Dylan Rakotomahefa
Victor Hatt
TP2

Rapport SAE S31

1. Présentation générale du projet

1. Architecture globale qui ne change pas

- Un jeu est représenté par une structure *jeu*

```
typedef struct {
    char NomJeu[25];
    char *Code; // NULL si le jeu n'est pas encore téléchargé
} Jeu;
```

- La base de donnée de jeux est représentée par un tableau de `jeu`
  Vue d'ensemble du système. Nous lui donnons une taille fixe de 10 jeux.
- Une requête est représentée par une structure *DemandeOperation*

```
typedef struct {
   int CodeOp; // Code de l'opération : 1 pour tester, 2 pour ajouter, etc.
   char NomJeu[25]; // Nom du jeu sur lequel l'opération est effectuée
   char Param[200]; // Paramètre supplémentaire comme l'adresse de
téléchargement
   int flag; // 0 par défaut, utilisé pour des informations supplémentaires
} DemandeOperation;
```

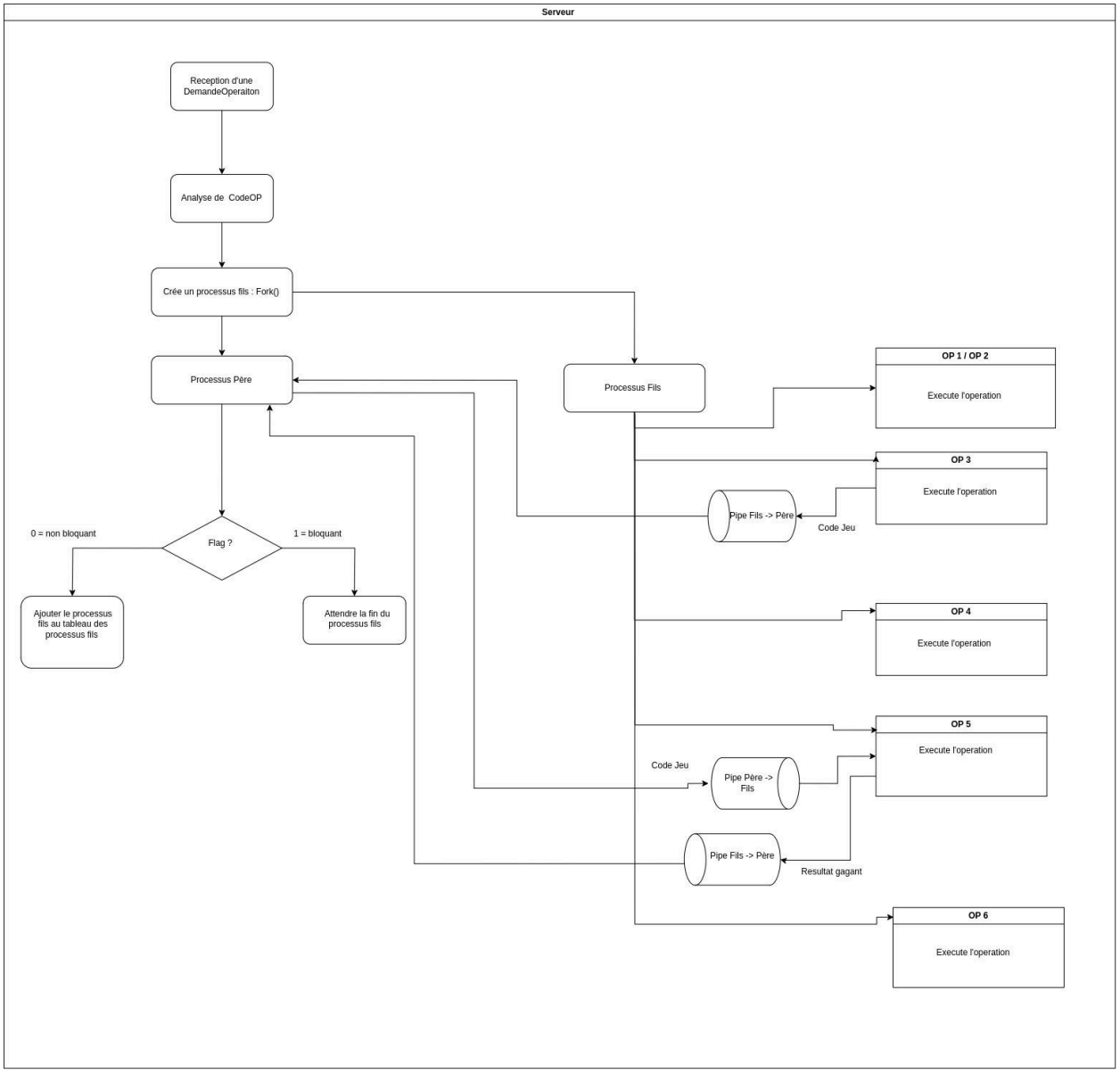
2. Implémentation de la version monolithique (V1)

