Pale INF570 – Systèmes d'exploitation

2008–2009

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

Problème

On étudie la lecture et l'écriture de fichiers stockés sur une clé USB.

On modélise la mémoire flash contenue dans la clé USB comme un ensemble de P pages mémoire, chacune comportant S octets. L'accès proprement dit se fait par l'intermédiaire du microprocesseur embarqué sur la clé USB, appelé *contrôleur*. Celui-ci simule le comportement d'un disque dur, dont les données sont découpées en blocs consécutifs de S octets. Pour lire un bloc donné, le contrôleur lit intégralement le bloc de la mémoire flash et copie son contenu dans une mémoire locale, puis transmet ces données à l'ordinateur ayant effectué la requête via le port USB. Pour écrire un bloc donné, le contrôleur récupère le contenu via le port USB, le copie dans sa mémoire locale, puis l'écrit intégralement sur la mémoire flash. On appellera R le temps de lecture d'un bloc et W le temps d'écriture, et T le temps de transmission d'un bloc via le port USB.

Voici des valeurs réalistes pour une clé USB classique :
- $S = 2^{10} = 1024$ octets ;
- $P = 2^{20} = 1048576$, soit un total de 1 giga-octets ;
- $R = 10\,\mu$s ;
- $W = 30\,\mu$s ;
- $T = 10\,\mu$s.

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

Le système d'exploitation gère la clé USB 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 la clé USB, 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* 8, et qu'il possède une unique partition. Lors de la reconnaissance automatique de la clé USB, le système lui a attribué le fichier de périphérique `/dev/sdb1`.

Question 1.1

La commande `ls -l /dev/sdb1` affiche :

```
brw-rw---- 1 root disk 8, 0 2008-11-07 09:04 /dev/sdb1
```

Expliquez intégralement ce message.

Question 1.2

On suppose que l'unique partition a été formatée avec un système de fichiers `ext3` (standard pour Linux) 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/stick
ls /media/stick
mount -t ext3 /dev/sdb1 /media/stick
ls /media/stick
```

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/sdb1` 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 sont exécutées 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`, et citez trois avantages du montage de l'unique partition de la clé USB dans le répertoire `/media/stick`, par opposition à un accès direct au fichier `/dev/sdb1` par les applications.

Question 1.5

Décrivez les grandes étapes (au moins 3) par lesquelles le noyau du système découvre que le périphérique en charge du fichier `record` est la clé USB.

2 Lectures non bloquantes

On suppose par la suite que les blocs de données du système de fichiers comportent *S* octets (identique à la taille des blocs simulés par le contrôleur de la clé USB).

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 port USB et du contrôleur.

Question 2.1

Pourquoi n'est il pas souhaitable que les utilisateurs aient un accès direct à la fonction `read_block()` ?

Question 2.2

Les caractéristiques de cette clé USB sont très différentes de celle d'un disque dur. Malgré ces caractéristiques avantageuses pour la mémoire flash, pourquoi est il intéressant de ne pas attendre la fin d'une requête (en lecture ou en écriture) avant d'en effectuer une deuxième ?

Comparez quantitativement l'approche bloquante (une requête à la fois) et l'approche non bloquante (sans attendre).

Question 2.3

Pendant le traitement de la requête, on ne souhaite donc 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 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 depuis le pilote de périphérique, mais à transmettre les requêtes à un processus dédié, appelé `kusbd`, s'exécutant également en mode privilégié. Le processus `kusbd` n'interagit qu'avec le pilote associé, et commande directement le contrôleur via le port USB.

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 `kusbd` 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

Pour implémenter ce protocole, on stocke la file d'attente dans un tableau `requests` (à définir), que l'on supposera infini pour simplifier. Cela permet d'implémenter la file avec un pointeur de tête et un pointeur de queue, incrémentés lors d'un défilement et d'un enfilement, respectivement.

On suppose que le pilote exécutant la fonction `read_block()` et le processus `kusbd` partagent un segment de mémoire partagé contenant ce tableau, et toutes les autres variables globales auxiliaires dont vous pouvez avoir besoin. De plus, on ignore pour l'instant les problèmes de *course critique* (*race condition*) entre les accès au tableau de requêtes.

Implémentez des fonctions `enqueue()` et `dequeue()`, telles que le pilote puisse enfiler des requêtes en appelant `enqueue()`, que le processus `kusbd` traite les requêtes dans leur ordre d'arrivée en appelant `dequeue()`, et qu'il se mette en attente lorsque la file est vide (vous aurez besoin d'appels système supplémentaires).

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 `kusbd`.

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 ? Indiquez quelles modifications doivent être apportées à votre programme (sans le réécrire).

3 Optimisation de la durée de vie de la mémoire flash

Les mémoires flash actuelles ont un défaut majeur : au bout de quelques millions de cycles d'écriture sur un bloc donné, une fine couche semi-conductrice finit par se détériorer en raison d'effets de transit ioniques : le semi-conducteur perd ses éléments dopants sous l'effet de champs électriques importants utilisés lors de l'écriture. Pour gérer et retarder cette détérioration, le contrôleur de la clé USB implémente trois mécanismes complémentaires :

- lorsqu'un bloc est écrit, il est lu immédiatement après pour vérifier que l'écriture s'est déroulée correctement ; en cas d'erreur, le contrôleur réessaye un certain nombre de fois pour s'assurer qu'il ne s'agit pas d'une erreur intermittente ;
- si l'erreur est bien définitive, le contrôleur marque ce bloc comme "mort" dans une table, elle même stockée en mémoire flash ;
- lors de l'écriture d'un bloc, le contrôleur sélectionne une page mémoire parmi les plus "fraîches" pour contenir ce bloc, évitant ainsi que les écritures répétées d'un même bloc du système de fichiers se traduisent par une disparition rapide de pages de la mémoire flash.

Le troisième mécanisme qui permet de retarder la mort de blocs est appelé *wear levelling*.

Question 3.1

Note : cette question est volontairement ouverte, et doit donner lieu à un effort original de conception, d'étude et de critique du système. Vous pouvez considérer qu'elle occupe une place équivalente à 4 ou 5 questions traditionnelles dans le barème du problème.

Proposez une représentation de la table des blocs morts, permettant au contrôleur d'identifier rapidement un bloc de remplacement. Décrivez comment stocker cette table en mémoire flash ; est il nécessaire de se préoccuper du wear levelling pour cette table ? Proposez une méthode pour implémenter le wear levelling garantissant une distribution (relativement) équitable des écritures au cours du temps, sans perdre de vue la nécessité pour le contrôleur de retrouver rapidement l'emplacement d'un bloc donné. Vous ferez attention aux exigences de votre

méthode en matière de mémoire locale pour exécuter l'algorithme de wear levelling ; le contrôleur est un processeur très simple, disposant de peu de mémoire en pratique (pour des raisons de coût, de place disponible sur la clé USB, et de restrictions sur sa consommation d'énergie).