TP1 : Simulation d'un processeur et d'un système

Les TP 1 et 2 sont à rendre ensemble au plus tard avant le TP 3. Les modalités seront précisées lors des séances suivantes.

1 Présentation de la machine

Pour suivre ce TP vous devez récupérer le prototype du simulateur [1] 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

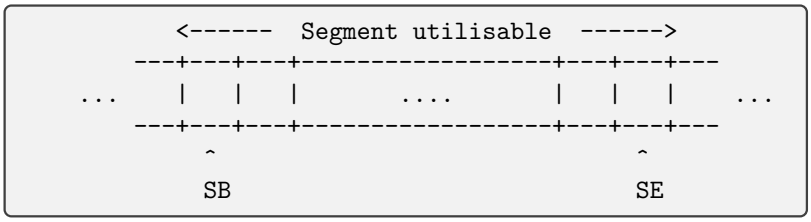
Le mot d'état du processeur est défini comme suit

```
typedef struct PSW { /* Processor Status Word */
    WORD PC;        /* Program Counter */
    WORD SB;        /* Segment begin */
    WORD SE;        /* Segment end */
    WORD IN;        /* Interrupt number */
    WORD DR[8];     /* Data Registers */
    INST RI;        /* Registre instruction */
} PSW;
```

Définition des registres :

- `PC` : Le **Program Counter** (compteur ordinal) est un pointeur sur la prochaine instruction à exécuter (en fait un entier).
- `SB` et `SE` : pointeurs vers le début (`SB`) et la fin (`SE`) du segment de mémoire utilisable :

1. simul.zip



- `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.
- `DR[i]` : Le processeur dispose de huit registres généraux pouvant contenir chacun un entier signé de 32 bits.

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 {
    unsigned op: 10; /* code operation (10 bits) */
    unsigned i:  3;  /* nu 1er registre (3 bits) */
    unsigned j:  3;  /* nu 2eme registre (3 bits) */
    short   arg;     /* argument (16 bits)     */
} INST;
```

Une instruction va ressembler à ceci : `add R1, R2, 1000`. Celle-ci va effectuer l'affectation `R1=R1+R2+1000`. Les instructions disponibles sont détaillées ci-dessous :

Les instructions de notre CPU

```
set Ri, arg        // Ri = arg
add Ri, Rj, arg    // Ri = Ri + Rj + arg
sub Ri, Rj, arg    // Ri = Ri - Rj - arg
load Ri, Rj, arg   // Ri = mem[ Rj + arg ]
store Ri, Rj, arg  // mem[ Rj + arg ] = Ri
nop                // ne rien faire
ifgt Ri, Rj, label // si (Ri > Rj) aller à label
ifge Ri, Rj, label // si (Ri >= Rj) aller à label
iflt Ri, Rj, label // si (Ri < Rj) aller à label
ifle Ri, Rj, label // si (Ri <= Rj) aller à label
jump label         // aller à
sysc Ri, Rj, arg   // appel du système
halt               // arrêter la machine
```

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

Un exemple de boucle avec évolution de R1 par incrément de 2000

```
    define INCR 2000

    set R1, 0                   // R1 = 0
    set R2, INCR                // R2 = INCR
    set R3, 6000                // R3 = 6000
loop:                           // définir loop
    ifgt R1, R3, end            // si (R1 > R3) aller à end
    add R1, R2, 0               // R1 = R1 + R2 + 0
    nop                         // ne rien faire
    jump loop                   // aller à loop
end:                            // définir end
    halt                        // poweroff
```

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) {
    for(PSW mep = system_init();;) {
        mep = cpu(mep);
        mep = process_interrupt(mep);
    }
    return (EXIT_SUCCESS);
}
```

La fonction `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].
2. Décompressez l'archive :

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

3. Compilez le projet :

```
$ cd simul
$ make
```

4. Exécutez le simulateur :

```
$ ./simul
```

2. simul.zip

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`). **Conseil** : utilisez un tube et la commande `less` pour freiner l'exécution :

```
$ ./simul | less
```

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 et les registres indiqueront les éventuels paramètres de cette action.

