SAE S31/32 - Rapport

Commun à toutes les versions

Le code est organisé comme suit :

- **common(.h/.c)** : Ensemble des "includes" et des éventuelles fonctions (helper-like) utilisées dans les autres fichiers source.
- **structures(.h/.c)** : Contient les deux structures (Operation et Jeu).
- **operation(.h/.c)** : Implémentation basique d'un système de "routage" (fortement inspiré d'un "custom-router" Express.JS) permettant l'appel aux fonctions sous-jacentes selon l'ID de l'opération.
- **operationUtils(.h/.c)** : Contient la "vraie" logique sous-jacente pour le traitement des opérations.

> Un fichier "makefile" a été utilisé et configuré afin de permettre une compilation très facile ; la commande "make" permet la compilation du projet.

V0

La V0 ne contient aucune implémentation particulière.
Pour lancer une opération, il suffit d'instancier la structure `DemandeOperation`, de remplir ses champs adéquatement, puis de lancer l'exécution de la demande.
Tout est synchrone, chaque opération se fait à la suite (champ `flags` ignoré). L'ordre est garanti.
Il n'y a qu'un seul processus lancé, tout se fait depuis celui-ci.

V1

Pour lancer une opération, il suffit d'instancier la structure `DemandeOperation`, de remplir ses champs adéquatement, puis de lancer l'exécution de la demande.

Mais contrairement à la V0, si flag = 0, l'exécution de la demande est asynchrone.
Pour ce faire, nous avons procédé comme suit :

- Une demande d'exécution est lancée.
- Si ce n'est pas déjà fait, les principaux éléments (Tableaux des jeux, Pipe principal et réaction au SIGALARM) sont initialisés.
- Un nouveau fork du processus principal est lancé et fait appel à la fonction du `operationUtils` correspondante.
- (Le parent n'attend pas la sortie, le processus continue avec les opérations suivantes.)
- Le fork écrit ensuite son PID et le code de sortie dans le pipe principal.
- Le fork envoie un signal SIGALARM à son parent.
- Le processus principal lit le pipe et affiche la fin de l'opération avec son code de sortie.
- Le fork s'arrête.

Si flag = 1 :

- Une demande d'exécution est lancée.
- Si ce n'est pas déjà fait, les principaux éléments (Tableaux des jeux, Pipe principal et réaction au SIGALARM) sont initialisés.

- Un nouveau fork du processus principal est lancé et fait appel à la fonction du `operationUtils` correspondante.
- Immédiatement après, le processus principal se met en attente de la fin du processus enfant.
- Le fork écrit ensuite son PID et le code de sortie dans le pipe principal.
- Le fork envoie un signal SIGALARM à son parent.
- Le fork s'arrête.
- Le processus principal arrête d'attendre et passe à l'opération suivante.

Cependant, un problème s'est présenté : certaines tâches peuvent être parallélisées sans soucis, mais il serait très utile de pouvoir, à un certain moment, attendre la fin de toutes les opérations en cours avant de continuer.
C'est exactement ce que fait la fonction `wait_for_pending_ops()`, le processus parent est bloqué tant que tous les processus enfants ne sont pas terminés.
Cette fonction est d'ailleurs pratiquement obligatoire à la fin du lancement de toutes les opérations pour attendre la fin des processus enfants avant la sortie du processus principal.

> Nota : Le pipe est mutualisé à tous les enfants et les données qui transitent sont très simples : deux entiers (pid, code de sortie).

> Afin de partager le tableau de jeu et sa taille, nous avons utilisé `mmap()` en mode `MAP_ANONYMOUS|MAP_SHARED` pour que ces deux zones soient accessibles par le processus principal et ses enfants.

> Bien que l'utilisation de pthreads (threads POSIX) aurait pu être envisagée, dans ce cas particulier, cela n'est pas nécessaire. En effet, les processus créés par fork fonctionnent de manière "isolée", et **aucune page mémoire partagée n'est modifiée** en dehors de celles qui sont explicitement mappées en mode partagé avec mmap. En pratique, cela signifie que le processus n'est pas copié en memoire, les pages restent partagées. C'est le principe de **copy-on-write**.

