Programmation Système Avancée

TP1 – Mini kernel avec gestion d'affichage

Introduction

Ce travail pratique couvre la première brique du future système d'exploitation (OS) que vous allez réaliser : un mini kernel capable de gérer l'affichage d'un PC en mode texte.

Objectif

Les objectifs de ce travail pratique sont les suivants :

- Mise en place d'un projet basé sur une hiérarchie de makefiles permettant la compilation aisée du bootstrap, du kernel, et des applications utilisateurs (future).
- Implémentation d'un bootstrap (assembleur) qui alloue une pile kernel puis saute au mini kernel (C).
- Implémentation d'un mini kernel, écrit en C, capable de gérer l'affichage en mode texte.
- Le kernel doit s'occuper de :
  - Initialiser une GDT.
  - Initialiser l'affichage en mode texte VGA.
  - Effectuer des tests de validation lorsque exécuté en mode « test ».
- L'image du kernel au format ELF est chargée et exécutée par GRUB.
- Le système final doit être contenu dans une image ISO bootable.
- La plateforme d'exécution et de test est QEMU.

Structure du projet

Afin de structurer le projet et que tout le code ne se trouve pas en vrac dans un seul répertoire, la structure de base de l'OS vous est donnée sous forme d'une archive disponible sur la page Cyberlearn du cours. L'archive `os_skeleton.tar.gz` contient les répertoires de base ainsi que les fichiers sur lesquels vous vous baserez pour développer votre OS. Celui-ci sera essentiellement divisé en deux parties distinctes : une partie kernel et une partie utilisateur que nous aborderons plus tard dans le semestre. La structure du projet, telle que définie dans l'archive, est la suivante :

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

Le répertoire racine, dans lequel se trouvent les répertoires ci-dessus, contient le makefile (vide)

racine (ou parent) permettant de générer le kernel, les applications utilisateurs, les outils, et aussi de lancer le boot de l'OS grâce à QEMU. Ce makefile s'occupera d'appeler les makefiles des sous-répertoires.

Bootstrap multiboot

Le fichier `kernel/bootstrap_asm.s` est une ébauche incomplète du bootstrap multiboot ayant pour but d'appeler le mini kernel que vous implémenterez en C. Pour que le bootstrap soit complet et fonctionnel, il reste à initialiser la pile et appeler la fonction de point d'entrée du kernel. J'attire votre attention sur le fait que ce bootstrap, chargé par GRUB, est implémenté en code 32-bits et non 16-bits. En effet, au moment où GRUB procède au chargement de votre OS, il a déjà activé le mode 32-bits du processeur et mis en place sa propre table de descripteurs (GDT). Pour plus de détails, voir la section « Machine state » dans la spécification multiboot ici : https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Machine-state. Etant donné que les registres de segments sont déjà initialisés par GRUB, ne les touchez pas pour le moment.

GRUB place dans le registre EBX, l'adresse d'une structure contenant des informations détectées durant le boot du PC (RAM détectée, etc.) ainsi que d'autres informations liées à GRUB lui-même (adresses des modules, ligne de commande, etc.). Le contenu de cette structure est défini par la structure `multiboot_info_t` se trouvant dans le fichier `multiboot.h`.

La première partie du TP est de modifier le bootstrap fourni afin de :

- Réserver de l'espace mémoire pour la pile kernel (minimum 1MB = 2^20). Assurez-vous que la pile n'est pas initialisée car nous ne voulons pas qu'elle fasse « gonfler » la taille de l'exécutable.
- Initialiser le pointeur de pile et le pointeur EBP (pointeur de « base » pour le stack frame) ; ceux-ci doivent avoir la même valeur.
- Passer les informations `multiboot_info_t` en provenance de GRUB à votre kernel ; rappel : l'adresse de la structure est placée dans le registre EBX par GRUB.
- Appeler votre kernel ; il s'agit simplement d'appeler la fonction qui est le point d'entrée de votre code kernel en C.

