G. Mercier, Y. LeBorgne, R. Namyst le 09/01/2004

Maîtrise d'informatique — Systèmes d'exploitation
Correction du Devoir Surveillé du 28/11/2003

Remarque préliminaire : *Certaines réponses proposées aux questions de cours sont volontairement "verbeuses", et ce dans un but pédagogique. Bien sûr, il n'était pas nécessaire de fournir autant de détails pour répondre correctement aux questions (et donc obtenir le maximum de points) ...*

1 Appels système (exercice d'échauffement)

Question 1 Dans un environnement multi-tâches (et *a fortiori* dans un environnement multi-utilisateurs), chaque processus se voit attribuer un nombre fini de *droits* (permissions). L'accès limité à son propre espace d'adressage en est un exemple. Les mécanisme des droits affectés aux fichiers en est un autre. Pour assurer le respect de ces droits et, plus généralement, afin d'empêcher l'accès direct à certaines fonctionnalités offertes par le matériel, bon nombre d'opèrations délicates doivent être exécutées sous-contrôle du système. C'est justement le rôle d'un appel système. Son principe est le suivant :

- Chaque opération nécessitant un accès contrôlé n'est pas exécutable directement par les processus. Un tel service est proposé par le noyau du système et est identifié par un numéro d'appel système.
- Un processus désirant exécuter un appel système (ex : `read`) déclenche une interruption logicielle (ex : `syscall` sur un processeur MIPS) après avoit stocké dans un registre le numéro correspondant (ex : 3 sous Linux). Les paramètres suivants sont également passés dans des registres, ou placés sur la pile lorsqu'ils sont nombreux.
- L'interruption provoque le basculement en mode privilégié et le déroutement de l'exécution vers une routine prédéfinie du noyau. Cette routine aiguille l'exécution vers la bonne routine de service (ex :`sys_read`) en utilisant une table. Le processus exécute donc, en mode privilégié, un traitement sur lequel il n'a pas le contrôle, qui est supposé fiable et sécurisé. Ce traitement peut donc vérifier les permissions du processus et refuser l'opération en cas de problème.
- Au retour de l'appel système (c'est-à-dire au retour de l'interruption), le processeur bascule de nouveau en mode utilisateur, et le code de retour est contenu dans un des registres.

Ce type de mécanisme est incontournable car l'utilisateur n'a pas accès à toutes les instructions et certaines ne peuvent s'executer qu'en mode superviseur.

Question 2 Même si système fournit un pointeur dans son espace d'adressage, il ne pourra pas faire confiance à l'application lors du passage d'argument et devra donc vérifier dans une table que le pointeur est correct. Cette tâche est équivalente à l'utilisation d'un identifiant. De plus même si le système à une confiance aveugle en l'application, une désallocation puis réallocation malheureuse de l'objet sera difficile à détecter.

Comme implémentation on propose d'utiliser deux niveaux de table, celle du premier niveau étant propre à chaque processus, la seconde concernant le système entier. Les applications utilisent un identifiant pour coder les objets du noyau. La table du premier niveau associe à cet identifiant et au type de l'objet un indice dans la seconde table. Ensuite le système consulte la seconde table pour enfin obtenir l'adresse de l'objet. La seconde table contient aussi le nombre de processus ayant connaissance cet objet, ce qui permet de gérer plus simplement les désallocations des objets. (Exemple : gestion d'un descripteur de fichier utilisé par plusieurs processus)

2 Processus légers

Question 1 Un processus léger (*thread*) est un flot d'exécution partageant éventuellement l'intégralité de son espace d'adressage avec d'autres processus légers. Un tel processus est nécessairement encapsulé au sein d'un processus lourd lorsqu'il est visible au niveau utilisateur. On distingue deux principaux modes d'ordonnancement de ces processus.

Niveau utilisateur

Pros Leur gestion est performante car pas (ou peu) d'appels système (création, destruction, changement de contexte, etc.). La politique d'ordonnancement peut être adaptée ou changée aisément (par

exemple en changeant de paquetage de threads). Plus généralement, les fonctionnalités sont plus souples (rares limites arbitraires sur le nombre de threads, etc.).

Cons L'ordonnanceur du système ignore l'existence de ces threads, ce qui occasionne les désagréments suivants : un appel système bloquant provoque le blocage de tous les threads du processus englobant ; la répartition du temps CPU n'est pas effectuée sur la base du nombre de threads, ce qui est parfois injuste ; il est impossible d'exploiter le parallélisme d'une machine multiprocesseurs au sein d'un processus.

Niveau noyau

Pros Tous les inconvénients cités précédemment n'existent plus. Les constructeurs fournissent souvent des versions "MT-safe" des bibliothèques standard vis-à-vis des threads du noyau.

Cons La gestion de ces threads passe par le système, et n'est donc pas si légère que cela... L'ordonnancement est figé dans le noyau du système.

