Projet de Systèmes d'Exploitation 2012

1 Présentation

Le but de ce projet est la mise en œuvre d'un simulateur (léger) des composantes matérielles principales d'un ordinateur usuel fonctionnant sous un système d'exploitation multitâche, en mettant l'accent sur l'unité centrale et sur une implémentation dans le noyau de la communication et synchronisation entre processus. Ce simulateur offre des éléments de programmation machine (assembleur fictif simplifié, gestion d'interruptions) et des éléments de programmation système (gestion de plusieurs processus, mémoire partagée et sémaphores).

Le noyau est le processus (sim) de PID 0, et offre aux processus des services sous la forme de plusieurs interruptions : élire le processus suivant qui est prêt (`int1`, très simple round robin), terminer le processus en cours (`int2` – un peu comme `exit()`), opération sur les sémaphores (`int4`), entrées-sorties "console" (`int5` et `int6`) ; l'interruption `int3` est réservée pour l'ordonnanceur.

Tous les processus sont chargés au début (il n'y a pas de shell) et exécutés un par un pas à pas jusqu'à leur fin our jusqu'à des requêtes d'opération de sémaphores bloquantes. Chaque processus dispose de sa propre tranche de mémoire vive (sim) "virtuelle" de longueur fixe, y compris le noyau (qui peut également accéder aux tranches des processus.

De surcroît, il y a une autre tranche de mémoire – partagée entre tous les processus.

Pour chaque type de mémoire il y a des instructions assembleur spécifiques, pour simplifier leur manipulation (e.g. `LDMEM`, `LDSHM`, etc.)

Le cadre de programmation, décrit plus bas, vous est fourni en état de marche et vous devez l'enrichir. Des classes représentent chaque élément décrit plus haut ; la mémoire vive est logée en entier dans un seul fichier, avec des offset pour chaque tranche ; les textes des programmes (sim) (des processus) sont des simple strings, gérés par des classes dédiées, la mémoire étant uniquement pour les données, pour simplifier le tout.

Ce cadre permet donc déjà tel qu'il est la compilation et son exécution, pour comprendre son fonctionnement et tester les fonctionnalités déjà fournies, à l'aide d'exemples inclus.

À la fin de ce document une section détaille votre travail à faire.

2 Tester un peu

- désarchivez et compilez (`make nom=System`, qui contient le `main()`)
- lancez le tout (`System.run`) et regardez l'écran, et/ou gardez-en une copie dans un fichier également (par exemple avec `tee`, ou simplement en dirigeant la sortie dans le fichier – n'oubliez pas de récupérer la sortie erreur standard également).
- le tout doit finir, avec un dump de la mémoire

3 Déroulement détaillé pas-à-pas

Au démarrage du simulateur, après construction des objets et chargement du code de tous les programmes (noyau et processus normaux), c'est le noyau qui est démarré (`Proc_0_Program.asm`). Celui-ci

- arrange sa mémoire, en y mettant
  - vecteur d'interruptions (codé en dur dans le texte du programme, $prep),
  - tables états processus (remplies avec `readytorun`),
  - vecteur avec adresses de début des vecteurs de requêtes de semop par processus
- se branche sur `int1`
  - cherche le premier processus P en état `readytorun`
  - change son état en `running` dans la table
  - se branche sur le code de P (passant en mode utilisateur) et commence ainsi son exécution (instruction `LDPSW` qui charge le PSW – Program Status Word – l'état de l'unité centrale (les registres, etc.) correspondant au processus P)

Lorsque le processus P en cours fait une requête noyau (donc appelle une interruption, avec l'instruction `CLINT`) :

- l'état de l'unité centrale est sauvegardé automatiquement (i.e. le PSW)
- le pid de P est noté dans la mémoire vive du noyau
- on se branche à nouveau sur le code du noyau, dans l'interruption en question (et en mode maître), qui, dans la plupart des cas
  - prépare les paramétres de la requête
  - l'effectue
  - se branche éventuellement sur `int1` pour essayer de trouver un autre processus à exécuter

4 Structure

La partie programmation s'appuie sur plusieurs classes qui vous sont fournies.

