université de BORDEAUX

Collège Sciences et technologies

Année	2022-2023	Type	Examen
Master	Informatique		
Code UE	4TIN705U	Épreuve	Systèmes d'Exploitation
Date	09/12/2022	Documents	Non autorisés
Début	11h30	Durée	1h30

1 Questions de cours (échauffement)

Question 1 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 ?

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 ?

2 Brazil

Tout au long de cet exercice, on utilisera les **moniteurs de Hoare** pour résoudre les problèmes de synchronisation ((les types et fonctions associées sont rappelés en fin de sujet).

On souhaite simuler le fonctionnement de l'accueil du public dans un service administratif, où les personnes entrent, prennent un ticket numéroté auprès d'un distributeur, puis patientent en salle d'attente jusqu'à ce que leur numéro s'affiche au-dessus d'un guichet, avant de s'y rendre pour obtenir le document qu'elles sont venues chercher.

Ces personnes sont modélisées au moyen de *threads*. Leur nombre est aléatoire, tout comme le moment où ils sont lancés et exécutent la fonction `entrer_prefecture`.

```
1  int prendre_ticket (void) { ... }
2
3  void entrer_prefecture ()
4  {
5    int num_ticket, num_guichet;
6
7    num_ticket = prendre_ticket ();
8
9    num_guichet = attendre (num_ticket);
10
11   retirer_document (num_guichet);
12 }
```

Question 1 Cette question s'intéresse uniquement à la fonction `prendre_ticket`. Donnez une implémentation de cette fonction qui renvoit successivement 0, 1, 2, ... et qui garantit que chaque personne récupère un numéro unique. N'oubliez pas d'indiquer les variables globales dont vous avez besoin.

Question 2 On suppose que l'administration n'a ouvert qu'un unique guichet (la fonction `attendre` renverra toujours le numéro de guichet 0), et on s'intéresse maintenant aux deux fonctions `attendre` et `retirer_document`. L'idée est que les personnes passent en respectant leur numéro de ticket : elles scrutent un afficheur et, lorsque le numéro correspond à leur ticket, elles se rendent au guichet.

Voici une implémentation préliminaire de ces fonctions.

```
1 int afficheur = 0; // Indique le numéro du ticket admis au guichet
2
3 int attendre (int ticket)
4 {
5   while (ticket != afficheur) /* rien */ ;
6
7   return 0;
8 }
9
10 void retirer_document (int guichet)
11 {
12   sleep (RETRAIT_DOCUMENT);
13   afficheur++;
14 }
```

Indiquez quels sont les problèmes posés par cette implémentation. Corrigez-les en ajoutant de la synchronisation (mutexes et conditions) dans ces deux fonctions.

Question 3 Mise sous pression par une affluence importante des personnes, l'administration décide d'ouvrir cinq guichets, et donc cinq afficheurs (un par guichet). Initialement, les personnes munies d'un ticket entre 0 et 4 peuvent donc se rendre au guichet sans attendre.

```
1 #define NB_GUICHETS 5
2 int afficheur[NB_GUICHET] = { 0, 1, 2 , 3, 4 };
```

Lorsqu'une personne récupère un ticket, elle doit désormais attendre que l'un des afficheurs corresponde à son numéro de ticket. On peut imaginer qu'un petit « bip » est émis à chaque fois qu'un afficheur est réactualisé, ce qui permet aux personnes de se réveiller uniquement lorsqu'elles entendent le bip...

Par ailleurs, à la fin de `retirer_document`, il faut évidemment veiller à ce que le nouveau numéro affiché au-dessus du guichet soit différent des deux autres.

Donnez le nouveau code des fonctions `attendre` et `retirer_document` en prenant bien soin de proposer une solution simple mais qui permette à plusieurs personnes de retirer leur document en parallèle.

3 Gestion mémoire

On souhaite implementer un appel système `Fork` dans Nachos qui crée un fils en clonant l'espace d'adressage du père pour le processus fils.

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 courant (ou "père") est accessible via `currentThread->space->pageTable` (et sa taille via `currentThread->space->numPages`).

Question 2 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 3 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 4 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`.

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);
```