R. Namyst
LaBRI, Bureau 323

le 2/12/2005

Licence d'informatique — Systèmes d'exploitation
Devoir Surveillé

Durée : 1h20 — Sans document

1 Gestion mémoire (question d'échauffement)

Expliquez brièvement le principe de *segmentation* mémoire utilisé par certains systèmes d'exploitation. De quel support matériel a-t-on besoin pour le mettre en œuvre ?

2 Nachos

Question 1 Dans le devoir 2, vous avez mis en place un appel système `ThreadCreate` permettant la création dynamique de *threads* depuis les programmes utilisateurs. L'espace d'adressage des processus étant une ressource finie, il existe évidemment un nombre maximal de *threads* pouvant s'exécuter simultanément au sein d'un même processus : lorsqu'il n'est plus possible d'allouer la moindre pile, l'appel système `ThreadCreate` échoue...

Dans une telle situation de pénurie de piles, une alternative serait de bloquer le *thread appelant* `ThreadCreate` jusqu'à ce qu'une pile se libère. Discutez de la pertinence d'un tel fonctionnement.

Question 2 Quel est votre sentiment concernant le choix de la taille des piles des *threads* (arbitrairement fixée à 3×128 octets en TP) ? Est-ce faisable/facile d'estimer une taille de pile « standard » ?

Puisqu'il y a toujours le risque de sous-estimer la taille des piles nécessaire, y a-t-il un moyen infaillible de détecter les débordements de pile ? Ou mieux, de les empêcher ? Expliquez.

3 Synchronisation

Dans cet exercice, il s'agit d'utiliser des *sémaphores* pour synchroniser l'implémentation des *tubes* au sein d'un système d'exploitation.

On dispose d'un type `pipe_t` qui peut contenir `MAX_PIPE` octets au maximum, et des opérations `__write_char` et `__read_char` qui permettent respectivement d'insérer ou d'extraire un caractère dans/depuis un tube :

```
typedef struct {
   ???
} pipe_t;

void __write_char(pipe_t *p, char c);

void __read_char(pipe_t *p, char *c);
```

Les primitives `__write_char` et `__read_char` ne contiennent pas de code de synchronisaion : l'exécution de `__write_char` provoque une erreur si le tube et plein, l'exécution de `__read_char` provoque une erreur si le tube est vide, et les appels concurrents à `__write_char` (resp. `__read_char`) ne sont pas supportés. Par contre, `__write_char` et `__read_char` peuvent s'exécuter en parallèle si le tube n'est ni vide ni plein. La structure `pipe_t` contient les champs nécessaires à la gestion interne du tube : vous n'avez pas besoin de connaître ces champs mais vous pouvez bien sûr en ajouter de nouveaux...

Question 1 En utilisant des sémaphores (n'oubliez pas d'indiquer leur valeur initiale), donnez le code des fonctions `write_char` et `read_char`, versions synchronisées respectives des fonctions `__write_char` et `__read_char`.

Question 2 On souhaite maintenant disposer d'une primitive `write` permettant d'écrire plusieurs caractères dans le tube. Voici une première ébauche de cette fonction :

```
int write(pipe_t *p, char *buf, unsigned len)
{
   unsigned i;

   for(i=0; i<len; i++)
      write_char(p, buf[i]);
}
```

On voudrait que l'exécution de la fonction `write` soit atomique, c'est-à-dire que les caractères injectés dans le tube ne se mélangent pas avec d'autres exécutions de `write` (on suppose que la fonction `write_char` n'est plus utilisée directement par les processus).

Donnez le nouveau code de `write`.

Question 3 En pratique, les systèmes ne garantissent l'atomicité des opérations d'écritures dans les tubes que jusqu'à une certaine taille (appelons cette constante `MAX_ATOMIC`). Lorsqu'un appel à `write` porte sur une quantité de données supérieure à `MAX_ATOMIC`, l'opération peut-être vue comme une séquence d'écritures atomiques de `MAX_ATOMIC` octets (sauf pour la dernière séquence qui peut bien sûr porter sur un nombre d'octets inférieur). Entre ces séquences, il est donc possible que s'insèrent des écritures effectuées par d'autres processus...

Donnez une nouvelle version de la fonction `write` fonctionnant selon ce principe.

Question 4 Il est parfois nécessaire de pouvoir privilégier l'exécution de certains processus, comme par exemple dans les systèmes « critiques ». Ainsi, on voudrait disposer d'une fonction `write_char_urgent`, de même profil que `write_char`, qui permettrait d'accéder au tube en écriture de manière ultra privilégiée par rapport à `write_char`[1]. Pour simplifier, on suppose qu'il n'y aura jamais deux appels à `write_char_urgent` en parallèle.

Par exemple, lorsqu'un tube est plein et que plusieurs processus sont bloqués en attendant de pouvoir y insérer des caractères, si l'un d'entre eux est bloqué dans `write_char_urgent`, alors le prochain caractère écrit dans le tube doit être le sien...

Donnez le code de `write_char_urgent` ainsi que la nouvelle version de `write_char`.

[1] Pour cette question, on considère que les applications n'utilisent plus l'opération `write`.