Implémentation d'une mémoire partagée transactionnelle entre processus

Pré-requis
– Connaître les bases du langage C

Objectifs techniques
– Fournir une bibliothèque et un module noyau Linux permettant de partager de la mémoire entre processus.
– Implémenter un méchanisme de mémoire transactionnelle.

Objectifs pédagogiques
– Comprendre et savoir utiliser une machine virtuelle (qemu)
– Comprendre le fonctionnement des modules noyau Linux
– Comprendre comment l'espace utilisateur peut intéragir avec l'espace noyau (appels système, système de fichiers...)
– Comprendre le fonctionnement d'un driver Linux de type character
– Comprendre le fonctionnement des mutexes
– Comprendre les problèmes de composabilité des mutexes et les problèmes de passage à l'échelle
– Comprendre le fonctionnement d'une mémoire transactionnelle : avantages et inconvénients
– (Comprendre le fonctionnement des signaux et savoir les utiliser)
– (Comprendre setjmp/longjmp)
– (Comprendre le fonctionnement de l'appel système mmap)
– (Savoir utiliser un débugger noyau avec une machine virtuelle : kgdb)

Dernière modification : 21 octobre 2010.
Contact : Sylvain HENRY (sylvain.henry@inria.fr)

1 Introduction

On cherche à partager de la mémoire entre différents processus sous Linux. Cependant, à la différence des moyens existants (mémoire partagée POSIX, mmap d'un même fichier, etc.) on souhaite utiliser un mécanisme de transaction. Il s'agit donc d'implémenter une mémoire transactionnelle logicielle (*Software Transactional Memory, STM*).

Un mécanisme de transaction fonctionne de la façon suivante :

1. Un processus commence une transaction
2. Il effectue différentes opérations (lectures, écritures)
3. Il tente de valider la transaction (commit). Si elle échoue, il recommence au début de la transaction, sinon il est assuré que ses modifications ont été prises en compte.

Cette approche est dite "optimiste" contrairement à une approche "pessimiste" où on réserverait l'accès à la mémoire à un seul processus à la fois (en bloquant l'accès avec un mutex par exemple). C'est-à-dire qu'on estime que les différents processus effectuent le plus souvent des actions sur des zones disjointes de la mémoire et peuvent donc travailler de façon concurrente sans problème.

Cette méthode est employée par les systèmes de gestion de bases de données (SGBD). Par exemple, lorsqu'une banque transfère de l'argent d'un compte à l'autre, il est inutile de bloquer l'accès à toute la base de données (i.e. tous les autres comptes) pendant toute la durée du transfert.

1.1 Bibliothèque

Pour implémenter ce système de mémoire partagée entre processus, vous devrez fournir une bibliothèque partagée (.so). Cette bibliothèque devra fournir l'API suivante :

- `int fd = stm_create(char *filename, int size);`
  Crée un fichier spécial (mknod) configuré (ioctl) avec la taille spécifiée et l'ouvre.
- `void stm_start(int fd);`
  Indique le début d'une transaction
- `int stm_commit(int fd);`
  Indique la fin d'une transaction. Renvoie 0 si réussi, 1 sinon.
- `void stm_cancel(int fd);`
  Annule une transaction.
- `void stm_retry(int fd);`
  Ré-exécute la transaction à partir de l'appel à `stm_start`

1.2 Module noyau

La mémoire partagée sera située en espace noyau. Vous devez donc créer un module noyau qui la gèrera.

La communication entre l'espace utilisateur (les applications) et l'espace noyau s'effectue par les appels système (*syscalls*). Le nombre d'appels système étant limité, il est très rare que de nouveaux appels soient ajoutés. On utilise donc ceux qui existent déjà.

Sous Linux, la méthode la plus courante est d'utiliser les appels système dédiés à la gestion des fichiers. On crée donc des fichiers spéciaux, généralement situés dans `/dev`. On verra que lorsqu'on écrit ou lit dans un fichier spécial, le noyau appelle les méthodes du driver correspondant au type du fichier spécial.

