Université de Strasbourg — Année 2008/2009
Département d'Informatique — L3 Informatique

Fondements des Systèmes d'Exploitation
Examen de première session - 8 janvier 2009

Durée : 2h.
Documents, calculatrices, portables interdits.
Le barême donné est indicatif.

Exercice 1 - Ordonnancement et threads (8 points)

Vous répondrez à ces questions en utilisant le langage C, sans oublier de le commenter. La seule fonction système à votre disposition est `malloc()`.

On considère un système d'exploitation gérant des processus et des threads. L'ordonnanceur, préemptif, ordonnance les threads uniquement, à l'aide d'un système de file d'attentes par priorité à 32 niveaux de priorité. Pour simplifier l'implémentation, vous pouvez considérer qu'un processus est simplement un thread "leader" d'un groupe de threads.

Lorsqu'un thread rend la main à l'ordonnanceur suite à l'expiration de son quantum ou un appel système, sa nouvelle priorité est calculée selon la formule suivante. `DeltaPri` est une fonction externe qui va affecter plus ou moins de priorité au thread concerné, selon différents paramètres - ne vous en souciez pas ; chaque thread dispose d'un niveau de priorité (*nice-level*).

```
pri = nicelevel + DeltaPri(threadID)
```

De plus, les threads devenus inactifs suite à un appel système, sont placés dans une file d'attente séparée *en attente de ressource*, jusqu'à ce que l'appel système termine.

1. Proposez une structure de données contenant les informations relatives aux processus/threads et à leur ordonnancement. Donnez également la structure de données des 32 files d'attentes par priorité, et de la file d'attente des threads en attente de ressources.

```
typedef struct th
{
   int threadID;      /* identifiant de thread */
   struct th *leader; /* pointeur vers le thread leader,
                         NULL si leader */

   int niceLevel;     /* nice level */
   int priLevel;      /* priorité */
```

```
   /* Listes d'ordonnancement (dans la structure !) : */
   struct th *next;   /* pointeur vers le suivant dans la liste */
   struct th *prev;   /* pointeur vers le précédent dans la liste */
} Thread;

Thread *priLevels[32];/* tableau de 32 files (doublement chaînées)
                         de threads ordonnaçables */
Thread *enAttente;    /* file des threads en attente de ressources */
```

2. Écrivez la fonction appelée lorsqu'un thread rend la main à l'ordonnanceur, qui met à jour la priorité d'un thread et qui le déplace dans la file d'attente de threads en attente de ressources.

Ces deux fonctions seront utilisées au long de cet exercice et facilitent la compréhension de la suite :

```
/* insertion d'un thread en tête dans une liste doublement chaînée
 * de Threads */
void InsereEnTete( Thread *t, Thread **liste )
{
   t->next = *liste;
   t->prev = NULL;
   if( *liste )
      (*liste)->prev = t;
   *liste = t;
}

/* suppression d'un thread dans une liste doublement chaînée
 * (qu'il soit en tête, en queue, ou au milieu) */
void Supprime( Thread *t, Thread **liste )
{
   /* chainage du suivant du précédent */
   if( t->prev )
      t->prev->next = t->next;    /* en milieu ou fin de liste */
   else
      *liste = t->next;                  /* en début de liste */
   /* chainage du précédent du suivant */
   if( t->next )                        /* s'il y a un suivant */
      t->next->prev = t->prev;
}
```

Et voici la réponse à la question :

```
void ThreadYield( Thread *t )
{
   /* supprime le thread t de sa file d'attente actuelle : */
   Supprime( t, &priLevels[t->priLevel] );

   /* calcule son nouveau niveau de priorité */
   t->priLevel = t->niceLevel + DeltaPri(t->threadID);

   /* le place dans la file des threads en attente de ressource */
   InsereEnTete( t, &enAttente );
}
```

3. Écrivez la fonction qui déplace le thread de cette file d'attente vers la file d'attente des threads ordonnançables.

```
void ThreadSchedule( Thread *t )
{
   /* supprime le thread t de la file attente de ressource : */
   Supprime( t, &enAttente );

   /* place le thread dans la file de priorité adéquate */
   InsereEnTete( t, &priLevels[t->priLevel] );
}
```

