Programmation Système Avancée

TP4 – Tâches utilisateurs et appels systèmes

Introduction

A ce stade de développement, votre kernel est capable de manipuler l'affichage en mode texte, de lire les touches du clavier et dispose d'un système de fichiers basique capable de localiser et lire des fichiers en mémoire.

Une partie manquante mais extrêmement importante est la gestion des tâches utilisateurs. En effet, pour arriver au but ultime d'un shell (certes très primitif) pouvant exécuter des applications, votre kernel doit nécessairement implémenter la gestion de tâches simples, mais sécurisées grâce à la virtualisation de la mémoire ainsi qu'un mécanisme d'appels systèmes. Ces derniers forment un « pont » entre le mode kernel (mode privilégié - ring 0) et le mode utilisateur (mode limité - ring 3) afin que les applications puissent utiliser les services offerts par le kernel.

Objectif

Le but final de ce travail pratique est d'aboutir à un mini système d'exploitation interactif monotâche mettant à disposition un shell fournissant des fonctionalités de base et permettant d'exécuter des applications utilisateurs. Il est important que les applications utilisateurs soient isolées les unes des autres et surtout isolées du kernel afin qu'une tâche buggée ne puisse ni affecter d'autres tâches, ni compromettre le bon fonctionnement du kernel.

Les différents points à implémenter sont les suivants :

- Au niveau du kernel (répertoire `kernel`) :
  - Système monotâche permettant la gestion de plusieurs tâches utilisateurs s'exécutant en mode non privilégié (limité).
  - Implémentation d'appels systèmes grâce au mécanisme d'interruptions logicielles.
- Au niveau utilisateur (répertoire `user`) :
  - Un shell simple.
  - Au moins une mini application de votre choix.
  - Implémentation d'une micro librairie C (ulibc) utilisée par les applications utilisateurs et le shell.

Fichiers fournis

Afin de vous faciliter le travail, l'archive `tp4.tar.gz` vous est fournie. Celle-ci contient plusieurs fichiers contenant du code sur lequel vous pouvez baser votre implémentation. Les fichiers fournis sont décrits dans les sections qui suivent.

Fichiers côté kernel

Ces fichiers se trouvent dans le répertoire `kernel`.

task.h

Contient le type `tss_t` décrivant la structure TSS associée à une tâche.

task_asm.s

Contient 2 fonctions assembleur :

- `load_task_register` : charge le task register avec le sélecteur de TSS passé en argument.
- `call_task` : commute vers la tâche dont le sélecteur de TSS est passé en argument.

gdt.c

Contient 4 fonctions et 1 macro :

- `gdt_make_tss(tss_t *tss, uint8_t dpl)` : créé et renvoie un descripteur de TSS.
- `gdt_entry_t gdt_make_ldt(uint32_t base, uint32_t limit, uint8_t dpl)` : créé et renvoie un descripteur de LDT.
- `uint gdt_entry_to_selector(gdt_entry_t *entry)` : renvoie le sélecteur d'une entrée dans la GDT.
- `setup_task()` : fonction **d'exemple** implémentant l'initialisation d'une tâche utilisateur au niveau de privilège ring 3 :
  - créé 2 descripteurs dans la GDT (indices 4 et 5).
  - créé une LDT contenant un segment de code et un segment de données partageant le même espace d'adresses.
    - les segments de code et données sont tous deux mappés à l'adresse 0x800000 et possèdent une limite de 64KB.
  - allocation d'une pile kernel de 8KB.
- `GDT_INDEX_TO_SELECTOR` : macro permettant de convertir un indice de descripteur dans la GDT en un sélecteur.

A la fin de la fonction `gdt_init()`, se trouve un code permettant d'initialiser le TSS initial du kernel afin que le processeur puisse y sauver son état courant avant de commuter vers la première tâche. Ce TSS est initialisé avec une pile kernel de 64KB.

