Examen INF552
Principes et programmation des systèmes d'exploitation

2007–2008

- L'examen dure 2 heures.
- Le sujet comporte 4 pages.
- Tous documents autorisés.
- Il est impératif de commenter les programmes et de justifier vos réponses.

On étudie le contrôle des opérations de lecture/écriture sur un disque optique.

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 supérieure ou égale à celle permettant le suivi de la piste hélicoïdale lors de la rotation du disque. Sur un CDROM classique, on peut prendre comme valeurs réalistes :

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

Le *contrôleur* est un processeur embarqué sur le périphérique (lecteur de disque optique) chargé de positionner la tête radialement et d'attendre le passage d'un bit donné sous celle-ci, afin de procéder à sa lecture.

Le système d'exploitation gère ce disque à l'aide d'un pilote de bas niveau s'exécutant en mode privilégié (noyau), et d'un processus `kcdromd` s'exécutant également en mode privilégié, seul à intéragir avec ce pilote.

Le pilote du périphérique commande directement le contrôleur via la fonction suivante.

```
/* Positionne la tête sur l'octet "o" puis lit
   "n" octets consécutifs dans le tableau "t".
   Retourne 0 si tout s'est bien passé, -1 sinon. */
int cd_read(int o, int n, void *t);
```

Les appels à `cd_read()` sont non bloquants (l'exécution du programme se poursuit dès l'ordre transmis au contrôleur). S'agissant de fonctions de bas niveau, on notera que tout appel à `cd_read()` interrompt un éventuel accès en cours. La terminaison de l'accès est notifiée par l'envoi du signal `SIGUSR1` au processus `kcdromd`. Les signaux sont gérés de la même façon pour `kcdromd` que pour un processus s'exécutant en mode utilisateur, bien que `kcdromd` ne fasse pas à proprement parler d'appels système (il s'exécute déjà en mode privilégié).

Lorsqu'un processus utilisateur effectue une lecture sur un fichier ouvert résidant sur le CDROM, l'appel système `read()` correspondant délègue l'opération au processus `kcdromd` en effectuant une "requête" contenant les arguments effectifs de l'appel à `read()`. Pour des raisons de temps, on ne s'occupera pas de la terminaison de l'appel système ni de la communication du nombre d'octets effectivement lus (valeur de retour de l'appel système `read()`).

On utilise un segment de mémoire partagée pour implémenter ces requêtes. On ne se préoccupe pas de la création du segment de mémoire partagée qui a lieu lors du chargement du pilote de CDROM (typiquement au boot du système). On suppose que ce segment est de taille 4 KB et situé à une adresse *physique* contenue dans la variable globale `shm_start` initialisée au démarage du processus `kcdromd`. Rappel : ce processus opère sur des adresses physiques car il s'exécute en mode privilégié.

Le processus `kcdromd` attend indéfiniment l'arrivée de requêtes.

Exercice 1 - Conception du système

On rappelle les prototypes des appels système utilisés dans cet exercice :

```
ssize_t read(int fd, void *buf, size_t count);
```

Sur une architecture 32 bits, chaque requête est associée à un bloc de 16 octets de données, dans lequel sont rangés : fd (4 octets), `buf` (4 octets) et `count` (8 octets).

On utilise les 4 premiers octets du segment de mémoire partagée (adresse `shm_start`) pour gérer un compteur de requêtes, que l'on suppose initialisé à 0, le reste servant à stocker les requêtes en attente de réception par `kcdromd`.

On suppose pour commencer que le noyau du système ne supporte pas l'exécution concurrente de plusieurs appels système.

Question 1.1

On utilise le signal `SIGUSR2` pour informer le processus `kcdromd` de l'arrivée d'une requête dans le segment de mémoire partagée.

Lors d'un appel système `read()`, décrivez (sans programme) les opérations nécessaires à l'envoi d'une requête à `kcdromd`.

Question 1.2

Lorsque le processus `kcdromd` reçoit le signal `SIGUSR2` et qu'il est en attente de requête (ou qu'il vient de terminer le traitement d'une requête en cours), il doit retirer les requêtes du segment de mémoire partagée et les insérer dans une liste chainée qui lui est propre.

En l'absence de section(s) critique(s), tant l'insertion de requêtes que leur retrait peuvent conduire à des états inconsistants. Montrez le en décrivant des entrelacements fautifs de l'exécution d'un appel système `read()` et du processus `kcdromd`.

Question 1.3

Indiquez comment remédier à ces "courses critiques" à l'aide d'un sémaphore, et, en précisant à quelle étape (pour l'appel système comme pour `kcdromd`) l'acquisition et la libération de celui-ci doivent avoir lieu.