1.3 Déroulement du projet

Ce projet est constitué de plusieurs étapes de plus en plus difficiles. Vous devez valider chaque étape avant de passer à la suivante.

2 Étape 1 : module noyau Hello-World

Cette section a pour objectif de vous familiariser avec le fonctionnement des modules du noyau Linux. Pour cela, vous allez tout d'abord compiler et charger un module très simple (listing 1).

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int stm_init(void) {
    printk("Hello world!\n");
    return 0;
}

static void stm_exit(void) {
    printk("Bye, cruel world\n");
}

module_init(stm_init);
module_exit(stm_exit);
```

Listing 1 – Module Hello-World

Pour compiler ce module, vous allez utiliser un Makefile standard (listing 2). Le Makefile fourni suppose que le code du module se trouve dans le fichier `stm_module.c`.

Utilisez les commandes `make` pour compiler et `make clean` pour supprimer les fichiers générés. Si la compilation termine correctement, vous devez obtenir un fichier `stm.ko` et vous pouvez passer à l'étape suivante. Sinon, corrigez les bugs éventuels dans votre code ou dans le Makefile.

2.1 Chargement et déchargement du module

Pour charger un module noyau, vous devez être root. Utilisez la commande `insmod stm.ko` pour charger le module. Vous pouvez vérifier que le module est bien chargé avec la commande `lsmod | grep stm`. Pour décharger le module, utilisez la commande `rmmod stm`.

```
MODULE_NAME = stm

$(MODULE_NAME)-objs = stm_module.o

ifneq ($(KERNELRELEASE),)

obj-m   := $(MODULE_NAME).o

else

KDIR    := /lib/modules/$(shell uname -r)/build
PWD     := $(shell pwd)

all:
        $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
        $(MAKE) -C $(KDIR) M=$(PWD) clean