Enfin, la fonction `load_task_register` est appelée afin de faire pointer le task register sur le TSS initial défini auparavant.

idt.c

Ce fichier contient seulement une ligne de code définissant une nouvelle entrée dans la table des descripteurs d'interruptions (IDT). A l'indice 48 est installé le descripteur référençant le gestionnaire d'appels systèmes du kernel (`_syscall_handler`). Ce descripteur est de type trap gate, car il n'est pas souhaitable que les interruptions matérielles soient inhibées durant l'exécution des appels systèmes. Le niveau de privilège du descripteur est de type ring 3, car le but est de pouvoir exécuter l'interruption logicielle 48 depuis le niveau de privilège utilisateur (ring 3). Ainsi, un appel système se fera simplement grâce à l'interruption logicielle 48.

syscall_asm.s

Contient la fonction assembleur :

- `_syscall_handler` : il s'agit de la partie « basse » du gestionnaire d'appels systèmes. Cette fonction appelle ensuite la fonction C, `syscall_handler` en prenant soin de lui passer le numéro d'appel système, les arguments et le sélecteur TSS de la tâche ayant fait l'appel système.

syscall.c

Contient la fonction :

- `int syscall_handler(syscall_t nb, uint32_t arg1, uint32_t arg2, uint32_t arg3, uint32_t arg4, uint32_t caller_tss_selector)` : il s'agit de la partie « haute » du gestionnaire d'appels systèmes. Lorsqu'une interruption 48 est déclenchée, cette fonction est appelée par la partie « basse » du gestionnaire d'appels systèmes. Les arguments permettent de déterminer le numéro de l'appel système désiré, ses arguments potentiels ainsi que la tâche ayant fait l'appel via son sélecteur de TSS.

Fichiers côté applications utilisateurs

Ces fichiers se trouvent dans le répertoire `user`.

app.ld

Similairement au fichier de link `kernel.ld` se trouvant dans le répertoire kernel, ce fichier défini, pour les applications utilisateurs, où les différentes sections (code dans la section `.text`, données non initalisée dans la section `.bss`, etc.) seront mappées au moment de l'édition des liens ainsi que le format (ou type) de fichier binaire généré. Ce fichier spécifie aussi que la première section à figurer dans le fichier binaire généré est la section `.entrypoint`. Ceci permet de garantir que la fonction `entrypoint` est toujours le point d'entrée de l'application, car elle se trouve dans la section `.entrypoint`.

app.c

Exemple de squelette d'une application utilisateur. Il s'agit d'un fichier quasiment vide. La seule contrainte est que le point d'entrée de l'application soit la fonction `main`. Contrairement à un système UNIX classique, la fonction `main` ne possède aucun argument et ne retourne rien à l'appelant.

app_stub.s

Ce code doit impérativement être lié à toute application utilisateur. Celui-ci implémente le point d'entrée de l'application (fonction `entrypoint`) et s'occupe d'appeler la fonction `main`, puis de revenir à la tâche appelante.

syscall.s

Contient la fonction assembleur `syscall` qui permet d'exécuter un appel système vers le kernel. Cette fonction prend 5 arguments : le numéro d'appel système suivi de 4 arguments. Pour chaque appel système, c'est à vous de définir quels arguments seront réellement nécessaires et utilisés.

Fichier commun

Ce fichier se trouve dans le répertoire `common`.

syscall_nb.h

Contient le type énuméré :

- `syscall_t` : défini les numéros des différents appels systèmes implémentés par le kernel.

Cahier des charges

Partie kernel

Tâches et exécution de programmes utilisateurs

Un des buts de ce travail pratique est que votre kernel soit capable de charger des programmes (tâches) en mémoire afin de les exécuter. L'exécution d'un programme par le kernel n'est rien d'autre que l'exécution des 3 étapes ci-dessous :

1. Chargement du fichier binaire contenant le programme à exécuter à une adresse mémoire choisie par le kernel ; le code et les données du programme sont donc chargés à cette adresse mémoire. On part du principe que le code se situe au tout début du fichier binaire, ce qui est le cas si le programme a été lié avec le fichier `app.ld` fourni (cf. section précédente).
2. Initialisation de la tâche qui exécutera le code du programme. Le TSS de la tâche doit être initialisé correctement : en particulier, le programme a dû être chargé à l'adresse où est mappé le segment de code de la LDT et le pointeur d'instruction doit pointer sur l'adresse de la première instruction (adresse 0).
3. Commutation vers la nouvelle tâche en spécifiant le sélecteur de TSS à la fonction `call_task` qui vous a été donnée.

