Architecture des Systèmes d'Exploitation
Maîtrise d'Informatique

Université de Lille 1 SCIENCES ET TECHNOLOGIES

Juin 2015

Devoir Surveillé

– tous documents autorisés –

Le langage C est requis pour les implémentations

Les différences parties peuvent être traitées indépendamment, tous les documents sont autorisés, mais ne perdez pas trop de temps à chercher l'information utile. Notez encore que l'énoncé contient 26 points, je m'arrêterais à 20/20 ;-)

Partie I. Mémoire virtuelle (14 points)

Nous nous intéressons ici au support d'une mémoire virtuelle d'un espace d'adressage de 32 bits qui permet pourtant d'adresser une mémoire physique d'un espace d'adressage de 36 bits. Le mécanisme de mémoire virtuelle étudié est basé sur une TLB, proche de celle manipulée en TP.
Le fonctionnement de la MMU que nous considérons est basé sur l'exploitation d'une TLB comme indiqué ci-dessous :

1. Le circuit décode une adresse placée par le microprocesseur sur le bus d'adresse.
2. En ne considérant que les 20 bits de poids fort (qui forment le numéro de page virtuelle) sur les 32 bits de l'adresse virtuelle, il cherche une entrée dans la TLB.
3. Si une entrée est trouvée alors le circuit forme l'adresse physique en remplaçant les 20 bits de poids fort de l'adresse virtuelle par les 24 bits d'équivalence trouvés dans la TLB. Ainsi sont formés les 36 bits d'adresse physique : les 24 bits de poids forts viennent de l'entrée trouvée dans la TLB et les 12 bits de poids faible de l'adresse virtuelle sont conservés tel quel dans l'adresse physique.
4. Si le numéro de page virtuelle n'est pas trouvé dans la TLB, le circuit repositionne le PC du microprocesseur sur le début de l'instruction en cours et génère une interruption.

En cas d'interruption, l'OS peut consulter l'adresse ayant généré celle-ci (dans la variable/registre MMU_FAULT_ADDR) et insérer une nouvelle entrée dans la TLB (en utilisant la variable/registre tlb_entry TLB_ADD_ENTRY de 47 bits, décrite ci-après).

Pour information, nous donnons l'extrait du fichier hardware.ini associé :

```
#
# Configuration de la MMU
#
MMU_ENABLE      = 1     # MMU disponible
MMU_IRQ         = 13    # niveau d'interruption associé à la MMU
MMU_CMD         = 0xEC  # commande => 0 : désactive la MMU
                        #             1 : reset la MMU (et réactive)
MMU_FAULT_ADDR = 0xF0   # registre contenant l'adresse accédée
```

```
MMU_FAULT_ID    = 0xF4   # registre à 0 : faute d'accès en lecture
                         #             1 : faute d'accès en écriture
                         #             2 : faute d'accès en exécution
TLB_ADD_ENTRY   = 0xF8   # registre de commande "ajout" dans la TLB
TLB_DEL_ENTRY   = 0xFC   # registre de commande "retrait" de la TLB
TLB_ENTRIES     = 0x100  # table des entrées de la MMU
```

La fonction `int _in(long int port);` réalise la lecture sur le port désigné. La valeur retournée correspond à la valeur qui a été lue sur ce port matériel. Les numéros de port sont identifiés dans le fichier de configuration du matériel `Hardware.ini`.
La fonction `void _out(int port, long int value);` réalise l'écriture d'une valeur sur le port désigné.
Il est à noter que les opérations `_in` et `_out` prennent ici un « `long int` » qui est, en pratique un mot de 64 bits.
Il est aussi rappelé que `void _sleep(int irqlvl);` suspend l'activité du microprocesseur jusqu'à l'occurrence d'une interruption de niveau au moins égal à `irqlvl`. Par ailleurs `IRQVECTOR` est un tableau de pointeur de fonction (de la forme `void func();`). Lorsqu'une interruption de niveau `n` est déclenchée par le matériel, c'est la fonction `IRQVECTOR[n]();` qui est exécutée, puis le programme en cours reprend son activité normale. Enfin `void _mask(int irqlvl);` permet d'inhiber l'appel à la fonction `IRQVECTOR[irqlvl]()` sur l'occurrence d'une interruption de niveau inférieure ou égale a `irqlvl`.

