Université de Strasbourg

S31 Projet bimestriel
Comprendre les Systèmes
gancarski@unistra.fr

IUT Robert Schuman Université de Strasbourg

Présentation et mise en oeuvre

Objectif : Comprendre la gestion et la synchronisation de processus

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

---

1 Présentation

On désire créer et lancer un serveur de calcul capable de réaliser des programmes formés d'instructions diverses.

1.1 Définition d'une instruction

Pour simplifier, on s'inspire du format RISC qui impose que les instructions soient toutes de taille fixe et de même structure. On définit donc une telle instruction par un tableau de 6 entiers :

code opération	arg 1	arg 2	res 1	res 2	flag

L'entier flag contiendra des informations sur l'instruction elle-même (sera vu plus tard)

Afin de rendre le projet réalisable, on commence avec uniquement des instructions arithmétiques (si ça vous amuse vous pouvez en définir d'autres sans problème...) numérotées comme suit :

1. Addition : add
2. Soustraction : sub
3. Multiplication : mul
4. Division : div
5. Division entière : divE

Ex : l'instruction "add 12 7 0" (addition de 12 et 7 avec un flag égal à 0) sera codée par :

1	12	7	0	0	0

Le résultat se trouvera toujours dans "res 1". Donc après exécution on obtient :

1	12	7	19	0	0

1.2 Code initial du serveur

Le code proposé pour le serveur (Version V0) est le suivant :

```
// Déclaration d'un type Instruction
typedef int Instruction[6] ;
// La fonction qui exécute l'instruction
void execute_instruction(Instruction OP) {
  switch (OP[0]) {
    case 1: OP[3] = OP[1] + OP[2]; break;
    case 2: OP[3] = OP[1] - OP[2]; break;
    case 3: OP[3] = OP[1] * OP[2]; break;
    case 4: if (OP[2] !=0) OP[3] = OP[1] / OP[2]; break;
    case 5: OP[3] = OP[1] / OP[2];
            OP[4] = OP[1] % OP[2]; break;
```

```
  }
}
// Le programme principal V0
int main() {
// Déclaration de l'instruction précédente "add 12 7 0":
 Instruction PI = {1,12,7,0,0,0};
// Affichage de l'instruction :
 printf("add 12 7 0 : | %d | %d | %d | %d | %d | %d |\n",PI[0],PI[1],PI[2],PI[3],PI[4],PI[5]);
// Exécution de l'instruction :
  execute_instruction(PI);
// Affichage du résultat :
  printf("Résultat : | %d | %d | %d | %d | %d | %d |\n", PI[0],PI[1],PI[2],PI[3],PI[4],PI[5]);
}
```

On trouvera sur Moodle différents fichiers correspondants à trois versions de ce serveur qu'il vous faudra (en TP) corriger/compléter si nécessaire, compiler et vérifier le bon fonctionnement :

1. `serveurV0.c` : Version initiale du serveur
2. `serveurV0.1.c` : Le serveur prend en paramètre l'instruction à exécuter. Par exemple, `./a.out add 12 7 0` permet d'exécuter l'instruction précédente.
3. `serveurV0.2.c` : Le serveur peut stocker un programme de 10 instructions maximum à exécuter. Compléter le pour que cela fonctionne avec la liste des instructions suivante :
   — add 12 7 0
   — sub 36 8 0
   — sub 8 6 0
   — div 13 2 0
   — divE 13 3 0
   — add 68 4 0

Rappel Une compilation se fait par `gcc -Wall Prog.c`. L'utilisation de l'option `-Wall` est **obligatoire** en S31 (et S41 aussi !) car elle permet d'afficher tous les warnings. A votre niveau, il n'y a aucune raison d'avoir un warning ⇒ un warning sera considéré comme une erreur !
Donc, si à l'exécution d'un programme vous avez un plantage ou une erreur d'exécution, vous ne pourrez demander de l'aide à un enseignant pour débugger votre programme que s'il n'y a plus de "warning" (sauf si c'est pour comprendre le warning bien sûr)

Remarque Si lors de la compilation le message `warning: implicit declaration of function` ou `error : call to undeclared function` apparait, c'est bien souvent qu'il manque un ou plusieurs #include. Pour savoir lesquels, utiliser la commande `man` (par exemple `man 2 printf`)

2 Mise en œuvre

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

- V1 : Version monolithique (le serveur contient le code de toutes les instructions via la fonction `void execute_instruction(Instruction OP)` comme dans la version V0.2).

(a) Le serveur doit pouvoir lancer des instructions non bloquantes (flag == 0) en parallèle mais aussi être capable d'attendre la fin d'une instruction bloquante (flag == 1). Par exemple, il pourrait lancer les 3 premières instructions en parallèle (mode `non bloquante`), puis lancer la 4ème et 5ème en mode `bloquante` et enfin lancer le dernière en mode `non bloquante` :
   — add 12 7 0
   — sub 36 8 0
   — divE 19 6 0
   — div 13 2 1
   — divE 13 3 1
   — add 68 4 0

(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

(a) Chaque serveur est spécialisé dans une opération (un serveur addService, un serveur subService...) avec des codes différents du coup : le serveur principal, qui ne contient plus le code correspondant aux instructions, devra lancer le serveur dédié à chaque exécution d'une instruction en lui transférant les arguments de celle-ci

(b) Les serveurs dédiés sont toujours actifs : le serveur principal va juste leur transférer les arguments des instructions à exécuter.

- V3 : Version dans laquelle les exécutions des instructions se feront sur une ou plusieurs machines distantes (Il faudra patienter jusqu'en S41...)

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 exécutions puissent se faire en même temps ? Comment mettre en attente le serveur ? Comment exécuter les instructions en parallèle dans un même espace d'adressage ? ...
- V2 : Comment lancer les serveurs dédiés à la demande ? Comme leur transférer les arguments des instructions 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 ? ...

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.