Avant de commencer à écrire le kernel, vous pouvez vérifier que le bootstrap est correctement chargé et exécuté par GRUB par exemple en écrivant des caractères dans la RAM vidéo de la carte VGA (mode texte vu en cours). Si rien n'est affiché, cela signifie que GRUB n'a pas chargé votre bootstrap ou alors que votre code d'affichage est incorrect.

Un premier kernel en C

Cette section décrit les objectifs du kernel à réaliser. Les sections qui suivent détaillent les différentes étapes pour y parvenir.

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

Le kernel à développer doit satisfaire aux quatre objectifs suivants :

1. Initialiser la table des descripteurs globaux (table GDT, cf. cours).
2. Récupérer la taille de la mémoire vive disponible (champ `mem_upper` de la structure d'informations multiboot initialisée par GRUB) et afficher la taille détectée.

3. Offrir un mode de fonctionnement « normal » affichant, au minimum :
    - Un message disant que l'affichage est initialisé.
    - Un message indiquant que la GDT est initialisée.
4. Offrir un mode de fonctionnement « test » :
    - Exécuter une batterie de tests de votre choix validant, aussi exhaustivement que possible, les routines d'affichage implémentées. Ces tests doivent pouvoir être validés à l'œil : le but est de pouvoir construire dessus pour la suite.

A noter que pour l'affichage, vous implémenterez les routines décrites dans la suite de l'énoncé (cf. plus loin).

Le mode de fonctionnement décrit ci-dessus (« normal » ou « test ») est 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. Par soucis de simplicité, nous partons du principe que l'utilisateur effectuera un `make clean` entre la compilation de modes différents (typiquement de « test » à « normal » ou inversement).

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

Par exemple, soit :

```
#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`, nous obtenons :

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

En compilant ce code sans l'argument -DXXX, nous obtenons :

```
$ 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 d'en comprendre les points importants, puis mettez en place une table de descripteurs globaux contenant les trois descripteurs suivants, dans l'ordre (l'ordre est très important cf. slides de théorie) :

- 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` qu'il vous faudra implémenter (cf. section suivante).

Fonctions de base

Les fonctions présentées ici sont des fonctions de base utilisées sur presque tout 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. N'hésitez pas à en implémenter d'autres si le besoin s'en fait sentir.

Fonctions d'accès aux périphériques (PIO)

Depuis votre kernel, vous allez rapidement avoir besoin d'accéder à des périphériques. Par exemple, pour gérer le curseur, lire des secteurs sur disque, initialiser le mode graphique VGA, etc. Pour ce faire, vous devrez utiliser l'adressage mémoire PIO grâce à l'utilisation des « ports » (cf. chapitre de cours « Ports_et_mode_texte_VGA »). Ce type d'accès peut uniquement être réalisé en assembleur avec l'utilisation des instructions `IN` et `OUT`.

Il est donc nécessaire que vous développiez les fonctions suivantes, appelables depuis 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 16-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);`

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, il est conseillé de post-fixer tout fichier source assembleur avec le postfix `_asm.s` (comme cela est fait pour le fichier source `gdt_asm.s`). En effet, cela permet d'obtenir des fichiers objets différents au cas où un fichier C et un fichier assembleur porteraient le même nom (par exemple `gdt.c` et `gdt.s`).

Fonctions d'affichage

Le but des fonctions d'affichage est d'afficher du texte, contrôler la position du curseur et surtout,

gérer le défilement du texte lorsque cela est nécessaire.

Votre OS devra au minimum implémenter les fonctionnalités 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 l'affichage d'un caractère à la position du curseur.
- Fonction permettant l'affichage d'une chaîne de caractères à la position du curseur.
- Fonction permettant de déplacer le curseur à une position 2D de l'écran (x,y).
- Fonction permettant de lire la position 2D du curseur (x,y).
- Fonction d'affichage à arguments variables, similaire à `printf` ; les formats suivants doivent au minimum être gérés :%c,%s,%d et %x. A vous de voir comment accéder aux paramètres sans utiliser le header `stdarg.h` (indice : comment sont passés les arguments d'une fonction ?).