Les fichiers `Board.h` et `Board.cxx` contiennent les classes qui représentent la mémoire vive (Memory) et un prototype d'unité centrale (BaseCPU) avec ses registres (généraux, spécialiés, etc.). Ces fichiers sont en état d'utilisation tels quels et n'ont pas nécessairement besoin d'apport pour ce projet.

Les fichiers `CPU.h` et `CPU.cxx` représentent l'unité centrale de notre système. Nous ne simulons pas la notion de bus avec une classe séparée : la classe BaseCPU "possède" également l'objet représentant la mémoire vive.

5 Mémoire

La mémoire vive du simulateur est implémentée à l'aide d'un seul fichier Unix normal, et donc permet également un accès concurrent si besoin est, à l'aide même de plusieurs processus (vrais – Unix). Le code qui vous est fourni n'en a toutefois qu'un seul.

Chaque processus du simulateur dispose d'une tranche de 1000 entiers (donc 1000*sizeof(int) octets), qu'il voit de 0 a 999 (mémoire "virtuelle"). Ces tranches sont en fait simplement logées une après l'autre dans le grand fichier unique (adresses en mots – taille d'un `int`) :

```
AddrFic  AddrProc   MemProc      Utilisation
-----------------Debut du fichier-------------------

   0       0       0 (noyau)     <speciale>
   10     10       0                ...
   ....   ....     0
   990   990       0

- - - - - - - - - - - fin tranche - - - - - - - - -

  1000     0       1         <mini-zone-u-dot>
  1010    10       1
   ....   ....     1                ...
  1200   200       1    <debut zone disponible>
   ....   ....     1                ...
  1990   990       1 <start pile (croit vers debut)>

- - - - - - - - - - - fin tranche - - - - - - - - -

  2000     0       2         <mini-zone-u-dot>
  1010    10       2          <detail +bas>
  1020    20       2                ...
   ....   ....     2
  1200   200       2    <zone disponible>
   ....   ....     2                ...
  1990   990       2 <start pile (croit vers debut)>

- - - - - - - - - - - fin tranche - - - - - - - - -
       ...         ...       ...
- - - - - - - - - - - fin tranche - - - - - - - - -
   Memoire Partagee (visible par tous les processus)
-----------------Fin du fichier-------------------
```

Donc la tranche du processus simulateur p commence dans le fichier mémoire à l'offset réel en octets $p * 1000*\texttt{sizeof(int)}$.

Ces constantes (parmi d'autres similaries) sont définies au début du fichier `CPU.h` :

```
#define MEMORYSIZEPERPROCESS 1000
#define KERNELSAVECURRENTPROCID 0
#define KERNELPID               0
#define PROCPSWFRAMESTART       0
```

6 L'unité centrale

- a un jeu de registres à usage général (les `Rn`)
- a des registres spécialisés : le mode (user ou master), le pointeur de pile, le compteur ordinal (program counter), le compteur de processus (process counter), le drapeau d'interruptions, et les registres d'entrée pour les communications.

```
class BaseCPU {
protected:
    InstructionParser iPrs;
    StackPointer     spReg;
    ProgramCounter   pcReg;
    ProcessCounter   prReg;
    RunMode          mdReg;  // user or master
    RegisterSet      genReg;
    Register         inReg0,inReg1; // for interCPU communication (e.g. CPU<->DMA)
    MemoryTable      mem;
    Memory           sharedMem;
    ProcessTable     proc;
    string           cpuId;
    ostream         &logStream;
    ostream         &consoleOutputStream;
```

```
    istream         &consoleInputStream;
    ConsoleInOut     consoleInOut;
    bool             qRun;
    bool             qInterruptible;
public:
    BaseCPU(const string &id, ostream  &log, const int nGenReg, const int nProc);
    virtual void dumpReg(ostream &os, bool qRegAsWell = false);
    virtual void execute(const Instruction &) throw(nsSysteme::CExc);
    virtual void run() throw(nsSysteme::CExc);
    virtual void interrupt(const int, bool) throw(nsSysteme::CExc);
    virtual void pendingIntIfAny() throw(nsSysteme::CExc);
};
```

