Pale INF422 – Composants d'un système informatique
INF422 Final Exam – Components of a Computing System

2008–2009

- L'examen dure 3 heures.
  The exam lasts 3 hours.
- Tous documents autorisés.
  All documents are authorized.
- Il est impératif de commenter les programmes et de justifier vos réponses.
  Programs must be commented and every answer must be justified.

Sujet en français

On étudie la lecture de fichiers stockés sur un disque optique de type CDROM.

On modélise un CDROM (simplifié) comme une longue *piste* hélicoïdale (en spirale) constituée d'une suite de N "trous" représentant chacun à un bit de données. L'accès proprement dit s'effectue par l'intermédiaire d'une unique tête de lecture capable de se déplacer *radialement*. La piste s'enroule C fois de l'extérieur vers l'intérieur, se rapprochant du centre du disque d'une distance de P micron à chaque tour (P pour *track Pitch*). On suppose que le disque est en rotation à une vitesse constante de T tours par minute, et que le déplacement radial de la tête de lecture s'effectue à la vitesse continue de P micron par microseconde. Notez que la tête se déplace progressivement lors de la lecture de bits consécutifs : on suppose que sa vitesse maximale est largement supérieure à celle permettant le suivi de la piste hélicoïdale lors de la rotation du disque. Voici des valeurs réalistes pour un CDROM classique :

- $N = 5.6 \times 10^9$ bits soit 7×10^8 octets ;
- $P = 2\,\mu m$;
- $C = 20000$, ainsi chaque tour de disque couvre 280000 bits en moyenne ;
- et $T = 300\,mn^{-1}$.

Ces valeurs numériques ne sont indiquées que dans un but d'illustration, et on se contentera par la suite de calculs symboliques (paramétrés par N, P, C et T).

Le *contrôleur* du lecteur de disque est un microprocesseur embarqué sur le périphérique lui même. Il gère le positionnement radial de la tête, l'attente du passage d'une position précise sous celle-ci, et peut procéder à la lecture d'un bloc d'octets consécutifs.

Le système d'exploitation gère le lecteur de disque à l'aide d'un pilote dédié à ce périphérique. Ce pilote s'exécute en mode privilégié (noyau). Lorsqu'un processus utilisateur effectue une lecture sur un fichier ouvert résidant sur le CDROM, celui-ci utilise l'appel système `read()`, lequel délègue l'opération au pilote du périphérique.

1 Scénario d'utilisation

On suppose que le périphérique est connu du système d'exploitation par le *numéro majeur* 11, lequel identifie le fichier de périphérique `/dev/cdrom`.

Question 1.1

La commande `ls -l /dev/cdrom` affiche :

```
brw-rw---- 1 root cdrom 11, 0 2008-11-07 09:04 /dev/cdrom
```

Expliquez intégralement ce message.

Question 1.2

On suppose qu'un CDROM a été inséré dans le lecteur, et que celui ci a été gravé avec un système de fichiers `iso9660` (standard pour les CDROM) contenant un unique fichier appelé `record`.

On exécute les commandes UNIX suivantes en tant que `root`, en supposant le répertoire `media` initialement vide :

```
mkdir /media/cdrom
ls /media/cdrom
mount -t iso9660 /dev/cdrom /media/cdrom
ls /media/cdrom
```

Qu'affiche cette séquence de commandes ? Quel est le rôle de la commande `mount` dans cette séquence ?

On suppose désormais que cette séquence a été exécutée.

Question 1.3

Par défaut, la commande `mount` utilisée en tant que `root` donne des droits d'accès restreints. Quelle commande utiliser (en tant que `root`) pour autoriser la lecture du fichier `record` à tous les utilisateurs ?

Observez les droits d'accès au fichier `/dev/cdrom` et ceux du fichier `record`. Y a-t-il un problème ? Si oui, lequel ? Si non, pourquoi ?

Question 1.4

On supposera par la suite que toutes les applications Java sont exécutées par la machine virtuelle Java avec les droits d'un simple utilisateur. Donnez deux raisons pour lesquelles il n'est pas souhaitable d'exécuter une application usuelle en tant que `root`.