V2

Pour lancer une opération, il suffit d'instancier la structure `DemandeOperation`, de remplir ses champs adéquatement, puis de lancer l'exécution de la demande.

Chaque type d'opération est opéré par un autre exécutable.
Pour ce faire, nous avons procédé comme suit :

- Une demande d'exécution est lancée.
- Si ce n'est pas déjà fait, les principaux éléments (Tableaux des jeux, Pipe principal et réaction au SIGALARM) sont initialisés.
- L'exécutable adéquat est lancé et fait appel à la fonction du `operationUtils` correspondante.
- (Le parent n'attend pas la sortie, le processus continue avec les opérations suivantes.)
- L'enfant écrit ensuite son PID et le code de sortie dans le pipe principal.
- L'enfant envoie un signal SIGALARM à son parent.
- Le processus principal lit le pipe et affiche la fin de l'opération avec son code de sortie.
- L'enfant s'arrête.

> Afin de partager le tableau de jeu et sa taille, nous avons utilisé `shm_open()` afin de permettre le partage d'une zone mémoire en la "nommant". Cela permet donc aux autres processus de lire et

> d'écrire "au même endroit" que le processus principal en ce qui concerne le stockage des jeux.

> Le pipe utilisé est, évidemment, un pipe nommé.

V3

La V3 est un véritable tournant dans notre façon de travailler. Une nouvelle brique a dû être introduite : un serveur de base de données. Servant de concentrateur de données (Nom et Code) pour les jeux. En effet, aucune autre architecture ne permet à chaque « brique » d'être exécutée sur des machines éventuellement différentes. La mémoire ne pouvant être partagée* entre plusieurs machines sur le réseau, un « concentrateur » nous semble être une nécessité. Une autre approche aurait été une architecture de type MQTT avec un courtier central et des serveurs périphériques s'abonnant à des canaux d'événements, mais un SSOT aurait été nécessaire de toute façon. Il ne peut effectuer que quelques actions : Lister, Vérifier l'existence, Envoyer le code et Recevoir un nouveau jeu. Toutes les requêtes sont parallélisées.
**(sauf avec un partage NFS d'un montage TMPFS, mais cela ajoute un SPF non redondant et un niveau de complexité hors du spectre de ce SAE.)*

GDBMP (Game Data Base Management Protocol) est un protocole simple basé sur TCP pour envoyer une requête et tous les arguments nécessaires.

Requete	Argument	Erreurs	Commentaires
count:	N/A	N/A	Répond le nombre de jeu dans la base de données.
list:	N/A	N/A	Répond la liste des jeu (un jeu par ligne) avec la taille du jeu.
exists:	N/A	N/A	Répond "true", si le jeu existe dans la base, "false" sinon.
delete:	(1) Nom du jeu	no_such_game	Supprime le jeu.
get:	(1) Nom du jeu	no_such_game	Répond le code du jeu.
post:	(1) Nom du jeu (2) Code du jeu	game_already_exists	Ajoute un jeu et son code dans la base de données

Le serveur répond alors, soit avec les informations demandées, soit avec un code d'erreur.

Erreurs	Commentaires
no_such_operation	L'opération demandée n'existe pas, vérifiez la requête.
no_such_game	Le jeu demandé n'existe pas.
game_already_exists	Un jeu du même nom existe déja.

> Il n'incombe pas au serveur de base de données de télécharger le jeu à partir du web. L'architecture proposée implique que le jeu doit être téléchargé par le client, puis envoyé au serveur de base de données.

Des clients de démonstration ont été développés pour fournir des exemples.

Conclusion

Nous estimons avoir atteint **99% de la finalité du projet**. Aucun aspect n'a, en tout cas, été volontairement ignoré.
Nous sommes satisfaits du résultat final, malgré le fait qu'au final aucune logique réelle concernant les jeux n'ait été implémentée.
L'utilisation de zones mémoire partagées a été un excellent complément de cours et c'est sans doute le point le plus intéressant de cette SAE.