Vérifiez que votre solution permet également l'exécution concurrente de plusieurs appels système `read()`, ou modifiez la en conséquence.

Exercice 2 - Première implémentation

Dans cet exercice, on suppose que `kcdromd` effectue les accès au disque (via `cd_read()`) dans l'ordre d'arrivée des requêtes.

Question 2.1

Représentez graphiquement les interactions entre le pilote de bas niveau, le processus `kcdromd` et deux autres processus effectuant chacun un accès à un fichier résidant sur le CDROM.

Question 2.2

On propose d'abord d'utiliser `pause()` pour suspendre l'exécution de `kcdromd` en attente de requêtes ou de la terminaison d'un accès disque en cours. Décrivez la séquence d'appels système et d'appels de handlers de signaux correspondant à la réception d'une requête, de son traitement et de l'attente de sa terminaison par le processus `kcdromd`.

Question 2.3

Quel est l'inconvénient de cette solution ? Décrivez comment l'utilisation de `sigsuspend()` permet d'y remédier.

Question 2.4

Implémentez la boucle infinie du processus `kcdromd`. Pour gagner du temps, utilisez des primitives de manipulation de de file d'attente (`ajouter(noeud)`, `retirer()`, etc.).

Exercice 3 - Optimisation des séquences d'accès

On se place dans une configuration où le processus `kcdromd` est saturé de demandes d'accès au disque : à chaque instant, la liste de requêtes contient de nombreuses demandes en patience.

On suppose pour simplifier que la piste hélicoïdale couvre le même nombre de "trous" à chaque tour de disque.

Question 3.1

Calculez le temps de lecture de n octets, une fois la tête de lecture positionnée devant le premier bit de cette séquence.

Question 3.2

Calculez le temps d'attente entre une configuration où la tête est positionnée devant le bit N et une autre où la tête est positionnée devant le bit N'. Vous pourrez considérer plusieurs cas en fonction du placement relatif de ces bits sur le disque.

Question 3.3

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*. Cette heuristique permet d'optimiser le temps de traitement d'une liste de requêtes en attente. La tête de lecture débute par exemple à la position la plus externe, puis on ordonnance les appels à `cd_read(`o`, `n`, `t`)` en ordre croissant de o. Lorsque l'octet le plus interne de la liste courante est atteint, le mouvement repart en sens inverse et on ordonnance les appels en ordre décroissant de o ; et ainsi de suite.

Le gain obtenu via cette heuristique dépend bien entendu de l'ordre d'émission des requêtes lors des appels système `read()`. Dépend-t-il également de la distribution des accès au disque ? Effectuez une analyse quantitative dans le cas d'une séquence d'accès 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 3.4

Modifiez la boucle infinie du processus `kcdromd` pour mettre en œuvre cette heuristique. Ne vous préoccupez pas de la complexité du tri : préférez un algorithme facile à lire, de type *tri par insertion* (analogue du *tri à bulles* mais plus naturel sur des listes), et utilisez des primitives de file de priorités (`ajouter(pri, noeud)`, `retirer()`, etc.).

Question 3.5

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.

Exercice 4 - Équité et réactivité

Question 4.1

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 4.2

Construisez une séquence entrelaçant envois de requêtes et appels à `cd_read()` 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 4.3

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 liste au moment de l'envoi de celle-ci.

Exercice 5 - Modularisation du système

L'utilisation d'un processus `kcdromd` s'exécutant en mode privilégié comporte de nombreux défauts par rapport au recours à un processus s'éxécutant en mode utilisateur.

Question 5.1

Indiquez 3 de ces défauts.

Question 5.2

On remplace `kcdromd` par un processus `cdromd` s'exécutant en mode utilisateur, avec les droits de `root`, et seul habilité à faire usage d'un appel système `cd_read()` appelant la fonction du pilote de CDROM du même nom.

On suppose que le segment de mémoire partagée est associé au nom `/cdromd_shm` (fichier dans un pseudo file-system sous Linux) lors du chargement du pilote de CDROM.

Programmez la projection de ce segment en mémoire virtuelle du processus `cdromd`.

On utilisera les appels système suivants :

```
int *shm_open(const char *name, int oflag, mode_t mode);
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

Question 5.3

Lorsque le nombre de requêtes en attente croît, le tri de celles-ci peut devenir un facteur limitant pour les performances. On propose donc de découpler le fonctionnement de l'heuristique des appels à `cd_read()`.

Décrivez informellement les modifications à apporter au programme pour autoriser le tri des requêtes à la volée indépendamment des communications avec le pilote de bas niveau, en considérant l'envoi et la réception de signaux et l'usage éventuel de sections critiques supplémentaires.