université de BORDEAUX

Collège Sciences et technologies

Année	2020-2021	**Type**	Examen
Master	Informatique		
Code UE	4TIN705U	**Épreuve**	Systèmes d'Exploitation
Date	11/12/2020	**Documents**	Non autorisés
Début	11h30	**Durée**	1h30

1 Question de cours (échauffement)

Les systèmes d'exploitation modernes utilisent la pagination en s'appuyant sur des tables de pages à plusieurs niveaux. Faites un joli dessin d'une table des pages à 3 niveaux, en détaillant le mécanisme de conversion d'adresses. Mentionnez le principal avantage d'un passage à une table à 4 niveaux (plutôt que 3). Y a-t-il une contrepartie ?

2 Copy-on-Write

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 crée un fils en clonant 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 1 On ajoute un paramètre supplémentaire « `forking` » au constructeur de la classe `AddrSpace` indiquant si l'on se trouve dans le contexte d'un appel à `Fork` lors de la création d'un espace d'adressage. Voici le code de la boucle allouant les pages d'un processus en cours de création :

```
AddrSpace::AddrSpace (OpenFile * executable, bool forking)
{
    ...
  for (i = 0; i < numPages; i++) {
    pageTable[i].physicalPage = pageProvider->GetEmptyPage();
    pageTable[i].valid = TRUE;
    pageTable[i].readOnly = FALSE;
    ...
```

Modifiez ce code de manière à dupliquer l'espace d'adressage du père lorsque `forking == TRUE`, c'est-à-dire copier le contenu des pages du père une à une vers les pages du fils.

On rappelle que la mémoire physique est contenue dans le tableau `machine->mainMemory`, et que la constante `PageSize` indique la taille des pages (en octets). On rappelle également que la table des pages du processus "père" est accessible via `currentThread->space->pageTable` (et sa taille via `currentThread->space->numPages`).

Question 2 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 3 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 `PageProvider`, qui gère les pages physiques :

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

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

      return page;
    }

    void ReleasePage(int page)
    {
      bitmap->Clear(page);
    }

    PageProvider () // Initialization
    {
      bitmap = new BitMap(NumPhysPages);
    }
  private:
    BitMap *bitmap;
};
```

Étendez cette classe de façon à associer un compteur de référence à chaque page. Ajoutez deux fonctions permettant de manipuler ces compteurs depuis l'extérieur de l'objet : `IncRefCount(int page)` et `DecRefCount(int page)`.

Question 4 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 5 On supposera qu'en temps normal les pages des processus sont toujours accessibles en écriture. Donc, lorsqu'une interruption de type *ReadOnlyException* est déclenchée, il s'agit forcément d'une situation liée au mécanisme de *CoW*. Expliquez brièvement les différentes étapes du traitement de cette interruption dans le noyau. Voici à quel endroit elle doit être traitée dans le noyau Nachos :

```
void ExceptionHandler (ExceptionType which)
{
    if (which == ReadOnlyException) {
      int VirtAddress = machine->ReadRegister (BadVAddrReg);
      int VirtPage = VirtAddress / PageSize;
      ... // À compléter
    }
```

Donnez le code du traitement d'interruption suite à un *CoW*. On rappelle que la table des pages du processus en cours peut-être retrouvée au moyen de `currentThread->space->pageTable`.

3 Salon de coiffure

On souhaite simuler le fonctionnement d'un salon de coiffure « *sans rendez-vous* » 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 fasse 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, c'est-à-dire qu'ils 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 des variables introduites.

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);
```