Question 1.5

À quoi sert la méthode Java suivante ?

```
static byte[] whatisit(String name) {
    byte[] buffer;
    try {
        File file = new File(name);
        int size = file.length();
        buffer = new byte[size];
        FileInputStream in = new FileInputStream(file);
        while (size > 0) {
            /* Essaye de lire 'size' octets, en les rangeant
               à partir de l'élément 'position' du tableau 'buffer'
               et retourne le nombre d'octets effectivement lus */
            n = in.read(buffer, position, size);
            position = position + n;
            size = size - n;
        }
        in.close();
    } catch (Exception e) {
        System.exit(1);
    }
    return buffer;
}
```

À l'aide de cette méthode, écrivez une méthode `main` qui lit le fichier `record` du CDROM.

Question 1.6

Lors de l'exécution de `in = new FileInputStream(file)`, le fichier `record` est ouvert et un objet `in` dédié à la lecture de ce fichier est créé. Quels sont les rôles respectifs de la machine virtuelle Java et du système d'exploitation dans cette opération ?

Question 1.7

Décrivez les grandes étapes par lesquelles le noyau du système découvre que le périphérique en charge du fichier `record` est le lecteur de CDROM.

Question 1.8

Citez deux avantages du montage du CDROM dans le répertoire `/media/cdrom`, par opposition à un accès direct au fichier `/dev/cdrom` par les applications.

Question 1.9

Lors d'un appel à `in.read()`, la machine virtuelle Java effectue un appel système `read()` sur le descripteur de fichier attribué à l'ouverture du fichier `record`. Cet appel système transmet au noyau le nombre d'octets à lire ainsi que l'adresse mémoire à partir de laquelle ces octets doivent être stockés. Le descripteur du fichier permet au noyau d'identifier les blocs de données contenant les octets à lire ; indiquez les étapes principales et structures de données impliquées dans cette identification.

2 Lectures non bloquantes

On supposera par la suite que les blocs de données du système de fichiers du CDROM comportent 1024 octets.

Question 2.1

Pour implémenter l'appel système `read()`, le noyau appelle itérativement une fonction standard du pilote de périphérique, dédiée à la lecture d'un bloc de données unique. Cette fonction, appelée `read_block()`, reçoit le numéro d'un bloc du système de fichier et l'adresse d'une zone mémoire où stocker les données de ce bloc. Elle les transmet sous forme de *requête* standard sur un bus de communication du système, à destination du contrôleur du lecteur de disque optique.

Pourquoi n'est il pas souhaitable que les utilisateurs aient un accès direct à une telle fonction ? Indiquez trois raisons importantes.

Question 2.2

En vous plaçant dans une configuration la plus défavorable possible (position du bloc, de la tête de lecture et angle de rotation du disque), calculez le temps de latence maximal entre l'appel à `read_block()` et la terminaison complète de la lecture du bloc (on négligera le temps d'exécution dans le pilote, y compris le temps d'écriture du bloc en mémoire de l'ordinateur).

Question 2.3

Pendant le traitement de la requête, on ne souhaite pas que la fonction `read_block()` soit bloquée en attente, mais au contraire que le pilote soit en mesure de préparer la requête suivante ou bien de rendre la main au noyau du système pour d'autres tâches (y compris l'exécution d'autres applications).

Le contrôleur du lecteur de disque optique informe de la fin du traitement d'une requête en envoyant un signal au processeur, interprété par ce dernier par une *interruption*. Que doit faire le pilote du périphérique pour préparer la réception d'une telle interruption ? Que doit il faire lors de la réception de l'interruption proprement dite ?

Question 2.4

Il est important de ne pas déclencher de longues opérations en réponse à une interruption, sous peine de diminuer drastiquement la réactivité du système. Une manière moderne de traiter ce problème consiste à ne pas commander directement le contrôleur du lecteur de disque depuis le pilote de périphérique, mais à transmettre les requêtes à un processus dédié, appelé `kcdromd`, s'exécutant également en mode privilégié. Le processus `kcdromd` n'interagit qu'avec le pilote associé, et commande directement le contrôleur de disque optique.

La transmission des requêtes se fait par le biais d'une zone mémoire dédiée, implémentant une structure de file d'attente. Pour implémenter un tel mécanisme, on dispose de l'appel système `pause()` qui suspend l'exécution d'un processus jusqu'à l'arrivée d'un signal (appel système `kill()`). Définissez et décrivez un protocole permettant :

- au processus `kcdromd` de suspendre son exécution lorsque la file de requêtes est vide ;
- au pilote de réveiller ce processus lors de l'enfilement de nouvelles requêtes ;
- au pilote de ne pas attendre la terminaison du traitement des requêtes.