2.1. V1.a

Afin d'implémenter (ou simuler) le parallélisme avec l'option d'opération (non) bloquant, nous utilisons le *fork().* Une opération bloquante fait donc attendre le processus père le temps de finir l'exécution de la demande. Dans le cas d'une opération non bloquante , le processus père pourra continuer l'exécution des demandes qui suivent.
Cependant, le partage de la mémoire peut s'avérer difficile à gérer car les processus fils travailleront sur une "copie" de la base de donnée.. Nous considérons donc les points suivants :

- On peut admettre que **OP2 Lister_jeux** ou **OP1 Tester_présence_jeu** soient déléguées même si il y a un risque que la BDD ne soit plus un copie "parfaite" de la table du père.
- Par contre, **OP4 Supprimer_jeu** ne peut être faite que par le père et il doit être bloqué le temps de le faire. On admet donc qu'il sera toujours exécuté en mode bloquant

- **OP3 Ajouter_jeu** : le fils simule le chargement dans un tableau de caractère puis le transfère dans un pipe vers le père. On admet donc qu'il sera toujours exécuté en mode bloquant sinon le père ne lit pas le pipe.
- Pour **OP5 Simule_Combat,** le père transmet le code au fils via un pipe. Le fils simule son chargement en "mémoire" et renvoie le résultat au père via un autre pipe. On admet qu'il sera toujours exécuté en mode bloquant sinon le père ne lit pas le pipe.



2.2. V1.b

Pour remédier au problème de partage de mémoire , on implémente une version qui utilise les thread.

Ainsi , pour les opération en mode bloquant , nous faisons un *pthread_join* après avoir créé le thread alors que pour les opérations en mode non bloquant , nous créons simplement le thread :

```
void execute_operation(void (*operation)(int), int mode) {
pthread_t thread;
if (mode == 1) {
        // Mode bloquant
        pthread_create(&thread, NULL, thread_function, (void *)operation);
        pthread_join(thread, NULL); }
else {
        // Mode non bloquant
        pthread_create(&thread, NULL, thread_function, (void *)operation); }
}
```

Pour la gestion des conflits d'accès aux données , on utilise un *pthread_mutex_t* dans chaque opérations

3. Version multi-serveurs (V2) Partie non implémentée

Cette version n'ayant pas été implémenté , les éléments ne seront que des idées

3.1. Architecture des serveurs spécialisés

Un fichier *serveur[nomOP].c* pour chacune des 6 opérations ainsi qu'un *serveurMain.c* qui fera appel aux fonctions des autres serveurs via un *exec()*

4. Version distribuée (V3) en utilisant les sockets

non implémenté

5. Comparaison des différentes versions

V0 : Version séquentielle

Toutes les opérations sont exécutées une par une, de manière séquentielle.

- **Avantages** :
    - Facilité de développement.

    - Pas de gestion de parallélisme ou de communication complexe.
- **Inconvénients** :
    - Temps d'exécution élevé, car aucune opération n'est effectuée en parallèle.
    - Non adaptée pour gérer plusieurs demandes simultanées.

V1.a : Version monolithique avec parallélisme (fork)

Le serveur peut exécuter des opérations en parallèle en utilisant des processus fils (*fork*), mais certaines opérations peuvent être bloquantes.

- **Avantages** :
    - Exploite le parallélisme pour accélérer les tâches non bloquantes.
    - Gestion centralisée : un seul serveur gère tout.
- **Inconvénients** :
    - Consommation élevée de mémoire à cause de la duplication des processus.
    - Complexité accrue pour gérer la synchronisation entre les processus.
    - Certaines opérations (comme supprimer un jeu) doivent rester bloquantes et nous devons faire beaucoup de suppositions notamment par rapport au partage de la mémoire

V1.b : Version monolithique avec threads

Les opérations sont exécutées en parallèle dans le même espace mémoire grâce à des threads.

- **Avantages** :
    - Meilleure gestion de la mémoire (les threads partagent le même espace mémoire).
    - Plus efficace pour des opérations parallèles nécessitant un partage direct des données.
- **Inconvénients** :
    - Risques de conflits sur les données partagées

V2.a : Version multi-serveur (serveurs démarrés à la demande)

Chaque opération est gérée par un serveur dédié, lancé à la demande par un serveur principal.

- **Avantages** :
    - Chaque serveur est spécialisé pour une tâche, ce qui permet une meilleure modularité.
    - Réduction de la charge du serveur principal.
- **Inconvénients** : non développés

Dylan Rakotomahefa
Victor Hatt
TP2

V2.b : Version multi-serveur (serveurs persistants)

Les serveurs dédiés sont toujours actifs et reçoivent des demandes via un mécanisme de communication (par exemple, des pipes).

- **Avantages** :
    - Temps de réponse plus rapide (pas besoin de démarrer les serveurs).
    - Bonne séparation des responsabilités entre les serveurs.
- **Inconvénients : (suppositions)**
    - Consommation de ressources constante (les serveurs restent actifs en permanence).
    - Gestion plus complexe des communications et des synchronisations.

V3 : Version distribuée (non développée)