PNL - 4I402

TP 07 – Mécanismes de gestion de la mémoire dans le noyau Linux

Redha Gouicem, Maxime Lorrillere et Julien Sopena

mars 2019

Le but de ce TP est d'étudier différents mécanismes offerts par le noyau Linux que vous permettront d'améliorer l'efficacité de la gestion de la mémoire de votre module de monitoring.

Exercice 1 : Monitoring de l'activité d'un processus (suite)

Dans cet exercice nous reprenons le module de monitoring que vous avez réalisé lors de l'exercice 4 du précédent TP. Le but est ici de l'améliorer pour conserver en mémoire l'historique des mesures que vous effectuez.

On souhaite conserver dans une liste en mémoire l'historique des statistiques CPU du processus monitoré. Pour cela, si vous ne l'avez pas fait lors du TP précédent, définissez dans votre module la structure `struct task_sample` comme suit :

```
struct task_sample {
  struct list_head list;
  u64 utime;
  u64 stime;
};
```

Cette structure contient ces statistiques mesurées à un instant t ainsi qu'un champ permettant de la stocker dans une liste.

Question 1

Ajoutez dans votre `struct task_monitor` une tête de liste, un entier pour compter le nombre de *samples* que vous stockez dans votre liste et une structure `struct mutex` pour protéger les accès à cette liste. L'initialisation de la structure `struct mutex` passe par la macro `mutex_init`.

Question 2

Plutôt que d'afficher périodiquement les statistiques CPU, votre `kthread` va sauvegarder ces statistiques. Pour cela, ajoutez une fonction `void save_sample(void)` qui sera appelée chaque seconde par votre

`kthread`. Cette fonction aura pour rôle d'allouer une nouvelle `struct task_sample`, de l'initialiser à l'aide de votre fonction `get_sample`, puis de l'ajouter dans la liste de votre `struct task_monitor`.

Attention ! Pensez à protéger les accès à votre liste correctement à l'aide du *mutex*. Pour cela, vous pourrez utiliser les fonctions `mutex_lock` et `mutex_unlock`.

Question 3

On souhaite pouvoir récupérer les dernières statistiques enregistrées à l'aide du sysfs en lisant le fichier `/sys/kernel/taskmonitor`.

Modifiez votre fonction `taskmonitor_show` que vous avez implémenté lors du précédent TP pour faire en sorte d'afficher les derniers échantillons. Par exemple :

```
insmod ./taskmonitor.ko target=267
cat /sys/kernel/taskmonitor
   pid 267 usr 0 sys 0
   pid 267 usr 2 sys 0
   pid 267 usr 0 sys 0
   pid 267 usr 3 sys 0
```

Attention ! N'oubliez pas que le *buffer* du sysfs est limité à la taille d'une page (`PAGE_SIZE`). Vous veillerez donc à n'affichez que les derniers échantillons, dans la limite de `PAGE_SIZE` octets.

Question 4

Dans la fonction de terminaison de votre module, libérez correctement la mémoire occupée par les `struct task_sample`.

Exercice 2 : Récupération automatique de la mémoire, le *shrinker*

Actuellement, votre module alloue de la mémoire périodiquement et ne la libère que lorsqu'il est déchargé. Au bout d'un certain temps, votre module aura alloué toute la mémoire disponible et votre machine virtuelle plantera. Il est donc nécessaire de rendre régulièrement la mémoire que l'on utilise.

Question 1

Le noyau fourni une *API* (*shrinker*) permettant de réclamer des objets alloués en cas de pression mémoire. Ouvrez le fichier `include/linux/shrinker.h` et analysez le fonctionnement de cette *API*. Quelles sont les fonctions que vous devez implémenter ? Quels sont leurs rôles ?

Question 2

Implémentez votre propre *shrinker*, et utilisez les fonctions `register_shrinker` et `unregister_shrinker` pour activer/désactiver celui-ci. Assurez vous que les `struct task_sample` sont bien manipulés en exclusion mutuelle dans le reste de votre code.

Astuce : cette *API* est utilisée à de multiples endroits dans le noyau, vous pouvez vous inspirer des implémentations existantes pour mieux comprendre comment elle fonctionne.

Question 3

Vérifiez que votre *shrinker* fonctionne. Pour cela, commencez par augmenter la fréquence d'échantillonnage de votre module de monitoring. Ensuite, vous pouvez générer de la pression mémoire pendant que

votre module monitore un processus. Une solution est de lire tous les fichiers de l'arborescence pour générer de l'activité mémoire :

```
find /usr /var -type f -print0 | xargs -0 cat > /dev/null
```

Exercice 3 : Gestion efficace de la mémoire avec les *slab*

Question 1

Dans votre module, affichez la taille de votre `struct task_sample` avec `sizeof` et avec la fonction `ksize`. Pourquoi la taille renvoyée par `ksize` est plus grande que la taille réelle de votre objet ?

Question 2

Pour optimiser les allocations mémoire, le noyau Linux dispose d'une couche appelée *slab layer*, qui dispose de plusieurs implémentations distinctes. Quel est le principe et l'intérêt du *slab layer* ? Des informations détaillées sont disponibles à l'adresse suivante : https://census-labs.com/news/2012/01/03/linux-kernel-heap-exploitation/.

Question 3

Créez une `struct kmem_cache` qui sera chargée de faire les allocations de vos `struct task_sample` en passant par le *slab layer*.

La fonction `KMEM_CACHE()` permet d'initialiser cette structure. Les fonctions `kmem_cache_alloc` et `kmem_cache_free` permettent d'allouer et libérer des objets. Modifiez votre module pour tenir compte de ces modifications.

Astuce : cette *API* est utilisée à de multiples endroits dans le noyau, vous pouvez vous inspirer des implémentations existantes pour mieux comprendre comment elle fonctionne.

