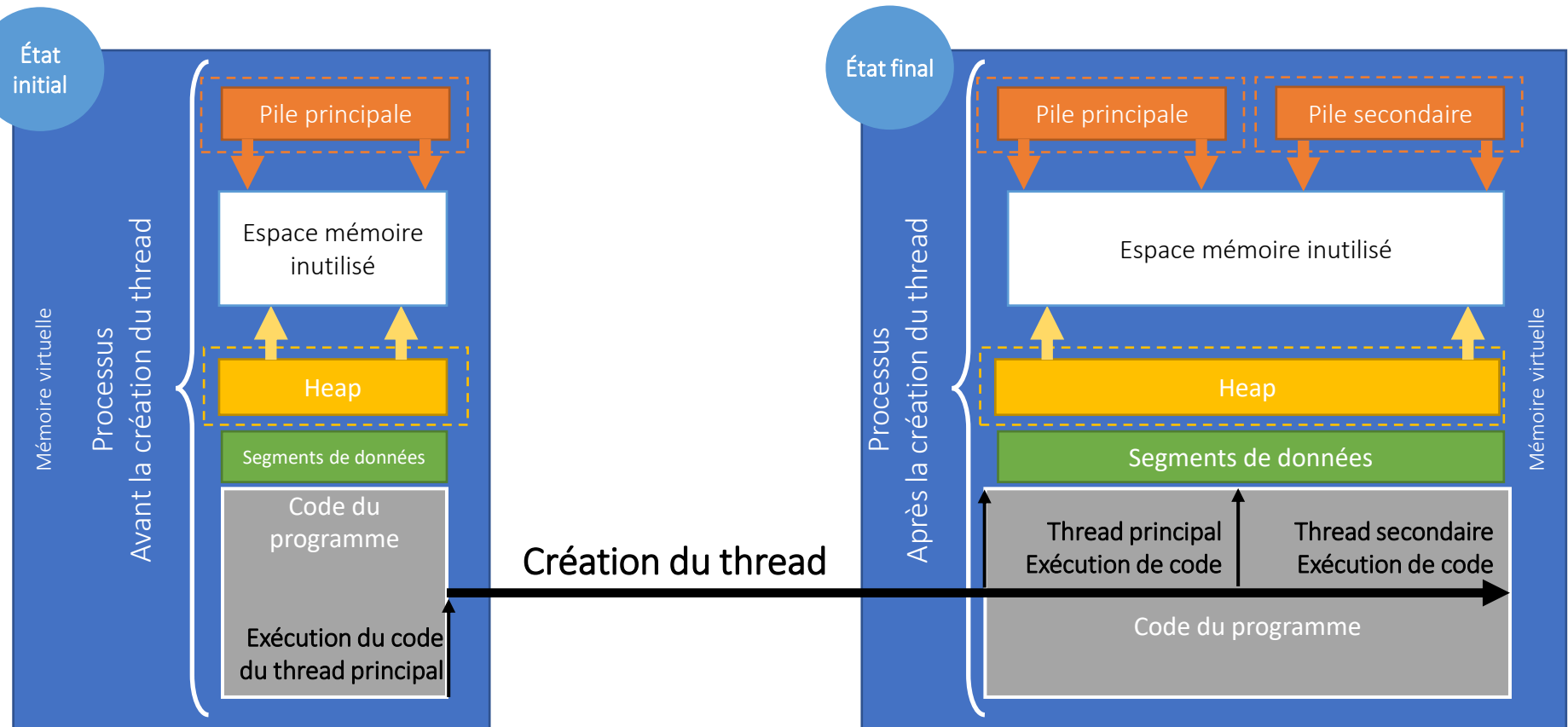
- 📖 Lorsqu'un nouveau thread est créé, la pile du thread d'origine est **dupliquée** et **utilisée** par le nouveau thread.
- 🏰 Les autres segments de mémoire de processus sont **partagés** par les différents threads créés à l'intérieur du processus. Par conséquent, ce nouveau thread peut être à l'origine de **problèmes d'accès concurrents**.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads

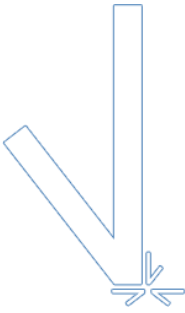
- Création de thread à l'intérieur d'un processus :



I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads



- ☛ Contrairement à la pile du thread principal du processus dont la taille maximale est fixée par défaut à 8192 kilo-octets (8 Mo), la taille maximale de la pile de chaque thread secondaire est fixée par défaut à 2048 kilo-octets (2 Mo).