La TLB considéré ici manipule des entrées de 47 bits organisaient comme suit :

- Numéro de page virtuelle noté `vrt` : 20 bits
- Numéro de page physique associée noté `phy` : 24 bits
- Droit d'accès en lecture noté `ac_r` : 1 bit
- Droit d'accès en écriture noté `ac_w` : 1 bit
- Droit d'accès en exécution noté `ac_x` : 1 bit

Question I.1 – Découpage de la mémoire – 1 point

Quel est la taille, en octet, d'une page de mémoire (virtuelle ou physique) selon le fonctionnement de la TLB que nous considérons ici ?

Question I.2 – Espace d'adressage virtuel – 1 point

Selon la description de l'entrée de TLB qui vous est donné, ci-dessus, quel est, en octet, la taille de l'espace d'adressage virtuel que le microprocesseur peut adresser ?
N.B. chaque adresse sollicitée par le microprocesseur est associée à 1 octet.

Question I.3 – Espace d'adressage physique – 1 point

Selon la description de l'entrée de TLB qui vous est donné, ci-dessus, quel est, en octet, la taille de l'espace d'adressage physique que le microprocesseur peut adresser ?
N.B. chaque adresse sollicitée par le microprocesseur est associée à 1 octet.

Question I.4 – Manipulation du matériel – 1 point

Définissez une structure C qui respecte ce découpage en champs de bits.

Question I.5 – Mapping statique – 2 points

Dans un premier cas d'utilisation, il est décidé que la MMU sera utilisée simplement pour découper la mémoire virtuelle selon 3 segments distincts :
– De 0x00800000 à 0x0081A000 : Code exécutable (et seulement exécutable)
– De 0x00900000 à 0x00902000 : Données manipulées en lecture/écriture
– De 0x00A00000 à 0x00A01000 : Données manipulées en lecture seulement
La mémoire physique disponible quant à elle commence en 0x000000000 et finit en 0x00001D000. Au dela, bien que les adresses physiques puissent être formées, aucune barrette de RAM n'est connectée. Par convention nous décidons que les pages de 0x00000 à 0x0001A seront associées au code exécutable, les pages 0x0001B à 0x0001C seront associés aux données manipulées en lecture/écriture et la page physique 0x0001D aux données manipulées en lectures seulement.

Proposez une procédure `_itdecl void tlbIrq(void)` qui sera attachée à l'interruption MMU_IRQ et qui convertira les adresses virtuelles en adresses physiques selon le schéma précédent. La fonction, vérifiera que l'adresse appelée est valide, sinon elle terminera le programme en affichant un message d'erreur. Si l'adresse est valide, l'interruption programmera la TLB de façon à ce que l'adresse virtuelle soit associée à une adresse physique et afin que les limitations d'accès (exécution, lecture, écriture) soient contrôlées par la MMU.

Question I.6 – Allocation de pages physique – 2 points

De manière moins statique, un OS gère en règle générale un ensemble de pages physique libres qu'il associe aux espaces d'adressages virtuels qu'il crée pour chaque processus. Ainsi les pages physiques sont allouées à la demande par le noyau du système d'exploitation.
Nous nous proposons d'implémenter une stratégie d'allocation des pages de mémoire physique, simple, basée sur le principe qu'une variable globale void *firstFreePage donne l'adresse du premier octet de la première page de mémoire physique libre, et qu'à cette adresse on trouve l'adresse de la page libre suivante (les pages libres sont chainées les unes aux autres, comme cela a été fait en TP pour les blocs du disque dur).

Proposez une implémentation de la procédure `void* allocPage(void);` qui retourne l'adresse d'une page de mémoire physique libre (et qui la retire de la liste des pages libres).
Proposez ensuite une implémentation de la procédure `void freePage(void* ptr) ;` qui permet de libérer une page de mémoire physique dont on donne l'adresse du premier octet dans `ptr`.

