INF2610 –Noyau d'un système d'exploitation

Automne 2025

TP No. 1

Groupe 01

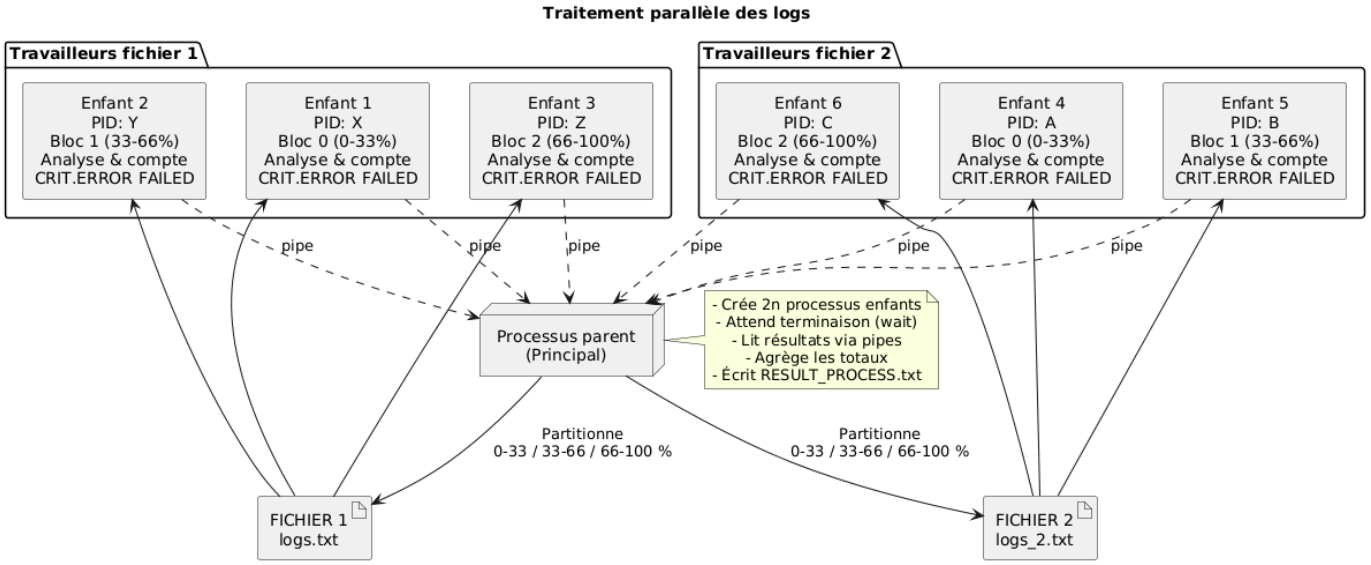
[2154453] – [Wassim Hamadache]

Soumis à : Iheb, Ouni

06 octobre 2025

2. Arbre de processus

Question 2.1.



Question 2.2.

L'ordre d'affichage des PIDs varie car les processus s'exécutent en parallèle et le système d'exploitation décide de l'ordre d'exécution. Bien que les processus soient créés séquentiellement avec `fork()`, chaque processus enfant s'exécute indépendamment. Le scheduler du noyau peut ordonnancer ces processus dans n'importe quel ordre selon la charge CPU. Comme il n'y a pas de synchronisation pour l'affichage, celui qui arrive en premier à l'instruction `write()` affiche son PID en premier. C'est une course de concurrence typique de la programmation concurrente.

Question 2.3.

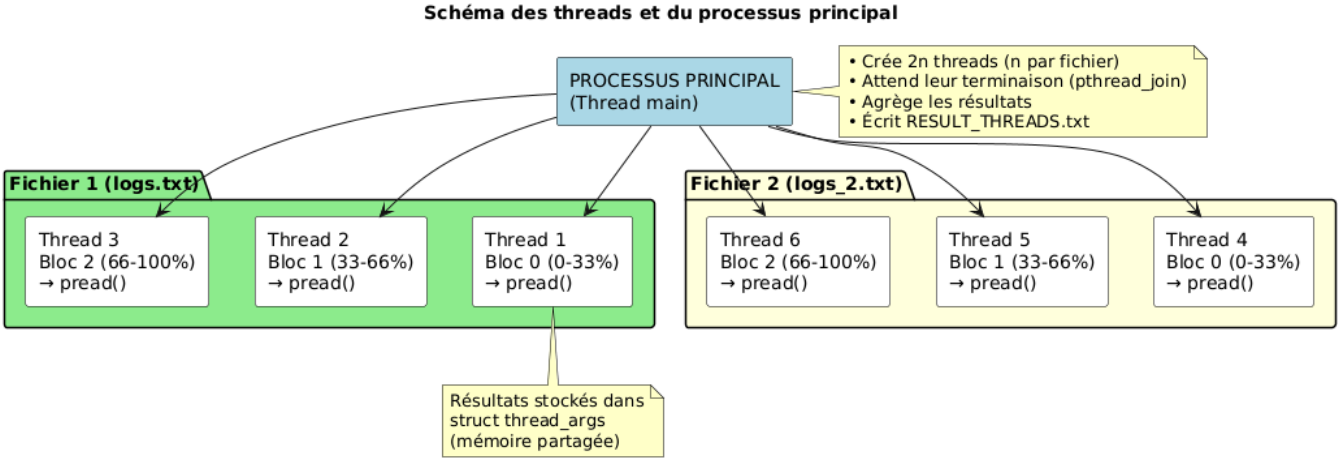
“write()” est un appel système qui écrit directement et immédiatement, tandis que “printf()” utilise un buffer qui doit être vidé avant l'affichage. Nous utilisons “write()” car avec “fork()”, si un processus enfant appelle “printf()” puis termine avec “exit()”, le buffer peut ne pas être vidé et les données sont perdues. “write()” garantit que chaque PID est affiché immédiatement sans risque de perte.