- ☛ Le **nombre maximal de threads** pouvant être créés par un processus est fixé par le système d'exploitation. Ce numéro peut être affiché avec :

cat /proc/sys/kernel/threads-max


(Utilitaire `cat` suivi du fichier dont le contenu doit être affiché)

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads







 Les threads créés par un processus peuvent être listés en utilisant :

ps -Tp <PID>

(ps suivi par les arguments -Tp et le PID du processus)

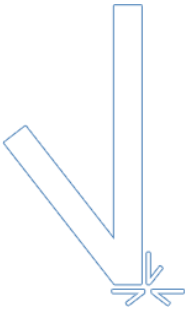
 Informations relatives aux threads :

-  **PID** : Identifiant unique de processus.
-  **SPID** : Identifiant unique de thread.
-  **Time** : Temps processeur consommé.
-  **CMD** : Programme qui a été utilisé pour démarrer le processus.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads

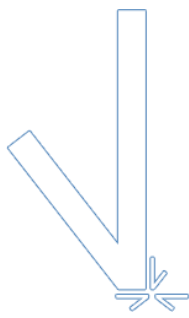


- ✍ Les threads **concurrents** disposent d'un accès en lecture aux variables globales (**dans les segments de données**) et en lecture écriture aux variables de taille dynamique (dans la **heap**).
- 🔒 Dans un processus « **multithread** », il est obligatoire de **synchroniser** l'accès aux variables pour éviter qu'un thread ne lise ou n'écrive dans une variable pendant qu'un autre lit ou écrit dans la même variable.
- ⚠ La partie du code où la **synchronisation** est **obligatoire** est appelée **section critique**.

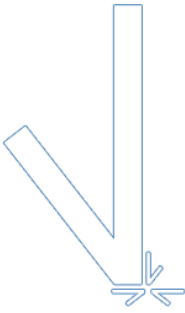
I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.b Threads



- ☞ Parmi les algorithmes de synchronisation qui sont utilisés pour empêcher les accès simultanés, les plus connus sont :
 - ☞ La technique MutEX (Mutual EXclusion), également connue sous le nom de **sémaphores d'exclusion mutuelle**, **empêche l'accès à une ressource** lorsqu'elle est actuellement accessible par un autre thread ;
 - 🕒 La technique du Semaphore, également connue sous le nom de **sémaphores numériques**, **crée une file d'attente** de threads avant une section critique où les threads attendent qu'un thread sorte de la section critique. Lorsqu'un thread quitte la section critique, il le signale aux autres threads afin que le suivant dans la file d'attente puisse entrer dans la section critique.



Child Processes (processus enfant)

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.c Child Processes



- Un processus peut effectuer un *appel système* au noyau pour charger et démarrer un autre processus :
 - 👤 S'il s'agit d'un processus totalement nouveau, on dit que le processus engendre un **processus enfant**
 - 👤 S'il s'agit du même processus dupliqué le nouveau processus est appelé un **fork**.
 - 🔧 Un **fork** est un **processus enfant** qui est le clone du processus d'origine mais dont le code reprend à partir du moment où l'appel système de duplication a été effectué.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.c Child Processes



👤 Le processus parent a accès au PID de ses processus enfant.

🚫 Les processus **ne partagent pas** leur **espaces mémoire respectifs**

🖥️ On peut afficher les processus enfant pour chaque processus avec :