Question I.7 – Arbre de translation d'adresse – 2 points

Chaque processus supporté par l'OS gère un arbre de translation qui lui permet de faire la translation entre adresse virtuelle et adresse physique. L'idée est la suivante : une page de mémoire physique associée au processus contient une table de 1024 entrées. Ces 1024 entrées correspondent à 1024 adresses physiques sur des pages qui sont autant de tables d'indirection de second niveau. Chacune de ces tables contient donc 1024 entrées : 1024 adresses de pages physiques. Lorsqu'une adresse virtuelle doit être convertie en adresse physique, l'adresse virtuelle est décomposée en 3 fragments :

- *indirection1* : les 10 bits de poids fort de l'adresse virtuelle
- *indirection2* : les 10 bits suivants
- *déplacement* : les 12 bits de poids faible de l'adresse virtuelle

En considérant l'entrée *indirection1* de la table d'indirection associée au processus (qui est stockée dans une page et une seule, de mémoire physique), le système d'exploitation trouve l'adresse de la table de second niveau (chaque table de second niveau tenant, elle aussi, sur une page et une seule de mémoire physique). En considérant l'entrée *indirection2* de cette table, il trouve l'adresse physique du premier octet de page physique associée à cette adresse virtuelle pour le processus courant. Il est évident que toutes les adresses virtuelles ne sont pas associées à des adresses physiques dans chaque processus. Aussi, bon nombre d'entrées des tables de premier ou second niveau sont nulles. Si un accès à une mémoire si un accès à une page mémoire non mappée a lieu, c'est une « erreur d'accès mémoire » (ou segmentation fault) pour le processus.

Donnez la déclaration des structures de données qui permettent de mettre en œuvre cette translation d'adresses. Il s'agira donc de donner la déclaration de la structure `TransTable` associée aux indirections de premier niveau, puis la déclaration de la structure `SubTransTable`, associée aux indirections de second niveau, et enfin de déclarer la variable globale `current_TransTable` qui donne l'adresse de la table de translation d'adresses courante (associée au processus courant).
Il est à noter que chaque structure, destinée à être stockée dans une page de mémoire physique, ne doit pas en excéder la taille.

Question I.8 – Création d'un espace d'adressage virtuel – 2 points

Lorsque l'OS crée un nouveau processus, il doit créer un nouvel arbre de translation.

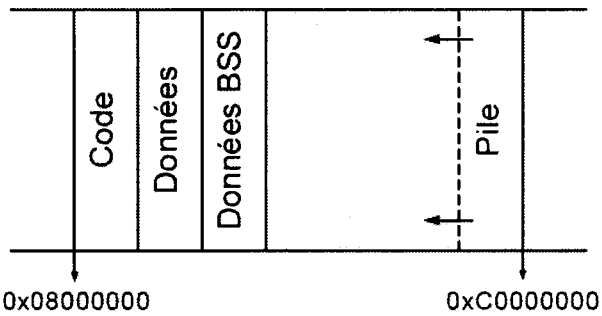

Fig. 1 – Le modèle de mémoire virtuelle utilisé par Linux.

Selon le modèle de mémoire virtuelle de Linux, l'espace d'adressage virtuel d'un processus est composé de 4 parties :
La première partie, destinée à stocker le binaire exécuté par le processus, débute à l'adresse virtuelle 0x08000000 et est associée à un ensemble de pages de mémoire physique (allouées via `allocPage`).
Ensuite viennent les données initialisées du programme, puis les données non initialisées.
Enfin, la pile d'exécution est initialisée à partir de l'adresse 0xC000000 mais conformément au fonctionnement des processeurs Intel, cette pile est décroissante.

Proposez le corps de la procédure :
`int allocTranslators(int codeSize, int dataSize, int bssSize, int stackSize);`
en respectant le modèle d'organisation de la mémoire virtuelle utilisé par Linux, illustré par la Fig. 1. Cette procédure prend en paramètre le nombre d'octets associés à chaque partie de la mémoire virtuelle identifiée sur la figure 1.
Cette procédure alloue et initialise un arbre de translation ainsi que les pages de mémoires physiques associées à chaque page de mémoire virtuelle du modèle utilisé par Linux. Le seul allocateur de mémoire physique disponible est celui implémenté par la question I.4. L'adresse de la structure de donnée ainsi allouée est ensuite enregistrée dans la variable globale `current_TransTable`. La valeur retournée par la fonction est zero en cas de problème d'allocation ou 1 si le processus s'est bien passé.

