Programmation Système Avancée

TP1 – Un mini kernel avec gestion d'affichage

Objectif

Le but de ce travail pratique (TP) est d'implémenter un bootloader multiboot chargeant un mini kernel capable de gérer l'affichage d'un PC en mode texte. Plus pécisément, les objectifs du TP sont les suivants :

- Mise en place d'un projet basé sur une hiérarchie de makefiles permettant le développement d'un système d'exploitation.
- Implémentation d'un bootloader multiboot (appelé par GRUB) exécutant un mini kernel.
- Implémentation d'un mini kernel permettant d'afficher du texte.
- Implémentation de la gestion de l'affichage en mode texte.
- Système final contenu dans une image ISO bootable.

Structure du projet

Afin de s'y retrouver et que tout le code ne se retrouve pas en vrac dans un seul répertoire, la stucture de base du système d'exploitation vous est donnée sous forme d'une archive disponible sur la page Cyberlearn du cours. L'archive se nomme `os_skeleton.tar.gz`. Elle contient les répertoires de base ainsi que les fichiers sur lesquels vous vous baserez pour développer votre système d'exploitation. Votre OS sera principalement divisé en deux parties distinctes : une partie kernel et une partie utilisateur que nous verrons plus tard dans le semestre. La structure du projet, telle que définie dans l'archive, est présentée ci-dessous :

- `kernel` : contient tous les fichiers sources du kernel ainsi qu'un makefile (vide) permettant de générer le binaire de votre kernel au format ELF.
- `grub` : contient la configuration de GRUB pour le chargement du kernel.
- `common` : contient les fichiers communs au kernel et aux applications utilisateurs.
- `user` : contient les applications utilisateurs (vide pour l'instant).
- `tools` : contient les outils permettant de manipuler votre future système de fichiers (vide pour l'instant).

Le répertoire racine, dans lequel se trouve les répertoires ci-dessus, contient un makefile (vide) parent permettant de générer le kernel, les applications utilisateurs, les outils, et aussi de lancer le boot de votre OS grâce à QEMU.

Bootloader multiboot

Le fichier `kernel/bootloader.s` est un squelette presque complet du bootloader qui a pour but d'appeler votre mini kernel implémenté en C. Pour que le bootloader soit complet et fonctionnel, il vous reste seulement à initialiser la pile et appeler la fonction du point d'entrée du kernel. J'attire votre attention sur le fait que ce bootloader, chargé par GRUB, est implémenté en code 32-bits et non 16-bits. En effet, GRUB se charge de configurer la table de descripteurs (GDT), puis de passer

le processeur en mode protégé 32-bits. L'implication de ceci est que vous ne devez pas toucher aux registres de segments, car ils ont déjà été initialisés par GRUB.

Vous devez donc modifier le bootloader fourni afin de :

- Réserver de l'espace mémoire pour la pile (minimum 256KB).
- Initialiser le pointeur de pile et le pointeur EBP (pointeur de « base » pour le stack frame) ; ceux-ci doivent avoir la même valeur.
- Appeler votre kernel ; il s'agit d'appeler la fonction qui est le point d'entrée du code de votre kernel en C.

A titre informatif pour un déboggage potentiel, les deux lignes ci-dessous écrivent directement dans la mémoire vidéo pour afficher un smiley jaune en haut à gauche de l'écran (cf. cours sur l'adressage de la mémoire vidéo en mode texte pour plus de détails) :

```
mov        eax,0xB8000
mov        word [eax],0x0E01
```

Création d'une image ISO contenant GRUB

Une image ISO 9660 est une image de système de fichiers aux caractéristiques propres pouvant être gravée sur un CDROM. Presque toutes les machines peuvent booter sur un CDROM et plus récemment, il est devenu possible de « graver » une image ISO sur une clé USB (avec `unetbootin` par exemple, ou même `dd`).

Pour qu'une image ISO soit bootable, il est nécessaire que GRUB ait été installé dans les premiers secteurs du disque. Les étapes decrites ci-dessous permettent de créer une image ISO contenant GRUB comme bootloader suivit d'une image de kernel hypothétique, `kernel.elf`.

En premier lieu, il faut créer un répertoire de travail dans lequel les fichiers seront copiés. Soit `pipo` le répertoire de travail, `kernel.elf` le kernel sur lequel on veut booter avec GRUB et `grub.cfg` la configuration de GRUB à utiliser. Les commandes suivantes permettent de créer `pipo.iso`, une image bootable du kernel au format ISO 9660 :

1. Créer l'arborescence : `pipo/boot/grub`
2. Copier `grub.cfg` dans `pipo/boot/grub/`
3. Copier `kernel.elf` dans `pipo/boot/`
4. Exécuter `grub-mkrescue -o pipo.iso pipo`

A noter que le fichier de configuration `grub.cfg` fait référence au kernel à charger (`kernel.elf`) ainsi qu'au répertoire `boot`.

