université de BORDEAUX

Collège Sciences et technologies

Année	2017-18	**Période**	Session 2
Master	Informatique		
Code UE	INAW11EX	**Épreuve**	Systèmes d'Exploitation
Date	25/06/2018	**Documents**	Non autorisés
Début	9h	**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 Gestion Mémoire

Question 1 Rappelez en quoi consiste le mécanisme appelé «*Allocation paresseuse des pages*» dans un système d'exploitation. À quoi sert-il ? Expliquez de quel type de structures de données le système a besoin pour mettre en place un tel mécanisme. Comment le système sait-il qu'il a affaire à une *allocation différée* lorsqu'il traite une interruption de type « *page invalide* » (i.e. un défaut de page) ?

Pour les questions suivantes, on se place maintenant dans le cadre du simulateur Nachos dans lequel on souhaite implanter un mécanisme d'allocation paresseuse des pages. L'idée est d'utiliser ce mécanisme pour allouer les pages constituant les piles des threads.

Question 2 Pour mémoriser que l'allocation d'une page de processus a été demandée, on se propose de rajouter un tableau à l'intérieur de la classe `AddrSpace` (appelons-le `lazyAlloc`).

Ajoutez la déclaration de ce tableau dans le code ci-dessous. À quel moment de l'exécution connaîtra-t-on vraiment la taille de ce tableau pour l'espace d'adressage courant ?

```
class AddrSpace
{
  private:
    TranslationEntry * pageTable; // Assume linear page table translation
    unsigned int numPages;         // Number of pages in the virtual address space
    ... // déclaration de lazyAlloc à définir ici
};
```

Question 3 On suppose que, lors de la création d'un thread au sein d'un processus, les pages nécessaires à l'allocation de sa pile sont allouées ainsi :

```
for (i = thread_stack_bottom; i < thread_stack_bottom + nb_stack_pages; i++) {
  pageTable[i].physicalPage = frameprovider->GetEmptyFrame();
  pageTable[i].valid = TRUE;
  ...
}
```

Modifiez ce code pour simplement « noter que la page aurait dû être allouée » au lieu de l'allouer directement. Que faut-il modifier d'autre pour que le système Nachos reprenne la main dès que le processus tente d'utiliser une telle page ? Donnez le code complet.

Question 4 Lorsqu'une interruption de type « erreur de protection » se produit, l'exécution bascule en mode noyau 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 l'allocation paresseuse d'une page lorsque c'est nécessaire. Dans les autres cas, on pourra simplement exécuter `interrupt->Halt()`.

Question 5 On pourraît souhaiter appliquer le principe de l'allocation paresseuse à toutes les pages d'un processus, pas seulement aux piles des threads qu'il crée. Dans le cas des pages contenant le code du processus et ses données initialisées, c'est toutefois plus compliqué. Pourquoi ?

Expliquez ce qu'il faudrait faire pour allouer ces pages de manière paresseuse (on ne demande pas de code) ? Que pensez-vous des performances d'une telle solution ?

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) ;
```