université de BORDEAUX

Collège Sciences et technologies

Année	2019-2020	Type	Examen
Master	Informatique		
Code UE	4TIN705U	Épreuve	Systèmes d'Exploitation
Date	13/12/2019	Documents	Non autorisés
Début	14h30	Durée	1h30

La plupart des questions peuvent être traitées même si vous n'avez pas répondu aux précédentes. *À la fin du sujet, vous trouverez un memento vous rappelant la syntaxe de quelques fonctions utiles.*

1 Question de cours (échauffement)

Les systèmes d'exploitation modernes utilisent presque tous des mécanismes de pagination s'appuyant sur des tables de pages à plusieurs niveaux. Faites un joli dessin d'une table des pages à 3 niveaux. Mentionnez un avantage à utiliser une table à 4 niveaux plutôt qu'une table à 3 niveaux seulement. Mentionnez un inconvénient.

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 `numBlocks` [1]) au moyen des primitives suivantes :

```
void Swap::ReadBlock (unsigned numBlock, void *destBuffer);
void Swap::WriteBlock (unsigned numBlock, void *srcBuffer);
```

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

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

Pour gérer l'espace de swap en permettant au noyau d'allouer et libérer des blocs, on utilise une variable globale `blockProvider` qui encapsule un *bitmap* : `blockProvider = new PageProvider (numBlocks);` Son utilisation est donc similaire à `pageProvider`, à la différence qu'il gère le disque plutôt que la mémoire.

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, ou y mettre 0 lorsque la page est réellement invalide.

Question 1 Écrivez une fonction `int SwapOut(AddrSpace *space, unsigned numVirtPage);` qui sera appelée par le noyau pour évacuer sur disque la page virtuelle n°`numVirtPage` du processus `space`. `SwapOut` doit simplement transférer le contenu de la page sur disque et modifier la table des pages en conséquence, sans restituer la page au système. On rappelle que la table des pages du processus « victime » est accessible via `space->pageTable`. La fonction renvoie -1 si l'opération a échoué. On ne se préoccupera pas des problèmes de synchronisation.

Question 2 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` (voir 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.

```
int PageProvider::GetEmptyPage()
{
  int page = bitmap->Find();
  if (page != -1)
    bzero(machine->mainMemory + ... ); // clear page
  return page;
}
```

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

Question 3 Les processus doivent 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 telle interruption 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.

3 Salon de coiffure

On souhaite simuler le fonctionnement d'un salon de coiffure en modélisant le comportement des clients au moyen de *threads* : leur nombre est aléatoire, tout comme le moment où chacun d'eux se décide à se rendre au salon de coiffure. Chaque thread exécute la fonction `client` (décrite ci-après) puis se termine.

```
int places = MAX_PLACES; // places assises dans la salle d'attente
int coiffeur_libre = 1;

void client ()
{
  if (places == 0)
    return; // trop de monde dans la salle d'attente

  places--;

  while (coiffeur_libre == 0)
    /* on s'assoit dans la salle d'attente */ ;

  coiffeur_libre = 0;
  places++;

  sleep (SE_FAIRE_COIFFER);

  coiffeur_libre = 1;
}
```

Question 1 Corrigez le code en introduisant des moniteurs/conditions (et sans doute d'autres variables) partagés. Profitez de l'occasion pour éviter l'utilisation de boucles d'attente active. Lorsque la salle d'attente est pleine, on souhaite que le client puisse faire demi-tour *sans délai*.

Question 2 Le salon de coiffure dispose désormais de `NB_PROFESSIONNELS` coiffeuses/coiffeurs, chacun pouvant donc s'occuper d'un client en parallèle. Indiquez les modifications à apporter à votre code pour implémenter cette nouvelle fonctionnalité.

Question 3 On souhaite maintenant que les clients ne se doublent pas dans la salle d'attente, c'est-à-dire qu'il se fassent coiffer dans l'ordre de leur arrivée dans le salon. Une idée est d'utiliser des tickets « comme à la boucherie » qui permettent à chaque client de récupérer un numéro unique, ainsi que d'utiliser un afficheur indiquant le numéro du prochain client autorisé à aller se faire coiffer. Donnez la nouvelle version du code. Expliquez le rôle de chaque variable introduite.

Memento

```
typedef ... mutex_t;
void mutex_lock(mutex_t *m);
void mutex_unlock(mutex_t *m);
```

```
typedef ... cond_t;
void cond_wait(cond_t *c, mutex_t *m);
void cond_signal(cond_t *c);
void cond_bcast(cond_t *c);
```