endif
```

Listing 2 – Makefile

Vous pouvez vérifier que le module Hello-World fonctionne correctement en consultant le fichier de log du noyau (`/var/log/kernel.log` sous ArchLinux). Vous pouvez aussi utiliser la commande `dmesg`.

2.2 Remarques

Vous pouvez constater que dans le code du module, on utilise la méthode `printk` là où on aurait utilisé `printf` dans une application usuelle. En effet, en espace noyau les fonctions de la `libc` telles que `printf`, `malloc`, etc. ne sont pas disponibles. Néanmoins certaines ont des équivalents dans le noyau avec un nom parfois similaire.

Il est important de comprendre que le code exécuté dans le noyau se trouve soit dans des modules chargés, soit dans le noyau lui-même. Vous pouvez consulter la liste de tous les symboles (dont les méthodes) disponibles dans le noyau avec la commande `cat /proc/kallsyms`.

Les manuels du noyau ont normalement été installés dans l'image qemu

mise à disposition. Vous pouvez donc les consulter comme d'habitude (e.g. `man printk`).

3 Étape 2 : driver de type character

Dans cette section vous allez implémenter un driver Linux de type `character`.

Avant de commencer, vous devez savoir que chaque driver Linux a un numéro unique appelé *major*. C'est ce numéro qui plus tard permettra de faire le lien entre le fichier spécial et le driver. La liste des numéros réservés par les drivers actuellement chargés dans le noyau se trouve dans `/proc/devices`. Vous devez donc en sélectionner un qui est libre.

3.1 Déclaration du driver

Pour créer un nouveau driver de type `character`, vous devez effectuer les action suivantes :

1. Déclarer une structure `stm_fops` de type `struct file_operations` [1]
2. Écrire les méthodes `open`, `release`, `read` et `write` dont vous trouverez les prototypes dans la définition de la structure `file_operations`.
3. Placer les adresses de ces différentes méthodes dans les champs de la structure `stm_fops` précédemment créée.
4. Enregistrer le nouveau driver auprès du noyau avec la méthode `register_chrdev` lors de l'initialisation du module. Vous passerez en arguments le *major* choisi, le nom de driver "stm" et l'adresse de `stm_fops`.
5. "Dés-enregistrer" le driver lors du déchargement du module avec la méthode `unregister_chrdev`.

1. Cette structure est définie dans `/usr/src/linux-*/include/linux/fs.h`

3.2 Test du driver

Pour tester le nouveau driver, vous allez créer un fichier spécial avec la commande `mknod fichier c major 0`. Vous pouvez choisir le nom de fichier que vous voulez. Les arguments `c` et `major` permettent d'indiquer le driver auquel est rattaché ce fichier spécial. Le dernier argument est appelé *minor*. Ce numéro permet à un driver d'identifier différents fichiers spéciaux (chacun ayant un numéro *minor* différent). Ici nous n'en avons pas l'utilité donc n'importe quelle valeur fera l'affaire.

Vous devez ensuite donner les droits en lecture et écriture sur le fichier avec `chmod o+rw fichier`.

Vous pouvez ensuite tester les méthodes `read` et `write` du driver en utilisant respectivement `cat fichier` et `echo abc > fichier` [2]. Vous pouvez vérifier avec `printk` que vos méthodes sont bien appelées.

4 Étape 3 : allocation de mémoire

Le noyau fournit la méthode `vmalloc`, similaire à `malloc` en espace utilisateur. Utilisez cette méthode pour allouer une zone de mémoire lors de l'initialisation du module.

Modifiez les méthodes `read` et `write` afin qu'elles lisent et écrivent dans la zone de mémoire allouée (sans dépasser !). Pour copier des données de l'espace noyau vers l'espace utilisateur, le noyau fournit les méthodes `copy_to_user` et `copy_from_user`.

Vous devez penser à mettre à jour le dernier argument des méthodes `read` et `write` en fonction du nombre de données lues ou écrites.

Vous pouvez tester le module comme à l'étape précédente. Cette fois ci, les données écrites avec `echo` doivent pouvoir être lues avec `cat`.

2. Les méthodes `read` et `write` retournent le nombre d'octets lus/écrits. Vous devez renvoyer une valeur cohérente pour éviter les plantages. Par exemple `read` peut renvoyer 0 et `write` peut renvoyer le nombre d'octets que l'appelant voulait écrire (troisième argument de `write`).

4.1 Configuration de la taille de la mémoire à allouer

Ajoutez le support de l'appel système `ioctl` au driver afin de permettre la configuration de la taille mémoire à allouer à partir d'un programme.

5 Étape 4 : verrouillage par mutex

Vous allez maintenant implémenter l'API utilisateur dans une bibliothèque partagée (.so). Implémentez la méthode `stm_create`. Pour cela, utilisez les appels système `mknod`, `ioctl` et `open` [3].

Utilisez un mutex dans le module noyau pour protéger l'accès aux données partagées. Implémenter les méthodes `stm_start` (verrouillage du mutex), `stm_commit` (libération du mutex).

Les lectures et les écritures dans la mémoire partagée s'effectuent avec les appels système `read` et `write`.