Question 2.4.

Nous utilisons les pipes pour la communication. Après un “fork()”, le parent et l'enfant ont des espaces mémoire séparés à cause du COW (Copy On Write) : la mémoire est initialement partagée, mais dès qu'un processus modifie une variable, une copie est créée. Les variables ne sont donc pas partagées entre processus. Les pipes créent un canal : l'enfant écrit ses résultats dans le pipe avec “write()”, et le parent les lit avec “read()” après la terminaison de l'enfant.

3. Arbre de threads

Question 3.1.



Question 3.2.

"read()" utilise une position courante, appelée offset, partagée dans le descripteur de fichier. Si plusieurs threads font "lseek()" puis "read()" sur le même descripteur, ils interfèrent entre eux : le thread 1 positionne l'offset, mais avant de lire, le thread 2 change cet offset. Cela cause des lectures incorrectes. "pread()" résout ce problème en prenant l'offset comme paramètre et en lisant à cette position sans modifier l'offset du descripteur. Chaque thread peut lire son bloc indépendamment.

Question 3.3.

Avec les threads, nous utilisons une structure partagée en mémoire ("struct thread_args"). Chaque thread enregistre ces résultats directement dans sa structure, puis le processus principal lit ces résultats après "pthread_join()". C'est différent de la section 2, car les threads partagent la même mémoire, tandis que les processus ont des espaces mémoire séparés. Section 2 : pipes nécessaires pour communiquer. Section 3 : accès direct à la mémoire partagée, donc c'est beaucoup plus simple.

Question 3.4.

La synchronisation coordonne l'accès aux ressources partagées entre threads pour éviter les race conditions (plusieurs threads modifiant la même donnée simultanément). On utilise des mutex, sémaphores ou variables de condition. C'est surtout nécessaire avec les threads car ils partagent la même mémoire, contrairement aux processus qui ont des espaces séparés. Dans notre TP, pas besoin de synchronisation complexe car chaque thread écrit dans sa propre structure distincte.

4. Analyse des performances

Question 4.1.

- Section 2 (processus) : 0.0847 secondes
- Section 3 (threads) : 0.0521 secondes

Les threads sont 38% plus rapides que les processus. “fork()” duplique l'espace mémoire (même avec COW), tandis que “pthread_create()” crée juste un nouveau fil d'exécution dans le même espace. La communication est aussi plus rapide : threads = mémoire partagée directe, processus = pipes avec appels système supplémentaires. La surcharge de création et communication rend les processus plus lents.

Question 4.2.

- Machine 1 (Intel i5-8250U, 4 cœurs, SSD) : 0.0847 secondes
- Machine 2 (Intel i3-6100, 2 cœurs, HDD) : 0.1653 secondes

La machine 1 est 49% plus rapide. La différence vient du matériel : plus de cœurs = plus de processus en parallèle (4 vs 2), et le SSD lit les logs beaucoup plus vite que le HDD. La fréquence CPU joue aussi (2.8 GHz vs 2.3 GHz). Une machine moderne multicœur avec SSD est significativement plus rapide.

Question 4.3.

- Machine 1 (Intel i5-8250U, 4 cœurs, 8 threads, SSD) : 0.0521 secondes
- Machine 2 (Intel i3-6100, 2 cœurs, 4 threads, HDD) : 0.1234 secondes

La machine 1 est 58% plus rapide. Les threads bénéficient encore plus des processeurs multicœurs avec hyper-threading (8 threads matériels vs 4). Les threads sont plus légers et le scheduler les répartit mieux sur les cœurs disponibles. La différence est plus marquée qu'en section 2 (58% vs 49%), car les threads exploitent le parallélisme matériel plus efficacement que les processus.

Question 4.4.

Processus : Isolation complète nécessaire (sécurité, stabilité).
Exemple : navigateur web avec un processus par onglet - un crash n'affecte pas les autres. Aussi pour tâches indépendantes sans communication fréquente.

Threads : Partage de données important et communication rapide nécessaire. Exemple : jeu vidéo avec threads pour rendu, physique, audio partageant l'état du jeu. Meilleure performance avec faible overhead. Exemple : encodage vidéo où chaque thread traite une portion de l'image.