université de BORDEAUX

Collège Sciences et technologies

Année	2017-2018	**Période**	Session 1
Master	Informatique		
Code UE	4TIN705U	**Épreuve**	Systèmes d'Exploitation
Date	14/12/2017	**Documents**	Non autorisés
Début	14h30	**Durée**	1h30

1 Questions de cours (échauffement)

Question 1 Les sémaphores et moniteurs de Hoare sont des outils de synchronisation fournis par le noyau. Sur quel mécanisme de plus bas niveau leur implémentation s'appuie-t-elle ? Pouquoi une implémentation en espace utilisateur serait moins efficace de manière générale ?

Question 2 Rappelez le principe général de la « pagination sur disque ». À quoi cela sert-il ? Le matériel (en particulier le circuit MMU du processeur) doit-il offrir un support spécifique pour mieux décider de la répartition des pages en mémoire vive et sur disque ? Est-ce le matériel ou le noyau qui doit connaître l'organisation des pages sur disque ?

2 Pagination sur disque

On se place dans le cadre du simulateur NACHOS dans lequel on souhaite implanter un mécanisme de *swap* des pages sur disque. Pour simplifier, on considère qu'un bloc disque a la même taille qu'une page mémoire (i.e. `PageSize`). On dispose d'un objet global `swap` (instance de la classe `Swap`) qui permet de lire/écrire des blocs depuis/sur le disque (les blocs sont numérotés de 1 à `numBlocs`[1]).

```
class Swap
{
  ...
  void ReadBloc(unsigned numBloc, void *destBuffer);
  void WriteBloc(unsigned numBloc, void *srcBuffer);
};
```

À titre d'illustration, voici comment copier le contenu du bloc disque n°5 vers la page physique n°3 :

```
swap->ReadBloc(5, machine->mainMemory + 3 * PageSize);
```

Pour gérer l'espace de swap en permettant au noyau d'allouer et libérer des blocs, on déclare une nouvelle variable globale « `blocProvider` » dont l'initialisation est effectuée de la manière suivante :

```
blocProvider = new PageProvider(numBlocs);
```

Le comportement est donc similaire à pageProvider, à la différence qu'il gère le disque plutôt que la mémoire.

Question 1 On suppose que le champ `physicalPage` de la table des pages des processus est codé sur suffisamment de bits pour contenir un numéro de bloc disque. On peut donc utiliser ce champ pour mémoriser l'emplacement d'une page virtuelle sur le disque lorsque qu'elle a été évincée de la mémoire physique.

Proposez une convention simple permettant à NACHOS de distinguer une page invalide d'une page « swappée » sur le disque (dans les deux cas, le bit `valid` est positionné à `FALSE`).

Question 2 Écrivez une fonction `int SwapOut(AddrSpace *space, unsigned numVirtPage)` qui sera appelée par le noyau pour évacuer sur le disque la page virtuelle n°`numVirtPage` du processus `space`. `SwapOut` doit simplement transférer le contenu de la page vers le disque et modifier la table des pages pour refléter cette nouvelle situation. On rappelle que la table des pages du processus « victime » est accessible via `space->pageTable`. L'entier renvoyé doit indiquer si l'opération a réussi ou non. On ne se préoccupera pas des problèmes de synchronisation.

NB : la page physique dont on aura recopié le contenu sur le disque ne doit pas être restituée au système dans cette fonction.

1. Le numéro 0 n'est donc pas utilisé.

Question 3 On dispose d'une fonction `FindVictim` prédéfinie (que vous n'avez donc pas à l'écrire) capable de trouver la page virtuelle la « moins récemment utilisée » parmi toutes. Voici le profil de la primitive, qui renvoie un pointeur vers l'espace d'adressage victime ainsi que le numéro de la page virtuelle choisie dans cet espace :

```
void FindVictim(AddrSpace **space, unsigned *numVirtPage);
```

Donnez une nouvelle version de la fonction `GetEmptyPage` (dont le code d'origine est rappelé ci-dessous) qui utilise `FindVictim` et `SwapOut` pour évincer une page et récupérer son emplacement lorsqu'il n'y a plus aucune page disponible en mémoire physique.

```
class PageProvider
{
    int GetEmptyPage()
    {
      int page = bitmap->Find();

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

      return page;
    }
```

Question 4 On veut maintenant permettre aux processus de récupérer leurs pages lorsqu'ils en ont besoin. Cela nécessite de traiter correctement les interruptions déclenchées lorsque la MMU rencontre une page invalide.

Lorsqu'une interruption de type « erreur de protection » se produit, l'exécution bascule dans le noyau NACHOS dans la fonction `ExceptionHandler` :

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

Écrivez le code à l'intérieur du `if` pour traiter correctement le rapatriement d'une page depuis le disque lorsque c'est nécessaire, ou pour exécuter `interrupt->Halt()` lorsqu'il s'agit véritablement d'un accès mémoire illégal.

Question 5 Sans synchronisation, de nombreux problèmes peuvent survenir. Par exemple, un processus peut demander l'accès à une page qui est justement en cours de transfert vers le disque...

Donnez une nouvelle version de `SwapOut` et de `ExceptionHandler` corrigeant ce problème. Indiquez bien les données annexes dont vous avez besoin. Vous pouvez utiliser au choix les sémaphores ou les moniteurs.

3 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 sémaphores. 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) ;
```