– la boucle principale d'exécution se trouve dans la méthode `run()`

```
void BaseCPU::run() throw(CExc) {
    for(iTick = uTick = 0; qRun && iTick < 10000; uTick += (mdReg . getVal() != 1),iTick++)
      execute(pcReg . fetch(proc[prReg . getVal()] . programText));
      if(qInterruptible) { pendingIntIfAny(); }
    }
    logStream << "CPU " << cpuId << " is now fully stopped, total " << iTick << " instructi
}
```

– sait exécuter les instructions qui sont décrites en détail dans `Board.h`

```
enum InstructionType {

NOPER=0,
DMPRG, // ex: DMPRG              ; dump content of all registers to logstream
MSTRM, // ex: MSTRM              ; MD <- MasterMode
USERM, // ex: USERM              ; MD <- UserMode
NINTR, // ex: NINTR              ; disable interrupts
INTRP, // ex: INTRP              ; enable interrupts
SETRI, // ex: SETRI R2 254       ; R2 <- 254
SETRG, // ex: SETRG R2 R3        ; R2 <- R3
SETPR, // ex: SETPR R10          ; PR <- R10 (only in mastermode)
CPPRG, // ex: CPPRG R2           ; R2 <- PR
ADDRG, // ex: ADDRG R2 R3 R4     ; R2 <- R3 + R4
SUBRG, // ex: SUBRG R2 R3 R4     ; R2 <- R3 - R4
PSHRG, // ex: PSHRG R5 R4        ; mem[(SP - R5) * sizeof(int)] <- R4
POPRG, // ex: POPRG R5 R7        ; R7 <- mem[(SP + R5) * sizeof(int)];
JMABS, // ex: JMABS R1           ; PC <- R1
JMPTO, // ex: JMPTO R1           ; PC <- PC + R1
JZERO, // ex: JZERO R12 R5       ; if(R12 == 0) { PC <- PC + R5 }
JNZRO, // ex: JNZRO R12 R5       ; if(R12 != 0) { PC <- PC + R5 }
JMBSI, // ex: JMBSI 23           ; PC <- 23
JMTOI, // ex: JMTOI 24           ; PC <- PC + 24
JZROI, // ex: JZROI R12 25       ; if(R12 == 0) { PC <- PC + 25 }
JNZRI, // ex: JNZRI R12 26       ; if(R12 != 0) { PC <- PC + 26 }
LDMEM, // ex: LDMEM R8 R21       ; R21     <- mem[R8] (for current proc memory space)
STMEM, // ex: STMEM R8 R21       ; mem[R8] <- R21  (for current proc memory space)
LDSHM, // ex: LDSHM R8 R21       ; R21     <- sharedMem[R8] (for all procs shared memory sp
STSHM, // ex: STSHM R8 R21       ; sharedMem[R8] <- R21  (for all procs shared memory space
LDPSW, // ex: LDPSW R3           ; <SP,PR,PC,MD,R0,R1,...> <- memOfProc[R3][PROCPSWFRAMESTA
SPSWR, // ex: SPSWR R3 R2 R4     ; memOfProc[R3][PROCPSWFRAMESTART+offset4.R2] <- R4 (maste
LDPRM, // ex: LDPRM R3 R8 R21    ; R21     <- memOfProc[R3][R8] (master only)
STPRM, // ex: STPRM R3 R8 R21    ; memOfProc[R3][R8] <- R21 (master only)
WKCPU, // ex: WKCPU R3 R5 R6     ; CPU[R3].I0 <- R5 ; CPU[R3].I1 <- R6 ;  signal CPU[R3]
GETI0, // ex: GETI0 R5           ; R5 <- I0 (the only way to read I0 (after a WKCPU))
GETI1, // ex: GETI1 R5           ; R5 <- I1 (the only way to read I1 (after a WKCPU))
CLLSB, // ex: CLLSB R7 R5        ; SP <- SP - R7; mem[SP * sizeof(int)] <- PC; PC <- R5
RETSB, // ex: RETSB R6           ; PC <- mem[SP * sizeof(int)] ; SP <- SP + R6
CLINT, // ex: CLINT R7           ; PC <- IntVec[R7] (saves the PSW except for the kernel)
SPECP, // ex: SPECP R2 R3 R10 R4 ; special instruction given by R2 with (at most) three arg
SCRASH,// ex: SCRASH             ; crash down the system (master only) -- major inconsisten
SDOWN};// ex: SDOWN              ; shut down system (master only)
```

