Devoir SMB137 2009/2010

Première Partie

1) Gestion mémoire - Contexte

La majorité des systèmes d'exploitation incluent un ou plusieurs services d'allocation mémoire.
L'un des plus connus est représenté par les deux fonctions "malloc()" et "free" disponibles sur les systèmes compatibles POSIX (tous les Unix, Linux, etc.).
Les noyaux des systèmes incluent également des services d'allocation mémoire génériques de type malloc/free. Les noyaux Unix incluent également des allocateurs de mémoire dédiés à des usages spécifiques, comme par exemple l'allocateur SLAB de FreeBSD, Linux ou Solaris qui utilisent le principe du cache pour optimiser les opérations d'allocation et de libération de structures de données tout en minimisant la fragmentation mémoire.

Le micro-noyau temps-réel Chorus exporte un service d'allocation mémoire dédié aux applications superviseur et aux drivers comprenant les appels système svMemAlloc() et svMemFree() qui gèrent plusieurs pools de blocs mémoire de tailles fixes comprises entre 32 et 2048 octets.

Ces allocateurs sont en fait des variations d'allocateur de type "best-fit" s'appuyant le plus souvent sur un allocateur de pages, la page étant dans ce cas l'unité de libération du ramasse-miettes associé.

2) Devoir – Partie 1

On se propose de réaliser pour des applications embarquées un allocateur mémoire qui opère sur une zone mémoire réservée à cet usage au démarrage de l'application, dans le double but de lui garantir un espace de travail donné et de limiter la mémoire totale occupée durant son exécution.
L'interface exporté par l'allocateur mémoire se compose des trois fonctions suivantes :

```
/*
 * Initialise le service de gestion mémoire dans la zone de
 * mémoire d'adresse de début "zone_addr" et de taille
 * "zone_size" utilisée pour l'allocation de blocs de mémoire
 * de tailles variables.
 * Cette fonction est appellée au démarrage de l'application.
 */
void mem_zone_init(void *zone_addr, unsigned int zone_size);

/*
 * Alloue un bloc mémoire de taille au moins égale à "bloc_size
 * et renvoie l'addrese de début du bloc alloué.
 * Retourne "NULL" en cas d'échec.
 */
void *mem_bloc_alloc(unsigned int bloc_size);

/*
 * Libère le bloc mémoire d'adresse "bloc_addr" et de taille
```

```
 * "bloc_size" précédement alloué par la fonction
 * mem_bloc_alloc();
 */
void mem_bloc_free(void *bloc_addr, unsigned int bloc_size);
```

2.1 Rappels

Décrire brièvement les principes des deux méthodes d'allocation mémoire dites de type "best-fit" et de type "first-fit".
Enoncer leurs principaux avantages et inconvénients respectifs.

2.2 Algorithmes

Réaliser en pseudo-code ou en langage C les fonctions de l'allocateur mémoire décrit précédemment, en choisissant une méthode de type "best-fit" ou de type "first-fit".
Pour se faire, on décrira :

- les structures de données utilisées pour représenter les blocs de mémoire libres et/ou les blocs de mémoire alloués,
- les algorithmes des trois fonctions exportées par le service de gestion mémoire,
- le principe de ramasse-miettes mis en oeuvre.

3) Devoir – Partie 2

La partie 2 du devoir porte sur la synchronisation entre threads d'une même application et est basée sur l'allocateur de blocs réalisé en première partie.
On introduit dans l'allocateur les deux problématiques suivantes :

1. Gérer l'accès en exclusion mutuelle aux données partagées par les fonctions de l'allocateur exécutées en parallèle par plusieurs threads de priorités quelconques.
2. Gérer la mise en attente des threads dont la requête d'allocation ne peut être satisfaite immédiatement, et le réveil d'une ou plusieurs threads en attente lors de la libération d'un bloc mémoire, si nécessaire.

3.1 Exclusion Mutuelle

Décrire les données partagées dont il faut protéger les acces concurrents.

Ajouter dans l'allocateur de blocs mémoire les structures de données et les opérations de synchronisation nécessaires pour assurer l'accès en exclusion mutuelle aux données partagées et garantir ainsi la cohérence de leur état.
Pour ce faire, on se basera sur les services système suivants :

```
mutex_t mutex; /* déclaration d'un [objet de type] mutex */

/*
 * Initialise le mutex "mutex".
```

```
 * La fonction "mutex_int()" doit être appellée avant de
 * pouvoir utiliser le mutex avec les fonctions
 * "mutex_acquire()" et "mutex_release()".
 */
void mutex_init(mutex_t *mutex);

/*
 * Obtient le mutex "mutex".
 * Bloque la thread appellante si le mutex appartient à une autre
 * thread, jusqu'à ce que le mutex soit relâché.
 */
extern void mutex_acquire(mutex_t *mutex);

/*
 * Relâche le mutex "mutex".
 * Si d'autres threads sont bloquées en attente du mutex,
 * réveille la première d'entre elles pour lui transmettre le mutex.
 */
void mutex_release(mutex_t *mutex);
```

Avec cette interface, l'accès en exclusion mutuelle à la variable partagée `int shared_var` protégée par le mutex `shared_var_lock` se programme de la façon suivante :

```
/* initialisé par mutex_init(&shared_var_lock) */
mutex_t shared_var_lock;
int shared_var = 0;

     mutex_acquire(&shared_var_lock);
        /*
         * Incrémente "shared_var" en exclusion mutuelle
         */
     shared_var++;
     mutex_release(&shared_var_lock);
```

3.2 Attente de mémoire Bloquante

Introduire le support de l'attente bloquante de mémoire lorsqu'une requête d'allocation ne peut être immédiatement satisfaite.
Décrire les structures de données utilisées pour représenter les threads bloquées en attente de mémoire.

Réaliser en pseudo-code ou en langage C :

- la mise en attente de la thread courante par la fonction mem_bloc_alloc() lorsque un bloc de la taille requise ne peut être alloué immédiatement.
- le réveil d'une ou de plusieurs des threads éventuellement en attente par la fonction `mem_bloc_free()` lorsque le bloc libéré le justifie.

Il est important de veiller à éviter toute possibilité d'inter-blocage lors de la mise en oeuvre des opérations de mise en attente et de réveil des threads combinées aux opérations de synchronisation introduites en 2.1 pour assurer l'accès en exclusion mutuelle aux données partagées.

On se basera sur les services système suivants permettant de bloquer la thread courante et de réveiller une thread :

```
/*
 * Toute thread est désignée par un identificateur
 * de type "thread_t".
 * Retourne l'identificateur de la thread courante.
 */
thread_t thread_self(void);

/*
 * Bloque la thread courante.
 * Retourne après que la thread a été réveillée par une autre
 * thread avec la fonction "thread_awake".
 */
extern void thread_wait(void);

/*
 * Réveille la thread d'idendificateur "thread_id" bloquée
 * dans la fonction "thread_wait".
 */
extern void thread_awake(thread_t thread_id);
```