Université de Strasbourg

S31/S32 Projet SAé

Comprendre les Systèmes et Réseaux

gancarski@unistra.fr

IUT Robert Schuman Université de Strasbourg

Présentation et mise en oeuvre

Objectif : Comprendre la gestion, la synchronisation et le communication entre processus

Notions : Appels système de création et gestion de processus / Communication et synchronisation inter-processus

---

1 Présentation

Un client désire que nous créions et lancions un serveur qui gère un base de données contenant des jeux (échecs, go, bataille navale, ...). Le but du SAé n'est pas de définir de façon optimale cette base mais juste de s'en servir. On va donc simplement utiliser un tableau de jeux où un jeu est défini par

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

Un utilisateur peut demander au serveur un certain nombre d'opérations sur les jeux stockés dans la base de données. Chacune de ces opérations à un code (codeOP) et prend un certain temps (que pour simplifier nous considérons comme constant pour chaque opération).

1. Tester si un jeu dont le nom est donné en paramètre est présent dans la base de données. Elle retournera -1 sur un échec, 0 en cas de succès. (Durée : instantané)
2. Affiche la liste des jeux disponibles et téléchargés. Elle retournera -1 sur un échec, le nombre de jeux en cas de succès. (Durée : 1 seconde)
3. Ajouter un jeu dans la base de données : le serveur va télécharger le jeu depuis l'adresse fournie directement. Il retournera retournera -1 sur un échec, sinon la taille en Mio du jeu téléchargé. (Durée : 10 secondes)
4. Supprimer un jeu de la la base de données donné par son nom. Elle effacera le jeu et retournera -1 sur un échec, la taille du jeu en cas de succès. (Durée : 2 secondes)
5. Simuler un combat : le serveur jouera contre lui-même le jeu dont le nom est donné en paramètre. Affichage du gagnant. Pas de retour (Durée : 20 secondes)
6. Lancer un jeu : le serveur charge le jeu en mémoire et le lance en interaction avec un joueur. Affichage du gagnant. Pas de retour (Durée maximale : indéfinie)

1.1 Définition d'une demande

Pour simplifier, on s'inspire du format RISC qui impose que les opérations soient toutes de taille fixe et de même structure. On définit donc une demande d'opération sur un jeu par une structure

```
typedef struct {
 int  CodeOp ; // Code de l'opération // 1 : test, 2 : ajout ...
 char NomJeu[25];
 char Param[200]; // Adresse de téléchargement
 int flag; // 0 par défaut : contiendra des informations sur la demande elle-même (sera vu plus ta
} DemandeOperation;
```

1.2 Exécution d'une demande

Une fonction devra être définie et permettra de réaliser une demande passée en paramètre :

```
int execute_demande(DemandeOperation OP) {
  switch (OP.CodeOP) {
    case 1: ...   ; // retour : -1 sur un échec, 0 en cas de succès.
    case 2: ... ; // retour : -1 sur un échec, sinon le nombre de jeux téléchargés.
    case 3: ... ; // retour : -1 sur un échec, sinon la taille en Kio du jeu téléchargé.
    case 4: ...
   ...
  }
}
```

De façon évidente, on ne réalisera pas concrètement toutes les opérations :

- Pour l'opération Ajouter :
  - L'entrée dans la table est créée avec le nom du jeu
  - On ne chargera pas le code réellement. On allouera simplement un nombre aléatoire (compris entre 1 et 1000) de caractères '*'.
  - Puis on simulera le temps de téléchargement en bloquant le serveur pendant la durée de l'opération (10 secondes).
- Pour l'opération Simuler :
  - On ne lancera pas réellement le code. On simulera simplement le chargement en copiant ce "code" dans un autre tableau "représentant" la mémoire
  - Puis on simulera le temps de combat en bloquant le serveur 20 secondes. Il affichera alors soit "gJ1 ou J2 (au hasard).
- Pour l'opération Lancer :
  - On ne lancera pas réellement le code. On simulera simplement le chargement en copiant ce "code" dans un autre tableau "représentant" la mémoire de jeu.
  - On simulera le temps de jeu en bloquant le serveur jusqu'à ce que l'utilisateur entre un caractère au clavier. Il affichera alors soit Serveur ou Joueur (au hasard).

1.3 Exemple d'un programme principal

Une première version (V0) de ce serveur va permettre d'initialiser et tester le service

```
int main() { // Exemple avec une seule demande d'ajout
// Déclaration d'un demande
 DemandeOperation DeO = {1,"Echec", "http:echecetmat.com", 0};
// Exécution de l'opération :
  int res = execute_demande(DeO);
// Affichage du résultat :
  printf("Résultat : %d \n",res);
}
```

Compléter ce programme pour qu'il réalise ces 6 opérations en séquentiel (i.e. l'une après l'autre).

2 Un serveur "parallèle"

Le client (acheteur de ce service) à une exigence assez forte → Le serveur doit pouvoir exécuter des opérations en parallèle tout en limitant la consommation de mémoire et en réduisant au maximum les temps d'exécution. Le client accepte de faire un développement incrémental et propose donc de tester 3 versions différentes : monolithique, multi-serveur et distante.

V1 : Version monolithique (le serveur contient le code de toutes les operation via la fonction `void execute_demande(opération OP)`.