4. Lorsqu'un nouveau processus est créé, il n'a qu'un thread, qui hérite du nice-level du thread qui l'a créé. Écrivez la fonction qui crée ce processus, le thread associé, et le place dans la file d'attente d'ordonnancement adéquate.

```
/* Crée un nouveau processus, retourne la structure Thread créée */
Thread *NewProcess( Thread *createur )
{
   Thread *n;
   /* alloue la structure thread et la remplit : */
   n = malloc( sizeof(Thread) );
   /* fonction qui donne un nouveau numéro de thread */
   n->threadID = NewNumThread();
```

```
        n->leader = NULL; /* c'est un leader */
        /* héritage du nicelevel du thread créateur */
        n->niceLevel = createur->niceLevel;

        /* calcule sa priorité */
        n->priLevel = n->niceLevel + DeltaPri( n->threadID );

        /* place le thread dans la file de priorité adéquate */
        InsereEnTete( n, &priLevels[n->priLevel] );

        return( n );
    }
```

Exercice 2 - Pagination (5 points)

L'algorithme de choix d'une page mémoire à remplacer et un élément très important pour l'efficacité du système de remplacement de pages dans le contexte du swapping. Cependant, il n'existe pas d'algorithme optimal implémentable dans un système d'exploitation généraliste où l'on ne connait pas à l'avance les pages qui seront accédées.

Voici un exemple d'accès à des numéros de page mémoire :

`3 2 1 0 3 2 4 3 2 1 0 4`

1. Montrez, à l'aide de cet exemple, que l'algorithme FIFO avec 3 cadres mémoires peut donner de meilleurs résultats que ce même algorithme avec 4 cadres mémoire (c'est l'*anomalie de Belady*).

FIFO, 3 cadres :

3	2	1	0	3	2	4	3	2	1	0	4
3	3	3	0	0	0	4	4	4	4	4	**4**
	2	2	2	3	3	3	**3**	3	1	1	1
		1	1	1	2	2	2	**2**	2	0	0

3 hits.

FIFO, 4 cadres :

3	2	1	0	3	2	4	3	2	1	0	4
3	3	3	3	**3**	3	4	4	4	4	0	0
	2	2	2	2	**2**	2	3	3	3	3	4
		1	1	1	1	1	1	2	2	2	2
			0	0	0	0	0	0	1	1	1

2 hits.

Il n'y a que 2 cache hits avec 4 cadres alors qu'il y en a 3 avec 3 cadres.

2. Montrez que l'algorithme de remplacement LRU (*least recently used*) peut être moins efficace que FIFO à l'aide de ce même exemple.

LRU 3 cadres :

3	2	1	0	3	2	4	3	2	1	0	4
3	3	3	0	0	0	4	4	4	1	1	1
	2	2	2	3	3	3	**3**	3	3	0	0
		1	1	1	2	2	2	**2**	2	2	2

2 hits.

Il y a 2 hits avec LRU 3 cadres, soit un de moins que FIFO 3 cadres. CQFD.

LRU, 4 cadres (*pas nécessaire*) :

3	2	1	0	3	2	4	3	2	1	0	4
3	3	3	3	**3**	3	3	**3**	3	3	3	4
	2	2	2	2	**2**	2	2	**2**	2	2	2
		1	1	1	1	4	4	4	4	0	0
			0	0	0	0	0	0	1	1	1

4 hits.

3. Quel est le nombre de défauts de pages minimal qu'il pourrait y avoir avec cet exemple, lorsque la mémoire est composée de 3 et de 4 cadres ?

Le nombre de défaut de pages minimal qu'il pourrait y avoir est donné par l'algorithme OPT.

OPT 3 cadres :

3	2	1	0	3	2	4	3	2	1	0	4
3	3	3	3	**3**	3	3	**3**	3	1	1	1
	2	2	2	2	**2**	2	2	**2**	2	0	0
		1	0	0	0	4	4	4	4	4	**4**

5 hits.

