université de BORDEAUX

Collège Sciences et technologies

Année	2015-2016	Période	Session 1
Master	Informatique		
Code UE	INAW11EX	Épreuve	Systèmes d'Exploitation
Date	17/12/2014	Documents	Non autorisés
Début	14h00	Durée	1h30

1 Question de cours

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 ? Dans quelle mesure le matériel doit-il connaître l'organisation des pages sur disque ?

2 Synchronisation

On s'intéresse aux entrées-sorties au sein d'un système d'exploitation, et plus précisément à l'affichage de caractères sur un teminal. Le code suivant illustre comment on a construit une fonction `put_char` à partir d'une primitive fonctionnant de manière asynchrone (dans cette version asynchrone, chaque appel à `async_put_char` est suivi d'une interruption, et il n'est pas possible d'appeler une nouvelle fois `async_put_char` dans l'intervalle de temps).

```
Semaphore mutex(1);
Semaphore write_done(0);

void terminal_interrupt_handler()
{
  V(&write_done);
}

void put_char(char c)
{
  P(&mutex);
  async_put_char(c);
  P(&write_done);
  V(&mutex);
}
```

Question On souhaite étendre le code précédent afin d'introduire un tampon (de taille `MAX` caractères) entre l'application et le terminal. L'idée est que la fonction `put_char` dépose le caractère dans le tampon sans se bloquer (sauf lorsque le tampon est plein), et que d'un autre côté un thread s'exécutant en tâche de fond récupère les caractères un par un (en ordre FIFO) pour les afficher à l'aide d'`async_put_char`. On supposera que ce thread exécute une boucle infinie depuis laquelle il appelle une fonction `terminal_daemon` qui se charge de traiter un caractère (ou qui se bloque si le tampon est vide).

Donnez le code de la fonction `terminal_daemon` et le nouveau code de la fonction `put_char`. Vous pouvez bien sûr définir de nouveaux sémaphores.

3 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 clone 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 Pour simplifier le problème, le constructeur de la classe `AddrSpace` possède un paramètre supplémentaire (`bool forking`) qui indique si l'on se trouve (ou non) dans le contexte d'un appel à `Fork()` lors de la création de l'espace d'adressage. Si c'est le cas, le processus père créera typiquement l'espace d'adressage de son fils de cette façon : `space = new AddrSpace (NULL, TRUE);` 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 = frameProvider->GetEmptyFrame();
      pageTable[i].valid = TRUE;
      pageTable[i].readOnly = FALSE;
      ...
    }
```

Modifiez ce code de manière à dupliquer l'espace d'adressage du père lorsque `forking == TRUE`. Autrement dit, il s'agit donc de copier le contenu des pages du père une à une vers les pages du fils.

On rappelle que les pages sont stockées dans le tableau `mainMemory`, et que la constante `PageSize` indique la taille des pages (en octets). On rappelle également que c'est le processus père qui exécute ce constructeur, et donc sa table des pages 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 `FrameProvider`, qui gère les pages physiques :

```
class FrameProvider
{
  public:
    int GetEmptyFrame()
    {
      int frame = bitmap->Find();

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

      return frame;
    }

    void ReleaseFrame(int frame)
    {
      bitmap->Clear(frame);
    }

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

Donnez une version étendue de cette classe permettant d'associer un compteur de référence à chaque page. Ajoutez deux fonctions `IncRefCount(int frame)` et `DecRefCount(int frame)` permettant de manipuler ces compteurs depuis l'extérieur de l'objet `frameProvider`.

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`.