Bien que ces tâches ne s'exécutent pas de manière concurrente, il est impératif que le kernel puisse en exécuter plusieurs de manière séquentielle. Par exemple, on peut tout à fait se trouver dans une situation où le shell (qui est une tâche) exécute la tâche app1 qui à son tour exécute la tâche app2. L'imbrication des tâches devient alors : shell → app1 → app2. Dans ce scénario, 3 tâches sont imbriquées d'où la nécessité à ce que le kernel puisse gérer plusieurs tâches.

Voici les consignes à respecter en ce qui concerne les tâches :

- Votre kernel doit pouvoir gérer un minimum de 8 tâches.
- Chaque tâche doit utiliser 2 entrées dans la GDT.
- Chaque tâche doit définir sa propre LDT.
- L'espace d'adressage de chaque tâche doit être au moins de 1MB.
- L'espace d'adressage de chaque tâche doit être disjoint de toutes les autres tâches et du kernel.
- L'espace d'adressage de chaque tâche est alloué au démarrage du kernel et est complètement statique ; il ne change donc pas au cours du temps.
- La pile kernel de chaque tâche doit être au moins de 64KB.

N'oubliez pas de mettre en place le TSS initial afin que l'état du processeur soit correctement sauvegardé lors de la première commutation de tâche.

Pour des raisons de simplicité, le seul moyen à disposition de votre kernel pour exécuter une nouvelle tâche est de charger un programme depuis le disque pour ensuite l'exécuter. On désire donc implémenter une fonction similaire à la fonction `exec` des système UNIX.

La fonction `exec` à implémenter doit respecter les consignes ci-dessous :

- Le nom du programme à charger et exécuter doit être un argument de la fonction.
- La fonction doit trouver une tâche libre afin d'y charger le contenu du fichier ; ensuite, la fonction commutera vers cette nouvelle tâche.
- La fonction doit pouvoir indiquer à l'appelant si l'exécution a réussi ou échoué.
- S'il n'y a plus de tâche de libre, alors la fonction doit échouer et retourner un code d'erreur à l'appelant.

Appels systèmes

La gestion des appels systèmes est réalisée grâce à l'interruption logicielle 48. Il vous faut donc ajouter l'entrée correspondante dans la table IDT de votre kernel.

En terme de fonctionalités à disposition des applications utilisateurs, votre kernel doit *au minimum* implémenter les appels systèmes ci-dessous :

- `syscall_putc` : affiche un caractère à la position courante du curseur.
- `syscall_puts` : affiche une chaîne de caractères à la position courante du curseur.
- `syscall_exec` : exécute, après l'avoir chargé en mémoire, le programme dont le nom est passé en argument.
- `syscall_getc` : renvoie le caractère lu du clavier.
- `syscall_file_stat` : renvoie les informations de type `stat_t` du fichier dont le nom est passé en argument.
- `syscall_file_read` : lit le contenu du fichier dont le nom est passé en argument dans la zone mémoire spécifiée.
- `syscall_file_remove` : supprime le fichier dont le nom est passé en argument.
- `syscall_file_iterator` : Renvoie un itérateur permettant d'itérer sur les fichiers du système de fichiers ; cet appel système est un wrapper autour de la fonction `file_iterator` du système de fichiers.
- `syscall_file_next` : itère sur le fichier pointé par l'itérateur ; cet appel système est un wrapper autour de la fonction `file_next` du système de fichiers.
- `syscall_get_ticks` : renvoie le nombre de ticks courant.

Vous remarquerez que ces appels systèmes ne font rien d'autre que d'encapsuler des fonctions déjà existantes dans le kernel.

