UNIVERSITÉ BORDEAUX 1 Sciences Technologies

Année 2009-2010
Étape : Master Sciences Technologies (semestre 1)
UE : INF463 (Systèmes d'Exploitation)
Modalités : Épreuve sans document

Date : 17 décembre 2009
Heure : 14h00
Durée : 1h30

Lisez l'intégralité du sujet *avant de commencer à répondre calmement aux questions.*

1 Gestion Mémoire et pagination

Question 1 (échauffement) Détaillez le contenu typique (i.e. les champs de la structure C correspondante) des entrées d'une table des pages. Précisez notamment le rôle des différents bits de contrôle. Pour chacun d'entre eux, expliquez quelle entité les positionne (système ? matériel ? les deux ?), quelle entité les consulte et à quel moment.

Gestion mémoire dans Nachos

On se place dans le cadre du simulateur Nachos. On souhaite implementer un nouvel appel système `Fork` qui a la même sémantique que sous `Unix`, c'est-à-dire qui clone l'espace d'adressage du père pour le processus fils. On ne s'intéressera ici qu'aux aspects ayant trait à la gestion des espaces d'adressage.

Question 2 Pour simplifier le problème, le constructeur de la classe `AddrSpace` possède un paramètre supplémentaire (`bool forking`) qui indique si l'on se trouve (ou non) dans le contexte d'un appel à `Fork()` lors de la création de l'espace d'adressage. Si c'est le cas, le processus père créera typiquement l'espace d'adressage de son fils de cette façon : `space = new AddrSpace (NULL, TRUE);`

Voici pour mémoire le code de la boucle allouant les pages d'un processus en cours de création :

```
AddrSpace::AddrSpace (OpenFile * executable, bool forking)
{
    ...
    numPages = divRoundUp (size, PageSize);
    ...
    for (i = 0; i < numPages; i++) {
      pageTable[i].physicalPage = frameProvider->GetEmptyFrame();
      pageTable[i].valid = TRUE;
      pageTable[i].readOnly = FALSE;
      ...
    }
```

Modifiez ce code de manière à dupliquer l'espace d'adressage du père lorsque `forking == TRUE`. Autrement dit, il s'agit donc de copier le contenu des pages du père une à une vers les pages du fils.

On rappelle que les pages sont stockées dans le tableau `mainMemory`, et que la constante `PageSize` indique la taille des pages (en octets). On rappelle également que c'est le processus père qui exécute ce constructeur, et donc sa table des pages est accessible via `currentThread->space->pageTable` (et sa taille via `currentThread->space->numPages`).

Question 3 Rappelez en quoi consiste le mécanisme appelé «*Copy-on-Write (CoW)*» et à quoi il sert. Lors du déclenchement d'une interruption suite à une tentative d'écriture, comment le noyau peut-il distinguer une situation de *CoW* d'une erreur d'accès imputable au programme ?

Question 4 On souhaite mettre en place stratégie *Copy-on-Write* au sein de Nachos. Les pages physiques vont dorénavant être (potentiellement) partagées entre plusieurs processus, on décide de rajouter un *compteur de référence* pour chaque page physique de la machine, qui indiquera à tout moment le nombre de processus référençant une page.

Voici l'essentiel du code de la classe `FrameProvider`, qui gère les pages physiques :

```
class FrameProvider
{
  public:

    int GetEmptyFrame()
    {
      int frame = bitmap->Find();

      if (frame != -1)
        bzero(mainMemory + ... ); // clear page

      return frame;
    }

    void ReleaseFrame(int frame)
    {
      bitmap->Clear(frame);
    }

    FrameProvider () // Initialization
    {
      bitmap = new BitMap(NumPhysPages);
    }

  private:
    BitMap *bitmap;
};
```

Donnez une version étendue de cette classe permettant d'associer un compteur de référence à chaque page. Ajoutez deux fonctions `IncRefCount(int frame)` et `DecRefCount(int frame)` permettant de manipuler ces compteurs depuis l'extérieur de l'objet `frameProvider`.

Question 5 Donnez la nouvelle version du constructeur de la classe `AddrSpace`, de manière à ce que le père et le fils partagent physiquement les mêmes pages (en lecture seule) au lieu de les copier.

Question 6 Expliquez ce que devra faire le noyau lorsque 3 processus partageant une même page vont réaliser chacun une opération d'écriture vers cette page. Le traitement sera-t-il identique dans les trois cas ?

Question 7 Toujours pour simplifier, on supposera qu'en temps normal les pages des processus sont toujours accessibles en écriture. Donc, lorsqu'une interruption de type *ReadOnlyException* (i.e. tentative d'écriture dans une page protégée) est déclenchée, il s'agit forcément d'une situation liée au mécanisme de *CoW*.

Voici à quel endroit une telle erreur peut-être traitée dans le noyau Nachos :

```
void
ExceptionHandler (ExceptionType which)
{
    if (which == ReadOnlyException) {
      int address = machine->ReadRegister (BadVAddrReg);
      ...
    }
```

Donnez le code du traitement d'interruption suite à un *CoW*, et expliquez-en les grandes lignes.

2 Synchronisation

On souhaite disposer de verrous similaires aux « *Mutex* », mais permettant d'établir facilement une synchronisation de type « lecteurs/rédacteurs » au sein des applications. L'idée est donc de fournir un type `rwlock_t` et des primitives associées (`rwl_readlock()`, `rwl_readunlock()`, etc.) qui permettent à un processus lecteur (resp. rédacteur) d'encadrer la zone de code critique où il accèdera aux données partagées en lecture (resp. écriture).

Donnez le code associé à la gestion des verrous en lecture-écriture, en utilisant des primitives fournissant la sémantique des moniteurs de Hoare. On ne demande pas d'implementer une version équitable du problème.

```
/* code à écrire */
typedef ... rwlock_t ;

void rwl_readlock(rwlock_t *l) ;
void rwl_readunlock(rwlock_t *l) ;
void rwl_writelock(rwlock_t *l) ;
void rwl_writeunlock(rwlock_t *l) ;
```

```
/* code disponible */
typedef ... mutex_t ;
typedef ... cond_t ;
void mutex_lock(mutex_t *m) ;
void mutex_unlock(mutex_t *m) ;
void cond_wait(cond_t *c, mutex_t *m) ;
void cond_signal(cond_t *c) ;
```