– gère des interruptions avec la méthode `CPU::interrupt()` : une interruption peut donc soit "survenir" (cause externe – ici ce sera principalement l'"alarme" du scheduler), soit être appelée par un processus utilisateur (instruction `CLINT`), pour demander un service (sortie console (interruption 5), terminaison de programme (interruption 2), etc.).

– sauvegarde son mot d'état du programme (PSW – Program Status Word) – voir méthode `CPU::interrupt()` dans `CPU.cxx`.

7 Remarques

- les instructions ne manipulent pas des registres spéciaux par leur nom (i.e. en tant qu'arguments), mais certaines instructions modifient tout de même directement les valeurs de ces registres spéciaux, comme `SETPR` (en mode maître).
- les instructions machine reconnues par la CPU travaillent beaucoup uniquement avec des registres et peu avec des valeur numérique (on les appelle des instructions "immédiates" : `SETRI`, `JNZRI`, etc.).
- puisqu'on ne se concentre pas sur des calculs approfondis, les instructions arithmétiques sont très peu nombreuses (additions, soustractions).
- toutes les valeurs d'addressage dans la mémoire vive (au niveau langage machine de ce projet) représentent des entiers (des "word"). Par contre, au niveau implémentation (donc par exemple dans la classe `Memory`), les adresses désignent des octets. Dans la méthode `CPU::execute()` on prend soin de multiplier par `wordSize` – la taille d'un entier – les valeurs qu'on récupère des registres avant d'appeler les méthodes de la classe `Memory` pour l'accès effectif.
- pour les sauts (`JNZRO`, etc.), le `PC` pointe déjà à l'instruction qui suit le saut, donc ceci est à prendre en compte lors du calcul du nombre d'instructions à sauter lors de sauts relatifs.
- pour les appels de procédure on dispose de `CLLSB` et `RETSB` dont le premier argument donne la taille de la zone de paramètres (avec l'adresse-programme de retour) sur la pile. Pour empiler/récupérer des arguments, on a `PSHRG` et `POPRG`.
- le code des programmes qui sont en cours d'exécution (i.e. les processus) est stocké ailleurs que les données – la classe `Memory` sert uniquement pour les données.
- le code ("machine") des programmes est en fait stocké sous forme de chaînes de caractères (ceci aide également la lisibilité)
- le registre `PC` (program counter) est un indice dans ce tableau de chaînes
- Un Proc_0 plus lisible se trouve dans le sous-répertoire `ini` – mais ce programme ne représente pas ce que la machine virtuelle voit. Pour des programmes en assembleur/code machine un peu plus larges il est plus facile de travailler de manière symbolique, au moins pour les branchements, et donc le programme du sous-répertoire `ini` est celui-ci, d'un peu plus "haut niveau". Pour le transformer en "vrai code machine" pour notre machine virtuelle il faut utiliser le script Perl `preProcAsm.pl` (en tant que filtre, donc par exemple ainsi

  ```
  cat ini/Proc_0_Program.asm | ./preProcAsm.pl > Proc_0_Program.asm
  ```