Un premier kernel en C

Dans un premier temps et pour des raisons de test et déboggage, votre kernel C pourra être extrêmement simple. Il pourra par exemple afficher un caractère en haut à gauche de l'écran. Cela permettra de confirmer que celui-ci est correctement appelé par votre bootloader. Une fois ceci vérifié, vous pourrez passer au développement du kernel proprement dit.

Dans le cadre de ce TP, le kernel à développer a deux buts :

- Initialiser la table des descripteurs globaux (GDT, cf. cours).
- Un *mode* « normal » :

    - Afficher un message disant que la console est initialisée.
    - Afficher un message disant que la GDT est initialisée.
- Un *mode* « test » :
    - Exécuter une batterie de tests validant toutes les routines d'affichage implémentées.

Le *mode* défini ci-dessus sera défini à la compilation. Lors d'une exécution normale du kernel, nous ne voulons pas que tous les tests de validité soient effectués. Par contre, nous voulons avoir la possibilité de vérifier que le code est fiable et de fait fonctionne correctement, d'où la possibilité d'exécuter le kernel en mode de test.

Une possibilité pour ceci est d'utiliser les fonctionalités du préprocesseur pour ne compiler que certaines parties du code lorsqu'un symbole spécifique est défini. Le compilateur `gcc` permet de définir un symbole du préprocesseur directement en argument. A titre d'exemple, exécuter `gcc -DXXX` défini le symbole `XXX` dans tout le code source passé à `gcc`.

Par exemple, soit le code :

```
#include <stdio.h>

void main() {
    printf("XXX is ");
#ifdef XXX
    printf("defined\n");
#else
    printf("NOT defined\n");
#endif
}
```

En compilant ce code avec l'argument `-DXXX`, puis en l'exécutant cela donne :

```
$ gcc -DXXX test.c -o test && ./test
XXX is defined
```

En compilant ce code sans l'argument -DXXX, puis en l'exécutant cela donne :

```
$ gcc test.c -o test && ./test
XXX is NOT defined
```

Fonctions liées à la GDT

Le code lié à la GDT se trouve dans `gdt.c`, `gdt.h` et `gdt_asm.s`.

Inspectez le code déjà présent afin que vous en compreniez les points importants, puis mettez en place une table de descripteurs globaux contenant les trois descripteurs suivants (dans l'ordre) :

- Un descripteur vide (NULL).
- Un descripteur de code ayant un DPL de 0 et couvrant tout l'espace physique adressable.
- Un descripteur de données ayant un DPL de 0 et couvrant tout l'espace physique adressable.

Assurez-vous aussi que la limite du pointeur de GDT (variable `gdt_ptr`) est correcte et que sa base pointe sur l'adresse de la GDT en mémoire. A noter que le code dans `gdt.c` nécessite la fonction `memset`.

Fonctions de base

Il s'agit ici des fonctions de base utilisées sur presque n'importe quel système.

Dans le cadre de ce TP, vous devez au moins implémenter les trois fonctions suivantes :

- `void *memset(void *dst, int value, uint count);`
- `void *memcpy(void *dst, void *src, uint count);`
- `int strncmp(const char *p, const char *q, uint n);`

Veillez à respecter les prototypes listés ci-dessus. Dans le doute, utilisez le manuel `man` pour déterminer la sémantique de ces fonctions.

Fonctions d'accès aux périphériques

Les fonctions d'accès aux périphériques à l'aide des mécanismes de « ports », doit être implémenté en assembleur par l'appel des instructions machines `IN` et `OUT`.

Vous devrez fournir les fonctions suivantes, appelables depuis le code C :

- Ecrit à l'adresse `port` la valeur 8-bits `data` :

  `void outb(uint16_t port, uint8_t data);`
- Ecrit à l'adresse `port` la valeur 8-bits `data` :

  `void outw(uint16_t port, uint16_t data);`
- Retourne 8-bits lus à l'adresse `port` :

  `uint8_t inb(uint16_t port);`
- Retourne 16-bits lus à l'adresse `port` :

  `uint16_t inw(uint16_t port);`

A titre d'exemple, le code ci-dessous écrit le byte 4 dans le port 0x60 :

```
mov     dx,0x60
mov     al,4
out     dx,al
```

Similairement, pour lire la valeur du port 0x60 (toujours dans `AL/AX`):

```
mov     dx,0x60
in      al,dx
```

Votre code de gestion de curseur aura besoin d'accéder aux ports de la carte VGA et fera donc appel à certaines fonctions C définies ci-dessus.