Un point important est la gestion des erreurs dans les appels systèmes, car ceux-ci peuvent bien sûre échouer (lorsque c'est applicable). A titre d'exemple, l'appelant doit pouvoir déterminer si la lecture d'un fichier a réussi ou pas.

Kernel

Après initialisation des différents sous-systèmes (GDT, IDT, affichage, etc.), le kernel s'occupera simplement de charger puis exécuter le shell développé dans la partie « applications utilisateurs ». A noter que si l'utilisateur décide de sortir du shell avec la commande « exit », le système affichera

simplement un message notifiant l'utilisateur, puis s'arrêtera.

Partie applications utilisateurs

Maintenant que vous avez un kernel pouvant charger et exécuter des programmes utilisateurs et que le mécanisme d'appels systèmes est en place, vous disposez des briques de base nécessaires à la réalisation d'applications utilisateurs. Il est important de comprendre que ces applications utilisateurs sont compilées complètement séparément du kernel. Les applications n'ont aucunes dépendences envers celui-ci, hormis les numéros d'appels systèmes. C'est donc maintenant que le répertoire `user`, vide jusqu'ici, va commencer à prendre tout son sens.

Shell

Arrivé à ce stade, votre kernel devrait être capable d'exécuter des programmes utilisateurs. Voilà déjà une petite révolution en soit !

La possibilité d'exécuter des programmes utilisateurs depuis le kernel est extrêmement utile, mais peu pratique d'un point de vue utilisateur. En effet, dans le cadre d'un système d'exploitation interactif, il est beaucoup plus flexible et utile de pouvoir charger des programmes dynamiquement à la demande et aussi de pouvoir manipuler les fichiers présent dans le système de fichiers de manière interactive. Ces fonctionalités ne sont rien d'autre que celles proposées par un shell.

Bien que le shell à implémenter soit volontairement très simple et primitif, il est toutefois demandé qu'il offre, au minimum, les commandes internes suivantes :

- `ls` : liste tous les fichiers du système de fichiers
- `cat file` : affiche le contenu du fichier `file`
- `rm file` : efface le fichier `file`
- `run file` : exécute le fichier `file`
- `ticks` : affiche le nombre de ticks courant
- `sleep N` : attend pendant N milli-secondes
- `exit` : sort du shell (même comportement que la commande `exit` de `bash`)
- `help` : affiche la liste des commandes disponibles

Veuillez aussi vous assurez que votre shell notifie l'utilisateur en cas de problèmes (commande inconnue, erreur d'une commande, etc.).

Mini application

En plus du shell, il vous est demandé d'implémenter, au moins, une mini application de votre choix.

La seule limite ici est celle de votre imagination !

Veuillez décrire précisément ce que fait votre application dans l'entête du fichier source principal.

Micro librairie C

Afin d'éviter à ce que chaque application réinvente la roue, vous allez implémenter une micro librairie C implémentant les fonctionalités de bases nécessaires aux programmes utilisateurs. Tout comme la libriarie GNU libc des systèmes GNU Linux, cette librairie C se situera au dessus des appels systèmes et offrira des fonctionalités de plus haut niveau.

Le rôle de la librairie C est d'offir aux applications utilisateurs les fonctionalités de base du système et aussi d'offrir une couche d'abstraction au dessus des appels systèmes mis à disposition par le kernel. Il est donc important qu'aucune application utilisateur, shell compris, n'appelle la fonction

`syscall`. C'est le rôle de la librairie C que d'exécuter les appels systèmes vers le kernel grâce à la fonction `syscall`. Par soucis de compatibilité, les applications doivent uniquement utiliser les fonctionalités mises à disposition par la librairie C et éviter d'interfacer le kernel directement.

Etant donné que les applications vont dépendre de la librairie C, il est probablement judicieux de commencer par le développement de celle-ci. Elle doit être implémentée dans les fichiers `ulibc.h` et `ulibc.c`.