Make et processus de 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. De même, si un fichier header est modifié, seulement le code qui en dépend doit être recompilé.
- Exécuter `make` dans le répertoire racine de votre projet doit générer l'image ISO de votre OS (par exemple `mykernel.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 `mykernel.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 effacer les fichiers finaux ainsi que les fichiers intermédiaires générés (il restera seulement les fichiers sources et de configuration).
- Le répertoire `grub` doit contenir uniquement les fichiers `menu.lst` et `stage2_eltorito`
- Lors de l'effacement de fichiers, pensez à passer l'option `-f` à `rm` afin d'éviter les messages d'erreur en cas d'effacement de fichiers non existants.

Compilation

- Le code C de votre kernel ne doit pas être compilé comme du code C classique car il ne doit pas dépendre de la librairie C et être du code 64-bits. Compilez-le donc avec les options : `-m32 -ffreestanding -nostdlib`
- Compilez aussi votre code avec les options `-Wall` et `-Wextra` et assurez-vous que celui-ci compile sans *warnings* !

Edition des liens

- Lors de l'édition des liens pour obtenir le kernel final, il est nécessaire de spécifier le fichier de script `kernel.ld` au linker avec l'option `-T`
- Plutôt que d'utiliser ld pour l'édition des liens, il est conseillé d'utiliser gcc car la librairie gcc fourni des fonctions utiles qu'il faudrait alors implémenter nous-même (division sur des long signés et non-signés, modulo, etc.).
- N'oubliez pas de générer un exécutable lié statiquement.
- 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` :
```
gcc -static -Tkernel.ld -m32 -ffreestanding -nostdlib
kernel.o gdt_asm.o -o kernel.elf -lgcc
```

Remarques importantes

- Rappelez-vous que vous n'avez pas de librairie C disponible ! Cela signifie qu'aucune des fonctions habituelles (`malloc`, `printf`, etc.) n'est disponible ! De même, vous ne pouvez pas réaliser d'allocation dynamique (à moins d'implémenter `malloc`).
- Pensez à écrire du code modulaire, aussi bien au niveau des fichiers, que du code divisé en fonctions réalisant chacune une tâche bien spécifique.
  - Typiquement, divisez l'écriture du kernel en modules : module pour la gestion de la GDT, gestion de l'affichage, accès aux périphériques (PIO), mini librairie C utilisée par le kernel, etc.
- A pondérer : que se passe-t-il lorsque vous sortez de la fonction `main` ?
- Toutes fonctions et variables *privées* à un fichier source doivent être déclarées avec le mot-clé `static` (autrement les symboles sont globaux, donc visibles par le linker, ce qui peut engendrer des conflits de noms).
- Toute fonction globale (aussi appelée *publique*, c'est-à-dire visible par le linker, donc depuis un autre module) 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 qu'en assembleur.
- A noter qu'en C, tout symbole global (fonction ou variable globale) possède une visibilité *publique* par défaut, alors qu'en assembleur tout symbole possède une visibilité *privée* par défaut.
- Enfin, une fois une fonctionnalité implémentée, n'oubliez pas de supprimez les commentaires TODO correspondants et disséminés dans le code car ils n'ont plus de raison d'être.

Travail à rendre

Vous me rendrez , sur Cyberlearn, l'arborescence complète de votre projet selon les formalités de rendu décrites dans le document « *Consignes pour les travaux pratiques* » sur la page du cours.

Le code devra respecter les consignes décrites dans le document « *Consignes pour l'écriture du code* » se trouvant sur la même page.

La date de rendu est le dimanche 5 novembre.