- V1.a : Le serveur doit pouvoir lancer des opérations non bloquantes (flag == 0) en parallèle mais aussi être capable d'attendre la fin d'une opération bloquante (flag == 1). Par exemple, il pourrait lancer les 3 premières opérations en parallèle (mode `non bloquante`), puis lancer la 4ème en mode `bloquante` (attendre que le jeu soit chargé) et enfin lancer le combat ou le jeu en mode `non bloquante` :
  — {1,"Echec", "http :echecetmat.com", 1}
  — {3,"Go", "http :goettic.com", 1}
  — {3,"Othello", "http :oterleau.com", 1}
  — {5,"Echec", NULL, 0}
  — {2,"Go",NULL, 0}
  — {6,"Othello",NULL, 0}
  — ...
- V1.b : Le serveur est toujours issu d'un seul code mais intègre le partage de mémoire tout en maintenant la possibilité de parallélisation des exécutions.

V2 : Version basée sur un ensemble de serveurs

- V2.a : Chaque serveur est spécialisé dans une opération (un serveur CombatServ, un serveur JeuServ...) avec des codes différents du coup : le serveur principal, qui ne contient plus le code correspondant aux opérations, devra lancer le serveur dédié à chaque exécution d'une opération en lui transférant les arguments de celle-ci dont le code du jeu si nécessaire
- V2.b : Les serveurs dédiés sont toujours actifs : le serveur principal va juste leur transférer les arguments des opérations à exécuter.

V3 : Version dans laquelle les exécutions des opérations se feront sur une ou plusieurs machines distantes

Chacune de ces versions amène un certain nombre de questions auxquelles il va falloir répondre.
À titre d'exemple et de manière non exhaustive :

- V1 : Comment faire pour que des demandes puissent s'exécuter en même temps ? Dans l'un des cas quelles sont les demandes que le serveur doit effectuer lui même et pourquoi (Ces opérations seront donc d'office bloquantes) ? Comment mettre en attente le serveur sur des opérations bloquantes ? Comment faire pour pouvoir exécuter toutes les opérations en parallèle en mode bloquant ou non bloquant ? ...
- V2 : Comment lancer les serveurs dédiés à la demande ? Comme leur transférer les arguments des opérations à exécuter ? ...
- V1 et V2 : Comment le serveur peut-il savoir quand un calcul non bloquant a pris fin ? Comment peut-il être sûr qu'il n'en reste plus en cours d'exécution à la fin ? ...
- V1 et V2 : Quelle est la meilleure version d'après vous ? ...
- V3 : Comment implanter le code distant ? Comme demander l'exécution distante ? Comment transférer les paramètres et le résultat ?

Corrigé

Quelques éléments de réponse :

- *V1a : On utilisera des fork pour autoriser le //. Du coup les opérations directes sur la table sont difficiles voire impossibles à déléguer à un fils car il travaillera sur une "copie" de la table. On peut admettre que Lister ou Tester soient déléguées même si il y a un risque que la table ne soit plus un copie "parfaite" de la table du père. Par contre, Supprimer ne peut-être faite que*

par le père et il doit être bloqué le temps de le faire. Ajouter peut à la limite, se faire par un processus : le fils simule le chargement dans une mémoire puis transfère cette mémoire dans un pipe vers le père –> l'opération ne peut qu'être bloquante sinon le père ne lit pas le pipe. Pour Combat, le père transmet le code au fils qui simule son chargement en "mémoire" et revoie le résultat. Si on ne s'intéresse pas au résultat, le combat peut être non bloquant. Sinon il doit l'être car sinon le père ne lit pas le pipe

- *V1b : Même espace d'adressage : utilisation de threads*
- *V2 : Utilisation de execv. On utilisera (et modifiera) la version V0.1a en ajoutant un mécanisme de passage des paramètres (par exemple, il faut "transférer" le code du jeu à simuler dans un Combat). On peut aussi introduire l'appel "system"*
- *V2b : Utilisation de pipe*
- *V1a et V2a : Comment le serveur peut-il savoir quand tous les calculs non bloquants ont pris fin ? Deux solutions :*
  1. *Utilisation de waitpid non bloquant sur les opérations non bloquantes –> ne peut être fait qu'à la fin des tous les calculs. On mémorise les pid des fils "bloquants" dans un tableau `PIDf` puis on les teste :*

     ```
     do{
      kidpid = waitpid(-1, &status, WNOHANG);
      PIDf[kidpid] = 1; nbfils_non_bloquants- }
     } while (nbfils_non_bloquants >0)
     ```

     Ne fonctionne pas en V2b car les processus ne s'arrêtent jamais
     Facultatif : Que se passe-t-il si un des processus fils (V1a ou V2a) ne se termine pas ? –> Fonction suppression des zombies en utilisant `PIDf`
  2. *Utilisation d'un signal (SIGUSR2 par exemple) : Pour compter le nombre de serveurs dédiés qui ont fini leur exécution : émission par le processus fils de SIGUSR2 juste avant sa fin / comptage de signaux SIGUSR2 reçus par le serveur : risque de perte de message.*
- *V3 : Socket ou RPC ?*

Dernière contrainte imposée par le client : l'application doit pouvoir être compilée en une seule commande et en plus, uniquement les fichiers mis à jour et ceux dépendant de ceux-ci doivent être compilés.

Corrigé

Utilisation de make bien sûr