Il est demandé que la librairie C offre *au minimum* les fonctions suivantes :

- Fonctions d'accès aux fichiers :
  - `int read_file(char *filename, uchar *buf);`
  - `int get_stat(char *filename, stat_t *stat);`
  - `int remove_file(char *filename);`
  - `file_iterator_t get_file_iterator();`
  - `int get_next_file(char *filename, file_iterator_t *it);`
- Fonctions de contrôle de processus (tâche) :
  - `int exec(char *filename);`
  - `void exit();`
- Fonctions sur les chaînes de caractères :
  - `uint strlen(char *s);`
  - `uint atoi(char *s);`
- Fonctions d'entrées/sorties :
  - `int getc();`
  - `void putc(char c);`
  - `void puts(char *str);`
  - `void printf(char *fmt, ...);`
- Fonctions liées au temps :
  - `void sleep(uint ms);`
  - `uint get_ticks();`

La fonction `exit` ci-dessus termine la tâche courante afin que la tâche parent puissent continuer son exécution. Vous êtes libre d'implémenter cette fonction comme bon vous semble à partir du moment où elle rempli le comportement demandé. A noter que l'appel à `exit` doit fonctionner correctement quel que soit l'emplacement où la fonction est appelée dans le code de l'application.

Makefile

Tout comme pour les répertoires `kernel` et `tools`, le répertoire `user` doit contenir son propre fichier makefile permettant de compiler la librairie C et les applications utilisateurs (dont le shell fait partie).

Remarques importantes

Le makefile global situé à la racine du système d'exploitation doit faire appel au sous makefile se

trouvant dans le répertoire `user`, afin de compiler les applications utilisateurs.

Le makefile global doit aussi se charger d'ajouter le shell et les applications créés dans le répertoire `user` à l'image du système de fichiers accédée par votre kernel. Si ce n'est pas le cas, le kernel ne pourra jamais lire, puis exécuter le shell. Il devrai bien-sûre se stopper et indiquer un message d'erreur fatal si c'est le cas.

QEMU ne réalise pas une émulation à 100 % correcte de l'architecture IA-32. En effet, pour des raisons de performance, lors des accès mémoire la limite des segments n'est pas testée systématiquement. Ce comportement semble dépendre de la version de QEMU utilisée.

Pour cette raison, il est recommandé d'exécuter votre système d'exploitation sur plusieurs émulateurs/machines virtuelles. Bochs est nettement plus lent que QEMU, mais propose une émulation du matériel parfois meilleure. Sur la page Cyberlearn du travail pratique, vous pouvez trouver le fichier `bochsrc`. Il s'agit du fichier de configuration utilisé par Bochs. Editez-le et remplacez `yourOS.iso` par l'image ISO de votre kernel et `yourfs.img` par l'image de votre système de fichiers. Pour exécuter Bochs, exécutez simplement la commande : `bochs -f bochsrc`

Questions

Veuillez développer vos réponses aux questions ci-dessous :

1. Qu'est-ce qui vous a le plus plus dans le développement de ce système d'exploitation ? Pourquoi ?
2. Qu'est-ce qui vous a le moins plus dans le développement de ce système d'exploitation ? Pourquoi ?
3. Quels seraient, selon vous, les 3 étapes suivantes à développer dans votre système d'exploitation ? Pourquoi ?
4. Sachant que les appels systèmes sont relativement prohibitifs en terme de performance, comment feriez-vous pour proposer aux applications utilisateur un affichage efficace et performant ?

Travail à rendre

Pour ce travail pratique, vous me rendrez :

- Les réponses aux questions posées dans la section précédente dans un document au format PDF. Notez qu'il est important que vous développiez et justifiez vos réponses le plus possible.
- Une archive au format ZIP comprenant l'arborescence complète de votre projet **ainsi** que le document PDF du point ci-dessus. Le projet étant un tout, votre archive doit contenir le code source complet que vous avez développé depuis le TP1.

Comme vous n'avez pas de rapport à me rendre pour ce TP, je tiens à ce que votre code soit **extrêmement lisible et particulièrement bien commenté**.

Le code doit respecter les consignes décrites dans le document **Consignes générales pour les travaux pratiques** sous la rubrique « Informations générales » sur la page Cyberlearn du cours.