Question 2 Dans le modèle d'ordonnancement hybride l'éxécution d'un processus est gérée par un ensemble de threads noyau. Lorsque ces threads sont executés sur un processeur sous le contrôle de l'ordonnanceur du noyau, leur role est de choisir un thread utilisateur prêt à l'éxécution du processus dont ils ont la charge. Ce choix de thread utilisateur est effectué par un ordonnanceur géré par l'application.

Lorsqu'un thread de niveau utilisateur effectue un appel système bloquant le thread noyau qui le propulse est aussi bloqué mais un autre thread noyau (éventuellement créé pour l'occasion) peut executer d'autres threads utilisateurs du même processus.

Question 3 Un code réentrant (*Multithread-safe*) est un code que deux threads peuvent executer simultanément ou qu'un thread peut réexecuter par un appel récursif. Une fonction qui manipule sans protection particulière une variable du tas ou une ressource matériel dans le noyau, n'est pas réentrante. Pour rendre un code réentrant on peut soit créer à chaque appel les objets manipulés par la fonction (c'est efficace dans la pile mais ce n'est pas toujours possible), soit mettre en place des mécanismes de synchronisation pour garantir l'exclusion mutuelle lors de la manipulation des objets communs. Notez que la synchronisation est parfois obligatoire comme lors de la manipulation d'une ressource matérielle.

Certaines fonctions de la bibliothèque standard peuvent être très difficiles à rendre *MT-safe*, notamment si elles ont des effets de bord internes. Parfois, il est même nécessaire de modifier leur interface. C'est le cas de fonctions telles que `readdir` (voir `readdir_r`) ou encore `strtok`.

Question 4 Une file globale permet de gérer plus simplement des variables globales comme par exemple la priorité des threads. De plus le rééquilibrage de charge est plus simple car les processeurs libres sont sûr de trouver du travail dans cette file. Par contre les couts en synchronisations sont plus important car lors de son choix d'ordonnancement le processeur doit s'assurer de l'exclusion mutuelle sur la file globale. De plus, et c'est un avantage des files propres aux processeurs, on a interêt à maintenir le même espace d'addressage sur un processeur pour conserver le TLB. Le cas idéal du point de vue de la mémoire est d'avoir un seul espace d'adressage par processeur.

3 Gestion Mémoire

Question 1 Cf cours.

Question 2 La taille de l'espace d'adressage d'un processus est limitée à 4 Go (1024 tables de 1024 entrées référençant des pages de 4096 octets). La place occupée par la table des pages d'un processus varie non seulement en fonction de la taille du processus, mais aussi en fonction de l'agencement des pages.

Si l'espace utilisé est compact (toutes les pages virtuelles sont contiguë), alors l'occupation est minimale. Le meilleur cas correspond à un espace utilisé multiple de 4 Mo. Le ratio $\frac{\text{taille de la table}}{\text{données du processus}}$ tend vers 1/1024.

Le pire des cas correspond à un espace discontinu composé d'une page tous les 4 Mo - 4096 octets : le système doit alors allouer une table secondaire pour chacune des pages utiles du programme ! Le ratio tend alors vers 1 ! Heureusement, en pratique cela n'arrive jamais car les processus sont principalement composés de quelques zones de mémoire contiguë (code, pile, tas, etc.)

4 Sémaphores construits avec des moniteurs

```
typedef struct {
   lock l;
   condition c;
   int valeur;
} semaphore;

void sem_init(semaphore *s, int value)
{
   s->valeur = value;
   lock_init(&s->l);
   cond_init(&s->c);
}

void P(semaphore *s)
{
   lock_acquire(&s->l);
   if(--s->valeur < 0)
      cond_wait(&s->c, &s->l);
   lock_release(&s->l);
}

void V(semaphore *s)
{
   lock_acquire(&s->l);
   if(++s->valeur <= 0)
      cond_signal(&s->c, &s->l);
   lock_release(&s->l);
}
```

5 Nachos et les sémaphores

Question 1 La stratégie adoptée par Nachos est de laisser les *threads* effectuant une opération `P` entrer en compétition avec les *threads* "fraîchement" réveillés. C'est pourquoi un *thread* reveillé par une opération `V` ré-exécute le même code que la première fois qu'il est entré dans la fonction `P`.

Question 2 Il a donc un risque d'aboutir à des situations de famines puisque que, par exemple, un processus ne cessant d'entrer et sortir d'une section critique pourra empêcher un autre processus d'y entrer à son tour si l'ordonnancement n'est pas préemptif. Dans l'exemple suivant, lorsque le *thread* courant exécute le premier `V`, il réveille peut-être un autre *thread* : qu'importe, puisque si le *thread* courant conserve la main il va ré-acquérir le sémaphore sans problème...

```
sem.P();
sem.V();
sem.P();
sem.V();
```