6 Étape 5 : écriture différée

Ajoutez dans le module noyau un système pour différer les écritures dans la mémoire partagée. Les écritures sont donc effectuées lors de l'appel à `stm_commit`, pas avant. Vous pouvez donc maintenant implémenter `stm_cancel`.

Lors des opérations de lecture, vous devez dorénavant vérifier si la zone mémoire lue a été écrite précédemment au cours de la transaction. Le cas échéant vous devez renvoyer la valeur écrite. Dans le cas contraire, vous devez renvoyer la valeur située en mémoire partagée comme précédemment.

3. Utilisez `man -s2 appel` pour obtenir la page de manuel de l'appel système et non de la commande shell.

7 Étape 6 : transactions

On constate qu'un processus qui met du temps à réaliser sa transaction pénalise tous les autres. Pour éviter ça, on va maintenant utiliser un mécanisme de transaction.

- Pour chaque transaction, on sauvegarde la liste des données lues et écrites. Utilisez `current->pid` pour obtenir le numéro du processus appelant.
- On effectue le lock du mutex uniquement au moment du commit et on rejoue la liste des actions sauvegardées de la façon suivante :
  - On compare les données précédemment lues (et sauvegardées) avec le contenu actuel de la mémoire partagée : si les données précédemment lues ont été modifiées entre temps par une autre transaction, la transaction est annulée, sinon on effectue les écritures et la transaction est réussie

Attention, les données lues au cours d'une transaction peuvent maintenant être incohérentes. Il faut donc bien les tester pour éviter certaines erreurs (divisions par zéro, etc.).

8 Fonctionnalités supplémentaires

S'il vous reste du temps, vous pouvez essayer d'ajouter les fonctionnalités suivantes.

8.1 Signaux et rollback

En cas de lecture de données incohérentes, une interruption peut être déclenchée (par exemple en cas de division par 0). On souhaite redémarrer automatiquement la transaction dans ce cas là. Pour cela on propose le mécanisme suivant :

- Appel à `setjmp` et activation de la gestion des signaux dans `stm_start`
- `longjmp` en cas de déclenchement de signal au cours d'une transaction
- Désactivation de la gestion des signaux dans `stm_commit` et `stm_cancel`

Le code de la transaction sera donc rejoué automatiquement en cas d'erreur, il faut donc qu'il soit référentiellement transparent Vous pouvez maintenant implémenter l'API `stm_retry` qui effectue simplement un `longjmp` pour rejouer la transaction (utile lorsqu'on détecte une incohérence dans les données lues)

8.2 mmap

Ajoutez le support de l'appel système `mmap` dans le module noyau.

A Préparation du projet

Cette section explique comment l'image fournie a été créée. **Vous n'avez pas besoin de le refaire**.

Au préalable, il faut avoir installé qemu.

Création de l'image On utilise la commande suivante pour créer une image de taille extensible jusqu'à celle indiquée (3Go) : `qemu-img create -f qcow2 linux.img 3GB`

Télécharger une distribution Linux J'ai choisi ArchLinux mais n'importe laquelle peut faire l'affaire. L'architecture supportée par la distribution doit être supportée par qemu. [4]

Installer la distribution Lancer qemu en considérant l'image créée comme étant un disque dur et l'image de la distribution téléchargée comme étant un CD : `qemu-system-x86_64 -hda linux.img -cdrom archlinux-2010.05-core-x86_64.iso`

4. Voir la liste des architectures supportées par qemu à l'adresse suivante http ://wiki.qemu.org/download/qemu-doc.html#intro_005ffeatures

Vous devez utiliser la command `qemu-system-*` qui correspond au type d'architecture que vous souhaitez virtualiser (ici x86_64) et utiliser une distribution adaptée à cette architecture.

B Utilisation de qemu

Création d'une image personnelle Pour utiliser l'image qemu mise à votre disposition, vous n'avez pas besoin de la copier. Utilisez simplement le mode copy-on-write de qemu.

Pour cela, créez une image personnelle, qui ne contiendra que vos modifications par rapport à l'image fournie, avec la commande suivante : `qemu-img create -f qcow2 -b image_fournie.img img_perso.img`

Configuration par défaut Le mot de passe du compte "root" est "abc123" dans l'image fournie.

Exécution Lancer le système virtuel avec la commande `qemu-system-x86_64 -hda img_perso.img`

Pour sortir le pointeur de la fenêtre qemu, il faut appuyer sur Ctrl et Alt simultanément.

Internet Il est possible de se connecter à Internet. Pour cela exécutez la commande `dhcpcd`. Vous pouvez donc utiliser `ssh (scp)`, `wget`, `svn` ou `git` pour transférer des fichiers dans le système virtuel..

Il est recommandé de travailler en dehors du système virtualisé avec un système de gestion de versions (git, svn, etc.).

Vous pouvez installer certains logiciels avec les commandes suivantes :
- emacs : `pacman -S emacs-nox`
- ssh (scp) : `pacman -S openssh`
- vim : `pacman -S vim`