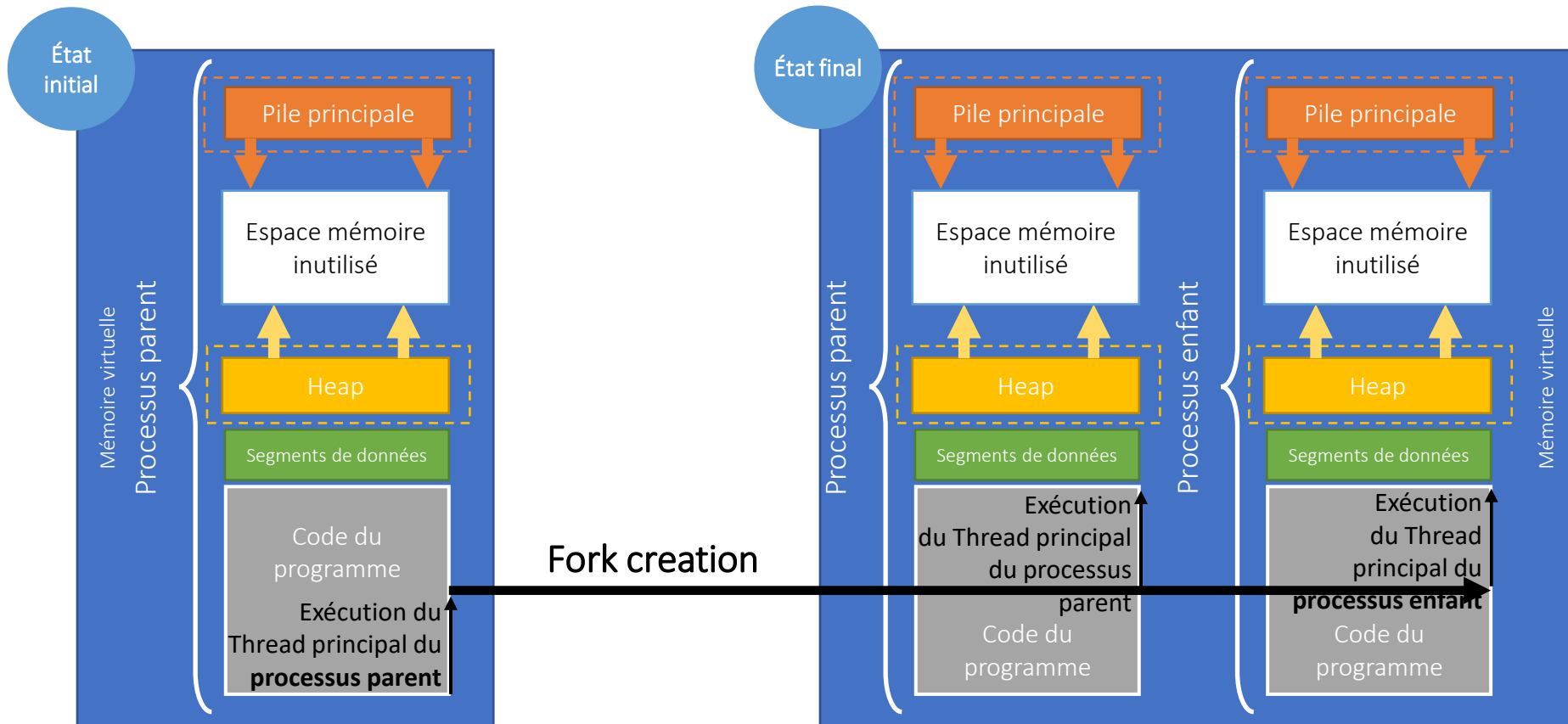
pstree

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.c Child Processes

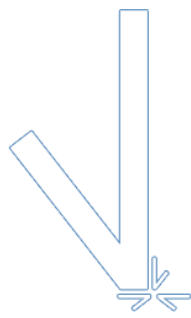
- Création d'un processus enfant de type **fork** :



I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.c Child Processes



- ☒ Chaque commande démarrée à partir du shell est un processus enfant du shell. Les **processus enfant du shell** sont appelés **jobs**.
- 😊 Les processus démarrés à partir de l'interpréteur de commandes sont exécutés de manière **synchrone**. Le shell **attend** la **sortie standard du processus** avant d'être disponible pour l'utilisateur. C'est ce que l'on appelle un **job de premier plan (foreground job)**.
- 👻 Cependant, le processus peut être exécuté de manière **asynchrone**. Le shell **n'attend pas** la **sortie standard du processus** et l'utilisateur peut effectuer d'autres actions. C'est ce qu'on appelle un **job d'arrière-plan (background job)**.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.c Child Processes



- Pour démarrer un **job** (processus enfant du shell) de manière **asynchrone** (avec l'esperluette **&**) :

ls &

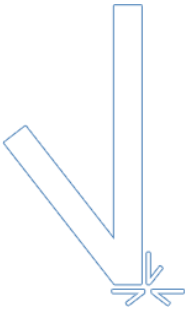
- Pour répertorier tous les jobs (processus enfants shell) et leur ID de job:

jobs

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.c Child Processes

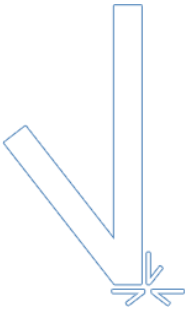


- Un job de premier plan peut être mis en **pause** avec CTRL+Z **puis** démarré en arrière-plan de manière asynchrone avec :

bg

- Un job d'arrière-plan peut être renvoyé au premier plan avec :

fg <job id>

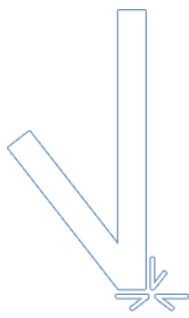


Communication Inter Processus - IPC

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.d Communication Inter Processus



- Bien que les processus **ne partagent pas** leur espace mémoire, le noyau fournit 4 mécanismes ([IPCs](#)) pour **échanger des données entre les processus** :