Question I.9 – Translation d'adresses – 2 points

Proposez une nouvelle implémentation de la procédure `_itdecl void tlbIrq()` qui convertit les adresses virtuelles en adresses physiques selon le nouveau schéma, c'est-à-dire en utilisant l'arbre de translation d'adresse référencé depuis la variable globale `current_TranslateTable`.

Partie II. Fiabilité du partage du microprocesseur (12 points)

Nous nous intéressons, dans cette partie à la mise en œuvre d'un ordonnanceur de tâches sur un système disposant d'un seul microprocesseur. Pour cela le matériel propose un support pour l'horloge interne dont les registres d'accès sont décrits dans l'extrait du fichier `Hardware.ini` ci-dessous :

```
#
# Configuration de l'horloge interne
#
TIMER_CLOCK = 0xC4 # registre de lecture de la date courante
TIMER_ALARM = 0xC8 # registre TIMER
TIMER_PARAM = 0xCC # registre de configuration du TIMER
                   # bit 4 : RESET general (=1)
                   # bit 3 : Alarm ON = 1, Alarm OFF = 0
                   # bit 2 : Déclenche la division Hz du Timer (=1)
                   # bit 1 \ Si la division Hz du timer est demandé :
                   # bit 0 / 00: 1 top alarme pour 2 top d'horloge,
                   #         01: 1 top alarme pour 16 tops d'horloge,
                   #         10: 1 top alarme pour 128 tops d'horloge,
                   #         11: 1 top alarme pour 512 tops d'horloge.
TIMER_IRQ   = 2    # Niveau d'interruption de l'horloge
```

La fonction `int _in(int port);` réalise la lecture sur le port désigné. La valeur retournée correspond à la valeur qui a été lue sur ce port. Les numéros de port sont identifiés dans le fichier de configuration du matériel `Hardware.ini`. La fonction `void _out(int port, int value);` réalise l'écriture d'une valeur sur le port désigné.

De plus, `void _sleep(int irqlvl);` suspend l'activité du microprocesseur jusqu'à l'occurrence d'une interruption de niveau au moins égal à `irqlvl`. Par ailleurs `IRQVECTOR` est un tableau de pointeur de fonction (de la forme `void func();`). Lorsqu'une interruption de niveau `n` est déclenchée par le matériel, c'est la fonction `IRQVECTOR[n]();` qui est exécutée, puis le programme en cours reprend son activité normale. Enfin `void _mask(int irqlvl);` permet d'inhiber le déclanchement de l'interruption et donc l'appel à la fonction associée : `IRQVECTOR[irqlvl]()` sur l'occurrence d'une interruption de niveau inférieure ou égale a `irqlvl`.

L'horloge interne est incrémentée 4 770 000 fois par seconde (on dit que l'horloge est cadencée à 4,77MHz). Il est possible de positionner la valeur (16 bits) du registre TIMER. Cette valeur sera périodiquement incrémentée si le bit 3 du registre de configuration du TIMER_PARAM est positionné à 1. Par défaut le registre TIMER_ALARM est incrémenté en même temps que le registre TIMER_CLOCK. Cependant cette incrémentation peut avoir lieu tous les 2, 16, 128 ou même 512 tops d'horloges. Le « diviseur » d'incrémentation est fixé par les bits 0 et 1 du registre TIMER_PARAM à condition que le bit 2 soit positionné. Si l'incrémentation du registre TIMER_ALARM produit comme résultat la valeur 0xFFFF alors l'horloge interne génère une interruption (niveau 2). A l'incrément suivant le registre TIMER_ALARM vaut 0x0000 puis 0x0001…

Question II.1 – 2 points

Proposez fonction `void init_timer()` qui initialise l'horloge interne de manière à produire une interruption au bout de 20 millisecondes. Expliquez le calcul qui vous

a amené à configurer les registres de l'horloge interne tel que vous les avez initialisés. Enregistrez la fonction `void irq_timer()` de tel manière qu'elle soit appelée au bout de 20 millisecondes.

Question II.2 – 1 point

Expliquez pourquoi la configuration proposée par la fonction que vous avez proposé en Question I.1, `void init_timer()`, ne suffit pas pour que `irq_timer()` soit appelée **toutes les** 20 millisecondes. Donnez une implémentation de `irq_timer()` qui assure que cette fonction soit automatiquement rappelée toutes les 20 millisecondes.

