TP n°1 : Simulation d'un processeur et d'un système

Les TP 1 et 2 sont à rendre ensemble. Les modalités seront précisées lors des séances suivantes.

1 Présentation de la machine

Pour suivre ce TP vous devez récupérer le prototype du simulateur qui est mis à votre disposition. Ce prototype a la structure suivante :

- `cpu.c` et `cpu.h` : définition de la CPU et de la mémoire
- `systeme.c` et `systeme.h` : définition du système
- `asm.c` : un mini-assembleur
- `simul.c` : fonction principale
- `prog1.asm` : un programme assembleur exemple

1.1 Mémoire centrale

La mémoire centrale de notre machine est composée de mots. Un mot mémoire est un entier de 32 bits (la taille des entiers sur une architecture PC classique). La mémoire contient quelques dizaines de mots simulés par un simple tableau. Les adresses physiques sont contiguës et varient de zéro à 127.

Un extrait du fichier `cpu.c`

```
typedef int WORD; /* un mot est un entier 32 bits */

WORD mem[128];   /* mémoire                      */

/* fonctions de lecture / écriture */
WORD read_mem(int physical_address);
void write_mem(int physical_address, WORD value);
```

1.2 Structure du processeur

Le mot d'état du processeur est défini comme suit

```
typedef struct PSW { /* Processor Status Word */
    WORD PC;        /* Program Counter */
    WORD AC;        /* Accumulator */
    WORD IN;        /* Interrupt number */
    INST RI;        /* Registre instruction */
    WORD IO;        /* input/output data */
} PSW;
```

Définition des registres :

- `PC` : Le **Program Counter** (compteur ordinal) est un pointeur sur la prochaine instruction à exécuter (en fait un entier).
- `AC` : Le processeur dispose d'un registre unique (l'**accumulateur**) pouvant contenir un entier signé de 32 bits et sur lequel se déroule les opérations de base.
- `IN` : En cas d'interruption, la CPU range dans ce registre la cause de cette interruption. Cette information peut être exploitée par le système d'exploitation.
- `RI` : Instruction qui est en cours d'exécution.
- `IO` : Registre de communication pour les entrées/sorties.

1.3 Instructions du processeur

Notre processeur exécute des instructions à taille fixe (un mot de 32 bits). Une instruction est composée d'un code opération, de deux numéros de registre et d'un argument. Vous pouvez voir les détails ci-dessous :

Définition d'une instruction

```
typedef struct {
    short op;     /* code operation (16 bits) */
    short arg;    /* argument (16 bits)       */
} INST;
```

Une instruction va ressembler à ceci : `add 1000` (une action et un argument). Les instructions disponibles sont détaillées ci-dessous :

Les instructions de notre CPU

```
set arg         // AC = arg; PC++
add arg         // AC = AC + mem[arg]; PC++
sub arg         // AC = AC - mem[arg]; PC++
load adr        // AC = mem[ adr ]; PC++
store adr       // mem[ adr ] = AC; PC++
nop             // PC++
ifeq adr        // PC++; si (AC == 0) PC = adr
ifgt adr        // PC++; si (AC > 0) PC = adr
iflt adr        // PC++; si (AC < 0) PC = adr
jump adr        // PC = adr
sysc arg        // PC++; appel du système
```

Un programme en assembleur est une séquence d'instructions. En voila un exemple (voir `prog1.asm`) :

Une boucle avec diminution de AC par pas de 2000

```
    set 6000            // AC = 6000
loop:                   // définir loop
    iflt end            // si (AC < 0) aller à end
    sub incr            // AC = AC - mem[ incr ]
    nop                 // ne rien faire
    jump loop           // aller à loop
end:
    jump end            // boucle infini

incr: data 2000         // valeur de l'increment
```

1.4 Simulation de la machine

Nous avons vu en cours que le processeur passe son temps à alterner des cycles ou il exécute du code utilisateur et des cycles ou il exécute du code système. Elle passe du code utilisateur au code système par une interruption et du code système au code utilisateur par un chargement du mot d'état processeur (ou **processor Status Word**). On peut donc simuler ce comportement par le code ci-dessous :

```
int main(void) {
    PSW cpu = system_init();
    for(;;) {
        cpu = simulate_cpu(cpu);
        cpu = process_interrupt(cpu);
    }
    return (EXIT_SUCCESS);
}
```

La fonction `simulate_cpu()` simule l'exécution du code utilisateur jusqu'à l'apparition d'une interruption. La fonction `process_interrupt()` reprends la main, traite l'interruption et redonne la main au code utilisateur.

2 Nouvelles fonctions à réaliser

2.1 Démarrer le simulateur

1. Récupérez l'archive.
2. Décompressez l'archive :

```
$ unzip simul.zip
$ cd simul
```

3. Compilez le projet :

```
$ cd simul
$ make
```

4. Exécutez le simulateur :

```
$ ./simul
```

5. Recompilez le simulateur :

```
$ make clean
$ make
$ ./simul
```

2.2 Tracer les interruptions

1. Pour l'instant le simulateur fonctionne sans rien afficher. Enlevez les commentaires associés à l'interruption `TRACE` dans la fonction `process_interrupt` du fichier `system.c`. L'affichage (`dump_cpu`) et l'attente (`sleep`) devraient vous permettre de suivre et de comprendre le fonctionnement du programme (`prog1.asm`).
2. Faites en sorte que le système indique les numéros d'interruption reçus (fonction `process_interrupt` de `system.c` à modifier).
3. Faites en sorte que les interruptions d'erreur (instruction inconnue `INT_INST` et erreur d'adressage `INT_SEGV`) provoque l'arrêt du simulateur (donc un appel à `exit()`). Modifiez le programme exécuté pour faire apparaître une des deux erreurs (revenez ensuite à la version d'origine).