OPT 4 cadres :

3	2	1	0	3	2	4	3	2	1	0	4
3	3	3	3	**3**	3	3	**3**	3	3	0	0
	2	2	2	2	**2**	2	2	**2**	2	2	2
		1	1	1	1	1	1	1	**1**	1	1
			0	0	0	4	4	4	4	4	**4**

6 hits.

Exercice 3 - Questions diverses (7 points)

Répondez à ces questions en quelques lignes (*maximum* 8 lignes par question).

1. Quel est le rôle d'un système d'exploitation généraliste comme linux ? Comment parvient-il à assumer ce rôle ?

Le rôle d'un SE généraliste est de fournir une interface entre l'utilisateur et le matériel : permettre aux utilisateurs d'une machine d'*exploiter* au mieux cette machine : accès

aux ressources, partage et protection des ressources (mémoire, processeur, fichiers, et autres entrée-sorties).

2. Quels sont les composants d'un système d'exploitation ? Précisez en particulier ce que sont le *bottom-half* et le *top-half*.

Le SE est constitué d'un *noyau*, qui est un ensemble de fonctions et de structures stockées en mémoire. Le top-half contient l'ensemble des fonctions accessibles au processus qui tournent sur la machine (appels-systèmes, on parle aussi d'interruption logicielle), alors que le bottom-half est appelé par une interruption matérielle (suite à une entrée-sortie sur un périphérique par exemple).

3. Qu'est-ce qu'un changement de contexte ?

Il y a un changement de contexte lorsqu'un processus qui s'exécutait rend la main pour laisser la place à un autre processus sur le processeur. Le contexte du processus est constitué de l'ensemble des *registres* du processeur, qui sont nécessaires à sa restauration, afin de permettre qu'il continue à s'exécuter dans les mêmes conditions qu'au moment où il a été interrompu. Le contexte d'un processus qui n'est pas en train de s'exécuter est stocké dans sa structure processus.

4. Quelle est la différence entre un driver (*pilote de périphérique*) en mode caractère et un driver en mode bloc ?

Un driver en mode caractère transmet des données du périphérique vers la mémoire (ou l'inverse), un octet après l'autre. Il est nécessaire de faire un appel système pour chaque octet transmis (read et write, les deux routines du driver, lit ou écrit un seul caractère à la fois).

Un driver en mode bloc transmet des données par bloc entiers, de taille maximale fixée par le driver. Il passe pour cela par un buffer en mémoire, le *buffer-cache*, et la routine du driver *strategy()* est chargée de la synchronisation entre le buffer-cache et le périphérique.

5. Qu'est-ce qu'un filesystem ?

Un filesystem est une organisation du disque. Il permet d'accéder à un disque selon une structure hiérarchique de répertoires et fichiers ayant un nom unique, de protéger les données entre les différents utilisateurs, et de dater les fichiers.

6. Un processus en cours d'exécution voit son image mémoire découpée en trois segments : TEXT, DATA et STACK. Quelle est l'utilité de la segmentation ?

La segmentation permet de protéger le segment TEXT en écriture. Les deux autres sont accessibles en lecture et en écriture (la distinction entre ces deux segments ne sert donc à rien ici).

Elle permet aussi, et surtout, de découper la mémoire en espaces logiques començant tous à l'adresse logique 0 : les données commencent à l'adresse 0 du segment DATA, idem pour le code (TEXT) et pour la pile (STACK), ainsi que pour les bibliothèque qui pourraient être chargées. Cela facilite grandement le fonctionnement du compilateur et du chargeur de programmes, qui n'a pas besoin de recalculer toutes les adresses de saut (jump ou call assembleur), ou des données, au moment où un programme ou une bibliothèque est chargé en mémoire.

7. À quoi sert le flag *copy-on-write* placé parfois sur une page mémoire accédée par un processus ? Dans quel cas est-il utilisé ?

Ce flag est utilisé pour indiquer que la page correspondante doit être dupliquée en mémoire si l'un des processus qui y accède y effectue une écriture.

Il est utilisé lors du fork. Le système n'a pas besoin de recopier tout l'espace mémoire du processus père lorsqu'il crée le fils : il suffit de marquer toutes les pages *copy-on-write* et de dupliquer les tables des pages du père dans le fils. Les pages seront ensuite copiées à la demande si un des processus les modifie.