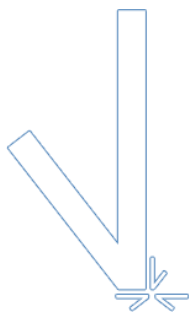
🔊 Les Signaux:

- Les signaux sont des codes numériques qui peuvent être émis par un processus à un autre processus à l'aide d'un *appel système*.
- Le noyau peut déclencher du code à l'intérieur du processus de manière asynchrone lorsqu'un signal est reçu.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.d Communication Inter Processus



✉ Messages et files d'attente de messages (Messages Queues):

- *Messages* est un mécanisme du noyau permettant d'échanger des données entre 2 processus.
- Le **processus expéditeur** envoie son message au noyau à l'aide d'un *appel système*.
- Le noyau met le message en file d'attente dans une file d'attente de messages.
- Le **processus récepteur** peut effectuer un *appel système* pour obtenir le message dans la file d'attente des messages.
- La taille maximale d'un message est limitée à 8192 octets (8 kilo-octets).

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.d Communication Inter Processus



| Pipes :

- Les pipes sont un **canal de communication unidirectionnel** entre les processus ;
- Un processus peut écrire des données dans un tube ;
- Un autre processus peut lire les données reçues dans un tube ;
- La taille maximale des données pouvant être échangées en un seul échange est limitée à 65536 octets (65 kilo-octets) ;

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.d Communication Inter Processus



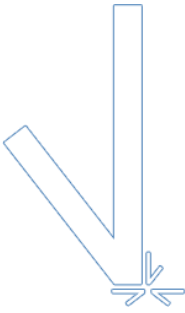
⬇ Shared Heap Memory (SHM) :

- Il s'agit de **segments de mémoire** qui peuvent être partagés entre les processus ;
- L'utilisation de la mémoire dynamique partagée - « shared heap memory » (**shm**) - nécessite **une gestion d'accès concurrents** et, par conséquent, le noyau fournit un **mécanisme d'échange de valeurs de sémaphore interprocessus** ;
- Les données de la mémoire dynamique partagée (**shm**) nécessitent un réglage des **permissions** pour empêcher les processus non autorisés d'accéder aux données.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.d Communication Inter Processus



📖 Les *appels système* pour utiliser des messages, des canaux, des signaux ou des SHM peuvent être effectués dans des programmes en utilisant des fonctions de la norme API POSIX disponible sur le système d'exploitation.

📄 Pour afficher les **messages**, les **segments dans la SHM** et les **sémaphores** :

ipcs

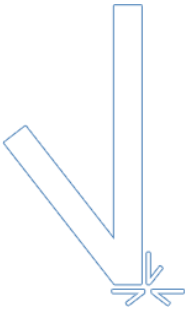


Ordonnancement (Scheduling)

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.e Ordonnancement

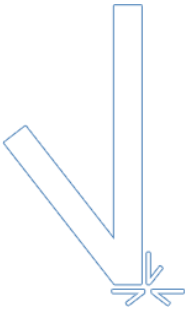


- 🕒 L'ordonnancement est l'opération par laquelle le noyau alloue du temps d'exécution du processeur à chaque processus en mémoire. Le composant de noyau responsable de cette opération est l'ordonnanceur.
- 💡 L'ordonnanceur du noyau Linux ([CFS – Completely Fair Scheduler](#)) est basé sur un algorithme de type **Round Robin**.
- 🔧 L'algorithme Round Robin alloue **un temps CPU fixe par défaut** pour exécuter les processus les uns après les autres. Cependant, il dispose d'un mécanisme de **priorité** qui peut être utilisé pour allouer plus ou moins de temps CPU à un seul processus.

I. Systèmes d'Exploitation

I.9. Programmes et Processus

I.9.e Ordonnancement



- ☞ Un programme peut être démarré avec une priorité d'ordonnanceur spécifique avec :

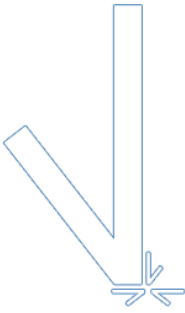
`nice -n 5 <chemin d'accès du programme à exécuter>`

(nice suivi de l'argument `-n` et de la priorité, puis le chemin d'accès du programme à exécuter)

- ☞ La priorité actuellement assignée à un processus par l'ordonnanceur peut être modifiée avec :

`renice -n 5 -p <pid>`

(renice suivi par l'argument `-n`, la valeur de priorité et `-p` le PID du processus)



Variables d'Environnement

(voir cours suivant : Systèmes d'Exploitation – Variables d'Environnement)