2.3 Appels au système

Pour l'instant les affichages de notre simulateur sont réalisés par les traces du **PSW** faites par le système. Il est temps maintenant d'utiliser une nouvelle instruction, que nous appellerons `sysc`, dont le but est de générer une interruption afin de donner la main au système. La partie argument de cette instruction indiquera au système l'action voulue.

Préparation :

- Consultez les numéros d'interruption prévus dans `cpu.h`.
- Analysez le contenu de la fonction `system_call` qui assure le traitement des appels au système.
- Testez le fonctionnement en plaçant une instruction `sysc` au coeur de la boucle.

Traitement de l'appel système EXIT :

- Cet appel provoque l'arrêt du thread demandeur et donc du système puisque nous avons, pour l'instant,

un seul thread.

- Définir les deux macros ci-dessous dans votre programme en assembleur :

```
define SYSC_EXIT 100
define SYSC_PUTI 200
```

- Remplacer dans le code du programme assembleur l'instruction `jump end` (la boucle infinie) par l'instruction d'appel au système ci-dessous.

```
sysc SYSC_EXIT   // Appel au système pour SYSC_EXIT
```

- Compléter la fonction `sysc_exit`.

Traitement de l'appel système PUTI :

- Cet appel provoque l'affichage de l'entier stocké dans l'**accumulateur**.
- Compléter la fonction `sysc_puti`.
- Placer cette instruction au coeur de la boucle et voir le déroulement de la boucle.

3 Introduction du multi-tâches

Le but de cette section est double : d'une part, ajouter des fonctions multi-tâches en temps partagé à notre mini système et d'autre part, utiliser ces nouvelles fonctions pour endormir les threads pendant un certain temps

3.1 Codage des threads

Nous avions déjà prévu dans le simulateur fourni, les structures de données ci-dessous. Elles permettent de représenter un ensemble de threads et leur mot d'état processeur.

```
#define MAX_THREADS (20) /* nb maximum de threads */

typedef enum {
    EMPTY = 0,           /* thread non-prêt       */
    READY = 1,           /* thread prêt           */
} STATE;                 /* État d'un thread      */

typedef struct {
    PSW   cpu;           /* mot d'état du thread  */
    STATE state;         /* état du thread        */
} PCB;                   /* Un Process Control Block */

PCB thread[MAX_THREADS]; /* table des threads     */

int current_thread = -1; /* nu du thread courant  */
```

Faites en sorte qu'au démarrage du système (fonction `system_init`), le système prépare la première case du tableau des threads. Il y a pour l'instant un seul thread.

3.2 Un ordonnanceur simplifié

A chaque interruption `TRACE`, le système va maintenant sauvegarder le PSW dans la case correspondante du tableau des threads et chercher un nouveau thread prêt pour lui redonner le processeur (voir recherche ci-dessous).

Fonction de l'ordonnanceur

```
PSW scheduler(PSW cpu) {
    /* À FAIRE : sauvegarder le thread courant si il existe */
    do {
        current_thread = (current_thread + 1) % MAX_THREADS;
    } while (thread[current_thread].state != READY);
    /* À FAIRE : relancer ce thread */
}
```

Travail à faire :

- Ajoutez l'appel à la fonction `scheduler` dans le traitement de l'interruption `TRACE`.
- Complétez la fonction `scheduler()` qui code l'ordonnanceur. À ce stade, le simulateur doit fonctionner correctement. Étant donné qu'il n'y a qu'un seul thread, il est systématiquement sauvegardé puis choisi pour être exécuté.
- Pour tester votre ordonnanceur, vous pouvez maintenant créer directement deux threads au démarrage du système (prenez l'exemple de la boucle de la section précédente). Les sorties des deux threads devraient se mélanger pour illustrer le multi-tâches simulé.

4 Les appels systèmes

4.1 La bonne version de EXIT

- Dans un premier temps, complétez la fonction `kill_thread` qui a la charge de supprimer un thread. Faites en sorte que le simulateur s'arrête si il n'y a plus de thread.
- Dans un deuxième temps, utilisez la fonction précédente pour supprimer le thread courant dans `sysc_exit`.

4.2 Création de thread

Complétez la fonction `new_thread` (voir les explications déjà présentes dans le code).

4.3 Appel système de création de thread

Ajoutez ensuite un nouvel appel système `sysc SYSC_NEW_THREAD` qui duplique le thread courant pour en créer un nouveau. Chez le père (l'appelant), le registre `AC` est affecté à `1`, tandis que chez le fils il est forcé à zéro. Le père et le fils continuent leur exécution à la première instruction qui suit l'appel au système.

Voici un exemple d'utilisation :

Exemple de création d'un thread

```
    define SYSC_EXIT          100
    define SYSC_PUTI          200
    define SYSC_NEW_THREAD    300

    // *** créer un thread ***
    sysc SYSC_NEW_THREAD      // créer un thread
    ifgt pere                 // si (AC > 0), aller à pere

    // *** code du fils ***
    set 1000                  // AC = 1000
    sysc SYSC_PUTI            // afficher AC
    nop
    nop

pere: // *** code du père ***
    set 2000                  // AC = 2000
    sysc SYSC_PUTI            // afficher AC
    sysc SYSC_EXIT            // fin du thread
```