Question II.3 – 1 point

Nous supposons maintenant que nous disposons de la fonction C d'ordonnancement `void yield()` dont l'implémentation a été vue en TD/TP. Cette fonction ne délivre un résultat correct que si elle n'est pas interrompue par elle-même. Proposez une solution pour assurer qu'aucune interruption n'aura lieu pendant l'exécution de `yield()`.

Question II.4 – 1 point

Donnez une nouvelle implémentation de `void irq_timer()` de tel sorte que toutes les 20 millisecondes l'ordonnanceur change la tâche active (via un appel à `yield()`).

Question II.5 – 1 point

Le code suivant fournit un mécanisme d'exclusion mutuelle. En considérant un mécanisme d'ordonnancement implicite (la procédure `yield()` qui active une nouvelle tâche est appelée périodiquement sous interruption) tel que celui vu en TD, expliquez pourquoi les procédures `mutex_request` et `mutex_release` ci-dessous peuvent ne pas fonctionner normalement à cause d'une interruption.

```
struct ctx_s { ...
  struct mutex_s *mutexList; /* mutex owned by the ctx             */
  struct ctx_s *lockList;    /* next task locked on the same mutex */
} ;
struct ctx_s currentCtx;     /* the current task                   */
...
struct mutex_s {
 int locked;                 /* TRUE when the mutex is locked      */
 struct ctx_s *owner;        /* the owner ctx of this mutex        */
 struct ctx_s *lockList;     /* List of tasks locked on the mutex  */
 struct mutex_s *mutexList;  /* next mutex owned by the same ctx   */
} ;
void create_mutex(struct mutex_s *mutex){       /* create a mutex */
  mutex->locked = 0;            /* the mutex is unlock            */
  mutex->owner = NULL;          /* nothing own it                 */
  mutex->lockList = NULL;       /* nothing is locked by mutex     */
  mutex->mutexList = NULL; /* mutex is not in a ctx requested List */
}
void mutex_request(struct mutex_s *mutex) {       /* lock a mutex */
  if(mutex->locked) {           /* if the mutex is not available  */
    currentCtx->state = Blocked;/* current task is suspended and  */
                /* the current task is added in the mutex lockList */
    currentCtx->lockList = mutex->lockList;
    mutex->LockList = currentCtx;
```

```
    yield();                        /* schedule another task           */
  } else {                          /* else, if the mutex is available */
    mutex->locked = 1;              /* the mutex is no more available  */
    mutex->owner = currentCtx;      /* currentCtx is the mutex owner   */
                  /* the mutex is added to the mutexList of the ctx */
    mutex->MutexList = currentCtx->mutexList;
    currentCtx->mutexList=mutex;
} }
void mutex_release(struct mutex_s *mutex) {       /* unlock a mutex */
  if(mutex->locked) {               /* if the mutex is not available   */
          /* remove the mutex form the mutexList requested by ctx */
    removeFromMutexList(currentCtx, mutex);
    mutex->owner = NULL;
    if(!mutex->lockList)     /* if nothing is waiting for the mutex */
      mutex->locked = 0;            /* the mutex is now unlocked       */
    else {                          /* else if something is pending    */
      mutex->lockList->state=Ready; /* we can unlock a locked task.*/
      mutex->owner=mutex->lockList; /*   It is the new mutex owner.*/
                /* And the next task is now the first waiting task */
      mutex->lockList=mutex->lockList->lockList;
} } } }
```

Question II.6 – 2 points

Protégez les parties critiques du code présenté en question I.5 au regard d'une commutation de contexte automatique (sous interruption TIMER).

Question II.7 – 1 point

Expliquez, à partir d'un exemple, ce qu'est une situation d'inter blocage lorsque différentes tâches utilisent différents `mutex` tel que définis précédemment.

Question II.8 – 3 points

Sur cette base, proposez une fonction C `int deadLock(struct mutex_s *mutex)`. Cette fonction permet de détecter un inter blocage. Elle retourne 1 lorsque le fait de réserver (via `mutex_request()`) le mutex, pointé par `mutex`, engendrerait un inter blocage. Et 0 si la tâche courante peut réserver le mutex « sans risque ».