Exercice 4 : Pré-allocation de la mémoire avec les *mempools*

En cas de forte pression mémoire, il peut devenir impossible d'allouer les `struct task_sample` dont nous avons besoin. Une solution est d'utiliser un *pool* de mémoire pré-allouée qui nous permettra d'allouer nos objets y compris lorsque la mémoire se fait plus rare : l'*API mempool* du noyau Linux permet justement cela.

Question 1

Analysez l'*API* présente dans le fichier `include/linux/mempool.h`. Quelles sont les types et les fonctions dont vous avez besoin ? Quelles fonctions devez vous implémentez ?

Question 2

Modifiez votre code pour qu'il utilise un *pool* de mémoire.

Astuce : cette *API* est utilisée à de multiples endroits dans le noyau, vous pouvez vous inspirer des implémentations existantes pour mieux comprendre comment elle fonctionne.

Exercice 5 : Récupération « *au besoin* » de la mémoire : *kref*

Actuellement, l'affichage des statistiques du processus monitoré doit être protégé par un *mutex* pour éviter que l'objet que vous affichez soit libéré pendant que vous le manipulez. Une solution pour éviter cela est d'utiliser un compteur de référence sur les objets utilisés.

Question 1

Le noyau Linux fournit une *API* permettant de compter des références sur des objets. Ouvrez le fichier `include/linux/kref.h` et analysez cette *API*. Quelles fonctions allez vous utiliser pour manipuler les références de vos objets ?

Question 2

Modifiez le code de votre module pour que les objets de type `struct task_sample` soient gérés via des références. Assurez vous que l'affichage de vos `struct task_sample` puisse être effectué *en même temps* que leur récupération via le *shrinker*. Pour cela, vous modifierez la signature de votre fonction `save_sample` pour qu'elle renvoie une `struct task_sample` qui pourra être manipulée une fois ajoutée dans la liste de votre `struct task_monitor`.

Question 3

Combien de références pointent vers une `struct task_sample` juste avant son affichage ?

Exercice 6 : Accès aux informations de monitoring depuis l'espace utilisateur

L'affichage des informations de monitoring sur la sortie console pose problème : cela consomme des ressources, en particulier si l'on augmente la fréquence d'échantillonnage. Il est donc nécessaire d'afficher les statistiques uniquement lorsque nécessaire.

L'utilisation du sysfs, que vous avez étudié lors du précédent TP, présente un avantage certain. Cependant, son tampon est limité à la taille d'une page (`PAGE_SIZE`, 4 ko).

Une autre approche est d'utiliser le système de fichiers virtuel *debugfs* pour qu'il soit possible d'avoir accès à ces informations depuis l'espace utilisateur : il suffira alors d'ouvrir le fichier `/sys/kernel/debug/taskmonitor` pour avoir accès à l'historique des mesures.

Question 1

Le fichier `Documentation/filesystems/debugfs.txt` contient la documentation de ce système de fichiers. Quelle est la différence entre le *sysfs* et le *debugfs* ? Quelles fonctions allez vous utiliser pour créer et détruire le fichier `/sys/kernel/debug/taskmonitor` ?

Question 2

Pour créer ce fichier, nous avons besoin de créer les *file operations* (*fops*) correspondant à notre fichier. Pour simplifier, le noyau dispose de l'*API* **seq_file** fournissant les fonctions pour manipuler des fichiers *séquentiels*. Analysez la documentation de cette *API* disponible dans le fichier `Documentation/filesystems/seq_file.txt`.

Quelle structure devez vous implémenter pour utiliser l'*API* **seq_file** ? Quelle(s) fonction(s) de la `struct file_operations` devrez vous implémenter ?

Question 3

Modifiez votre module pour que les statistiques de monitoring soient accessibles via le fichier `/sys/kernel/debug/taskmonitor`.

À titre d'exemple, vous pouvez vous inspirez du code du module *kmemleak* que vous avez utilisé lors du TP débogage qui est présent dans le fichier `mm/kmemleak.c`.
Attention ! *kmemleak* est nettement plus compliqué que votre module, ne vous laissez pas avoir ;-)

Exercice 7 : Monitoring de plusieurs processus simultanément

Votre module est pour le moment assez limité : il n'est possible de monitorer qu'un seul processus à la fois, et vous devez décharger/recharger votre module si vous voulez monitorer un nouveau processus.

Le but de cet exercice est d'améliorer le code de votre module pour qu'il soit capable de monitorer simultanément plusieurs processus.

Pour cela, vous allez configurer votre module depuis l'espace utilisateur, en utilisant le fichier *debugfs* que vous avez créé dans l'exercice 4. Par exemple, pour activer le monitoring du processus de PID 312 :

```
echo 312 > /sys/kernel/debug/taskmonitor
```

De même, pour désactiver son monitoring, on pourra utiliser :

```
echo -312 > /sys/kernel/debug/taskmonitor
```

Question 1

Modifiez votre `struct task_monitor` pour que vous puissiez les chaîner entre eux. Au lieu d'avoir une variable globale, déclarez une tête de liste `struct list_head tasks`. Pour tester, vous pouvez ajouter un nouveau paramètre à votre module pour être capable de monitorer 2 processus en même temps.

Question 2

Modifiez votre fonction `monitor_fn` pour qu'elle enregistre les statistiques CPU/mémoire de toutes les `struct task_monitor` de votre liste.

Question 3

Ajouter une fonction `write` dans la `struct file_operations` de votre fichier *debugfs*. Cette fonction devra traiter l'ajout et la suppression de processus à monitorer.

Vous pouvez vous inspirer de l'implémentation de la fonction `write` du module *kmemleak*.