8 Travail à faire

1. Étude générale – cherchez sur le web et dans des livres des renseignements sur les points suivants et rédigez **individuellement** ensuite un rapport d'environ quatre pages, *à rendre pour le 18 novembre 2012, 23h59 heure de Paris* :
   (a) schéma de base de gestion d'un seul processus dans le noyau
   (b) schéma de base du noyau (ordonnanceur, gestion mémoire (virtuelle), communication entre processus, pilotes de matériel (device drivers), etc.)
   (c) concrètement – définissez les notions suivantes, et expliquez-en le but, les coûts, les particularités, leur place dans le cadre du système d'exploitation, etc. :
      i. registres de l'unité centrale (CPU) et mémoire RAM
      ii. changement de contexte
      iii. interruption, appel système
      iv. mode utilisateur et mode maître
      v. espace mémoire d'un processus
      vi. mémoire partagée et exclusion mutuelle
   (d) si/comment les concepts ci-avant sont-ils disponibles en C (ou C++) dans les systèmes de type Unix, comme Linux, etc. ?
2. Compréhension du code fourni (l'implémentation des instructions machine est dans `CPU::execute()`) – travail en équipe, à rendre ensemble avec le code (voir point suivant), pour le 23 décembre 2012, 23h59 heure de Paris :
   (a) Quelle est la différence entre `STMEM` et `STPRM` ? Pourquoi vaut-il mieux permettre la seconde instruction uniquement en mode maître ? Pourquoi suffit-il pour le noyau de faire `LDPSW` d'un processus P pour "se brancher sur le code de P" ?
   (b) Détaillez pas à pas ce que fait `int2` (comme est décrit le déroulement de chacune des trois autres dans ce document).
   (c) Détaillez la structure de la tranche de mémoire de chaque processus (regardez la méthode `CPU::interrupt()` et l'instruction `LDPSW`, ainsi que le code noyau de `int3`, et enfin les instructions manipulant `spReg` ainsi que `Memory::setUp()`)
   (d) À quoi sert la ligne 168 de `Proc_0_Program.asm` ?
   (e) Détaillez la structure de la tranche de mémoire du noyau (regardez son code dans `Proc_0_Program.asm` et la ligne

   ```
   mem[KERNELPID] . storeAt(KERNELSAVECURRENTPROCID, prReg . getVal());
   ```

   de `CPU::interrupt()`).
   (f) Quelles sont les limitations (nombre de processus, nombre de sémaphores, de semop()s, etc.) présentes dans ce projet, telles qu'elles sont imposées par la structure de la mémoire et par les valeurs de ces constantes définies dans le code.
3. Construction code – travail en équipe, à rendre ensemble avec le code (voir point précédent), pour le 23 décembre 2012, 23h59 heure de Paris :
   (a) Observez que l'interruption 5 en fait ne peut pas afficher plus d'un nombre ou un caractère – elle n'est pas "vectorisée" – elle ignore la valeur du registre `R2`, lequel est pourtant censé être un de ses paramètre. La première tâche consiste donc à réécrire cette interruption `int5` pour qu'elle soit vectorisée, donc qu'elle prenne bien en compte `R2` également – vous devez donc rajouter une boucle, etc.

(b) Écrire l'interruption `int6` qui permet de lire depuis le clavier des nombres ou des caractères, donc de manière complémentaire à ce que fait l'interruption `int5`.

(c) Élargir les programmes `Proc_1_Program.asm` et `Proc_2_Program.asm` (qui forment un couple producteur-consommateur) pour faire écrire les nombres de 1 a 10 en mémoire partagée depuis l'addresse 11 á l'adresse 20, et les faire lire, mais en écrivant un nombre à la fois dans la zone critique et en relâchant donc les sémaphores. (n'oubliez pas que vous avez plusieurs sémaphores à votre disposition)

(d) Étendre l'infrastructure de ce système (en écrivant du C++ et aussi en améliorant ensuite `Proc_0_Program.asm`)
- rajouter la possibilité d'avoir des nombres aléatoires, avec une lecture memory-mapped dans le noyau, similaire aux entrées-sorties console (bien entendu, utilisnt une autre adresse),
- augmenter le scheduleur pour le randomiser ensuite,
- rajouter une gestion CPU multi-core
- ...