Question 2.5

Dans cette question, on utilise le langage Java pour simuler le protocole précédent.[1]

On stocke la file d'attente dans un tableau d'objets d'une classe `Request` (à définir), que l'on supposera infini pour simplifier.

[1] Une aberration pour du code s'exécutant en mode privilégié, mais il ne s'agit ici que d'une illustration.

On fait comme si le pilote exécutant la fonction `read_block()` et le processus `kcdromd` étaient de simples *threads* concurrents, et on ignore pour l'instant les problèmes de *course critique* entre les accès au tableau de requêtes.

Implémentez la classe `Request`, ainsi qu'une classe `FIFO` munie de fonctions statiques `enqueue()` et `dequeue()` (et de variables statiques auxilliaires) telles que le pilote puisse enfiler des requêtes en appelant `FIFO.enqueue()`, que processus `kcdromd` traite les requêtes dans leur ordre d'arrivée en appelant `FIFO.dequeue()`, et qu'il se mette en attente lorsque la file est vide. Vous supposerez qu'il existe des fonctions `pause()` et `kill()` effectuant les appels système du même nom.

Question 2.6

Des courses critiques entre les threads peuvent conduire à des états inconsistants. Identifiez une course critique dans votre programme impliquant un enfilement concurrent avec un défilement. Montrez le en décrivant un entrelacements fautifs de l'exécution d'un appel à `read_block()` et du processus `kcdromd`.

Question 2.7

Selon certaines conditions, une course critique peut aussi exister entre l'enfilement de plusieurs requêtes. Quelles sont ces conditions ? Quelle hypothèse faut il faire sur l'exécution des appels système par le noyau ?

Question 2.8

Comment remédier à ces courses critiques en Java ? Indiquez quelles modifications doivent être apportées à votre programme (sans le réécrire).

3 Optimisation du traitement des requêtes

Étant donné la lenteur des déplacements de la tête de lecture et de la rotation du disque par rapport à la lecture proprement dite, on utilise souvent une heuristique appelée SCAN inspirée du *principe de l'ascenseur*. L'heuristique SCAN permet d'optimiser le temps de traitement d'une *liste de requêtes en attente*. La possibilité d'initier plusieurs requêtes sans attendre leur terminaison est donc essentielle.

Question 3.1

Le principe est le suivant : la tête de lecture débute à la position la plus externe du disque, puis le processus `kcdromd` défile les requêtes en attente en ordre croissant du numéro de bloc. Lorsque le bloc le plus interne de la liste courante est atteint, le mouvement repart en sens inverse et on ordonnance les appels en ordre décroissant du numéro de bloc ; et ainsi de suite.

Le gain obtenu via cette heuristique dépend bien entendu de l'ordre d'émission des requêtes lors des appels à `read_block()`. Dépend-t-il également de la distribution des numéros de blocs accédés ? Effectuez une analyse quantitative dans le cas d'une séquence de lectures de blocs uniformément distribués, ou bien tassés vers l'une des extrémités. Vous pourrez construire une séquence type correspondant à chaque cas au lieu d'effectuer une analyse statistique dans le cas général.

Question 3.2

Montrez que l'heuristique ne prend pas parfaitement en compte les caractéristiques temporelles du lecteur de CDROM (elle dérive en réalité d'une heuristique définie pour les disques durs). Décrivez informellement une amélioration préservant le principe de l'ascenseur, mais fondée sur un ordre différent.

Question 3.3

Montrez que l'heuristique SCAN est équitable, c'est-à-dire que toute requête en attente finira par être traitée au bout d'un laps de temps fini.

Question 3.4

Construisez une séquence entrelaçant traitement de requêtes et appels à `read_block()` telle que le temps d'attente d'une lecture donnée puisse croître bien au delà du temps de traitement des requêtes en attente à tout instant.

Cette propriété de l'heuristique SCAN montre que l'équité théorique n'est pas suffisante pour garantir une réactivité bornée "raisonnable".

Question 3.5

Proposez une variation de l'heuristique SCAN où le temps d'attente maximal pour toute requête ne dépend que du nombre de requêtes dans la file d'attente au moment de l'envoi de celle-ci.