Préparation :

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

Utilisation :

1. Commencez par structurer la fonction `process_system_call` afin de traiter les différents appels au système :

```
enum {
    SYSC_EXIT = 100, // fin du processus courant
    SYSC_PUTI = 200, // afficher le contenu de Ri
};

static PSW process_system_call(PSW cpu) {
    // suivant l'argument de sysc Ri, Rj, ARG
    switch (cpu.RI.arg) {
        case SYSC_EXIT:
            /* À FAIRE : réaliser l'appel système EXIT */
            break;
        case SYSC_PUTI:
            /* À FAIRE : réaliser l'appel système PUTI */
            break;
        default:
            printf("Appel␣système␣inconnu␣%d\n", cpu.RI.arg);
            break;
    }
    return cpu;
}
```

2. Commencez par implanter l'appel système `SYSC_EXIT` qui provoque l'arrêt du processus demandeur et donc du système puisque nous avons, pour l'instant, un seul processus. Suivez les étapes ci-dessous (toujours dans `system.c`) :
   - définir la macro `SYSC_EXIT` dans votre programme en assembleur :

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

   - remplacer dans le code du programme assembleur l'instruction `halt` par l'instruction d'appel au système ci-dessous

     ```
     sysc R0, R0, SYSC_EXIT  // Appel au système pour SYSC_EXIT
     ```

   - prévoir une fonction `PSW sysc_exit(PSW c)`,
   - prévoir l'appel à `sysc_exit` dans `process_system_call`.
3. Continuez avec l'appel `SYSC_PUTI` qui affiche l'entier stocké dans le premier registre de l'instruction `sysc`. Vous pouvez maintenant placer cette instruction au coeur de la boucle et voir le déroulement de la boucle.

Optionnel Utilisez l'instruction de lecture mémoire (`LOAD`) dans la boucle pour provoquer une erreur de débordement de l'accès mémoire. vérifiez que le résultat obtenu est conforme au paramètre utilisé dans l'initialisation du système (registres `SB` et `SE`).

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 processus pendant un certain temps

3.1 Codage des processus

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

```c
#define MAX_PROCESS (20) /* nb maximum de processus */

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

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

PCB process[MAX_PROCESS]; /* table des processus  */

int current_process = -1; /* nu du processus courant */
int nb_ready      =  0;   /* nb de processus prêts */
```

Faites en sorte qu'au démarrage du système, le système prépare la première case du tableau des processus. Il y a pour l'instant un seul processus.

3.2 Un ordonnanceur simplifié

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

Fonction de l'ordonnanceur

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

Travail à faire :

- 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 processus, il est systématiquement sauvegardé puis choisi pour être exécuté.
- Pour tester votre ordonnanceur, vous pouvez maintenant créer directement deux processus au démarrage du système (prenez l'exemple de la boucle de la section précédente). Utilisez le même segment pour ces deux processus. Ce sont plus des **threads** que des **processus** puisqu'ils partagent leur code et leur données. Les sorties des deux processus devraient se mélanger pour illustrer le multi-tâches simulé.

4 Les appels systèmes

4.1 La bonne version de EXIT

Faites en sorte que l'appel système `SYSC_EXIT` provoque l'arrêt du système quand il n'y a plus de processus (vérifiez notamment que les deux processus arrivent à leur terme).

4.2 Création de thread

Commencez par réaliser une fonction de création d'un thread fils par duplication du thread courant (le paramètre) :

```c
int new_thread(PSW cpu) { ... }
```

4.3 Appel système de création de thread

Ajoutez ensuite un nouvel appel système `sysc Ri,SYSC_NEW_THREAD` qui duplique le thread courant pour en créer un nouveau. Chez le père (l'appelant), le registre `Ri` 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 R1, SYSC_NEW_THREAD // créer un thread
    set R3, 0                // R3 = 0
    ifgt R1, R3, pere        // si (R1 > R3), aller à pere

    // *** code du fils ***
    set R3, 1000             // R3 = 1000
    sysc R3, SYSC_PUTI       // afficher R3
    nop
    nop

pere: // *** code du père ***
    set R3, 2000             // R3 = 1000
    sysc R3, SYSC_PUTI       // afficher R3
    sysc SYSC_EXIT           // fin du thread
```