Par convention, postfixez tout fichier source assembleur par `_asm.s` (comme cela est fait pour le fichier source `gdt_asm.s`).

Fonctions d'affichage

Le but des fonctions d'affichage est de pouvoir afficher du texte. On veut aussi pouvoir contrôler la position du curseur et surtout il est important que votre implémentation s'occupe de faire défiler le texte lorsque nécessaire.

L'implémentation des fonctions d'affichage devra se trouver dans un ou plusieurs fichiers séparés.

Je ne veux pas voir de code lié à l'implémentation de l'affichage dans le fichier kernel principal.

Vous devez au moins implémenter les fonctionalités décrites ci-dessous. A noter que les noms des fonctions et structures sont libres, de même que le arguments passés aux fonctions (types, sémantique, etc.).

- Fonction d'initialisation : initialise l'affichage en effaçant l'écran et en positionnant le curseur en haut à gauche.
- Fonction d'effacement : efface l'écran.
- Fonction(s) permettant de changer la couleur du texte ainsi que la couleur du fond.
- Fonction(s) permettant de récupérer la couleur du texte ainsi que la couleur du fond.
- Fonction permettant d'afficher un caractère à la position du curseur.
- Fonction permettant d'afficher une chaîne de caractères à la position du curseur.
- Fonction d'affichage à arguments variable similaire à `printf`, implémentant au moins les formats suivants :%c,%s,%d et %x
- Fonction permettant de définir la position du curseur.
- Fonction permettant d'obtenir la position du curseur.

Make et build

- Les dépendances de votre projet doivent être gérées correctement : si un fichier source du kernel (ou autre) est modifié, seulement celui-ci doit être recompilé et aucun autre.
- Exécuter `make` dans le répertoire racine de votre projet doit générer l'image ISO de votre OS (par exemple `kernel.iso`). Ce makefile racine doit appeler le makefile se trouvant dans le répertoire kernel qui doit lui uniquement créer le fichier ELF du kernel (par exemple `kernel.elf`).
- Exécuter `make run` doit lancer l'exécution de votre OS par QEMU (et bien entendu s'occuper de construire les dépendances nécessaires pour le faire).
- Exécuter `make clean` doit faire complètement le ménage en effaçant tous les fichiers intermédiaires générés (il ne doit rester que les fichiers sources et de configuration).
- Le répertoire `grub` doit uniquement contenir le fichier `grub.cfg`

Remarques importantes

- Rappelez-vous que vous n'avez pas de librairie C disponible ! Cela signifie que vous ne pouvez utiliser aucune des fonctions habituelles (`malloc`, `printf`, etc.). Vous ne pouvez donc pas faire d'allocations dynamiques.
- Réfléchissez à ce qui se passe-t-il lorsque vous sortez de la fonction `main`...
- Le code C de votre kernel ne devra pas être compilé comme du code C classique, car il ne doit pas dépendre des fonctions de la librairie C. Compilez-le avec les options suivantes :

  ```
  -std=gnu99 -m32 -fno-builtin -ffreestanding -Wall -Wextra -c
  ```

- L'édition des liens pour obtenir le kernel final devra être faite en spécifiant le fichier de script pour le linker `ld`. Vous passerez donc les options suivantes à `ld` :

  ```
  -Tkernel.ld -melf_i386
  ```

- A titre d'exemple, pour lier `kernel.o` (généré depuis `kernel.c`) avec `gdt_asm.o` (généré depuis `gdt_asm.s`) pour produire le fichier final `kernel.elf` :

  ```
  ld -Tkernel.ld -melf_i386 kernel.o gdt_asm.o -o kernel.elf
  ```

- Toutes les fonctions et variables propres au fichier source dans lequelles elles sont implémentées doivent être déclarées `static` (autrement les symboles sont globaux, donc visibles par le linker, ce qui peut engendrer des conflits de noms).
- N'oubliez pas que toute fonction globale (visible dans d'autres modules) doit avoir son entête de fonction déclarée avec la directive `extern` dans un fichier header (.h). Ceci est valable aussi bien pour une fonction implémentée en C ou en assembleur.

Travail à rendre

Vous me rendrez l'arborescence complète de votre projet (zippée) ainsi qu'un rapport expliquant le travail effectué.

Le code et le rapport doivent tous deux respecter les consignes décrites dans le document **Consignes générales pour les travaux pratiques** sous la rubrique "Informations générales" sur la page Cyberlearn du cours.