Virtualisation

Professeur : Florent Gluck

March 15, 2023

Mini hyperviseur avec KVM

Introduction

Dans ce travail pratique, vous programmerez vous même un mini hyperviseur utilisant l'API KVM de Linux. L'hyperviseur à implémenter est volontairement extrêmement rudimentaire pour des raisons de temps. Le but n'est pas d'émuler une plateforme matérielle existante complète comme un PC (à l'image de ce que réalise QEMU), mais seulement quelques aspects de celui-ci.

Objectif

L'objectif de ce travail pratique est de comprendre, de manière approfondie, le fonctionnement d'un hyperviseur, à savoir :

- la création de machine virtuelle (VM)
- l'exécution d'un vCPU, notamment les mécanismes de `VMexit` et `VMentry`
- la *full virtualization* de périphérique grâce à l'émulation
- la paravirtualisation de périphérique
- le mécanisme d'hypercall
- le gain de simplicité et de performance de la paravirtualisation de périphérique par rapport à l'emulation

Pour atteindre cet objectif, vous réaliserez l'émulation de trois périphériques d'un PC :

- affichage graphique
- contrôleur de disque
- disque dur

Cette émulation sera partielle, mais suffisante pour obtenir une VM fonctionnelle. Vous verrez aussi que de réaliser une émulation, même partielle, n'est pas trivial.

Cahier des charges

Les différents points à réaliser sont décrits ci-dessous :

1. Emulation partielle du mode texte VGA de 80 colonnes par 25 lignes en 16 couleurs :
    - L'affichage texte VGA à émuler est décrit précisément dans les slides `resources/VGA_text_mode.pdf`
    - L'OS guest doit pouvoir écrire des caractères à l'écran, à la position et à la couleur désirée
    - L'affichage produit dans une fenêtre graphique par l'hyperviseur doit être identique à celui obtenu sur un PC réel, que le guest écrive dans le framebuffer sur 8, 16, ou 32 bits
        - l'affichage doit être identique à celui produit par QEMU dans le projet `resources/bare_metal_kernel`
    - L'affichage doit être "rafraîchi" à une fréquence de 60Hz par un thread dédié

- Le code permettant de créer une fenêtre graphique dans laquelle afficher des pixels en 24BPP vous est fourni dans `vmm/gfx.c` et `vmm/gfx.h`
- La palette de 16 couleurs utilisées (par défaut) par le mode texte VGA se trouve dans `resources/VGA_16cols_pal.png` ; utilisez le programme `gimp` pour inspecter les composantes RGB de chacune des couleurs afin de les reproduire dans votre code

2. Emulation et paravirtualisation partielle du contrôleur de disque ATA (IDE) :

   - Écriture d'un secteur en mode PIO sur le premier disque
   - On considère qu'un secteur possède toujours une taille de 512 bytes
   - On considère que l'indice du secteur à écrire est représenté sur 28-bits (2^28 secteurs aux maximum) et le premier secteur se trouve à l'indice 0
   - on considère qu'il n'y a qu'un seul disque
   - L'écriture de plusieurs secteurs consécutifs n'a pas à être gérée

3. Points spécifiques à l'écriture d'un secteur en mode emulé (*full virtualization*) :

   - Le code du driver permettant d'écrire un secteur dans l'OS guest se trouve dans le fichier `template/guest/shared/ide_emul.c`
   - L'émulation de l'écriture d'un secteur nécessite d'implémenter une machine à états, similairement à ce qui est expliqué dans le cours

4. Points spécifiques à l'écriture d'un secteur en mode paravirtualisé :

   - La fonction à implémenter `ide_write_sector_pv` est donnée dans `ide_pv.c`, mais elle est vide, donc à vous de l'écrire
   - Vous devez implémenter la partie *frontend* manquante dans le guest sous forme d'un hypercall et la partie *backend* correspondante dans l'hyperviseur
   - L'écriture sur un port (PMIO) de votre choix par le guest doit notifier l'hyperviseur d'un hypercall
   - Le mécanisme et protocole pour transférer les arguments et la valeur de retour entre hyperviseur et guest est libre
   - Pensez à écrire une fonction générique (p.ex. `hypercall`) permettant de réaliser des hypercalls
     - le but est de pouvoir facilement ajouter un hypercall sans avoir à changer l'implémentation de la fonction `hypercall`
     - si l'implémentation de la fonction doit changer lorsqu'un nouvel hypercall est ajouté, alors votre méthode n'est pas suffisamment générique

5. L'hyperviseur, nommé `vmm`, doit être capable de gérer les arguments suivants, passés sur la ligne de commande :

```
vmm -guest BIN -disk FILE

BIN is a file that contains the guest OS to load and execute.

FILE is a file storing the content of the disk the VM must expose. It's simply a
     raw disk image and its size must be a multiple of 512 bytes (sector size).
```

6. L'exécution de la VM doit pouvoir se terminer proprement via la pression sur une touche, p.ex. "ESC".

   - ceci doit typiquement être le cas lors des tests d'affichage VGA où le guest exécute une boucle infinie et ne se termine donc jamais.

7. Assurez-vous que votre hyperviseur passe tous les tests appelés dans le `Makefile` racine (cible `test_all`).

   - La Figure 1 illustre l'affichage graphique que vous devriez obtenir
   - Pour les tests portant sur l'émulation d'écriture de secteurs, vous devriez obtenir l'affichage ci-dessous en cas de succès :

```
vmm/vmm -guest guest/test_disk_emul.bin -disk disk.raw
```

```
Tests passed?
diff disk.raw tests/disk_ref.raw
Tests passed :-)
```

- Pour les tests portant sur la paravirtualisation d'écriture de secteurs, vous devriez obtenir l'affichage ci-dessous en cas de succès :

```
vmm/vmm -guest guest/test_disk_pv.bin -disk disk.raw
Tests passed?
diff disk.raw tests/disk_ref.raw
Tests passed :-)
```

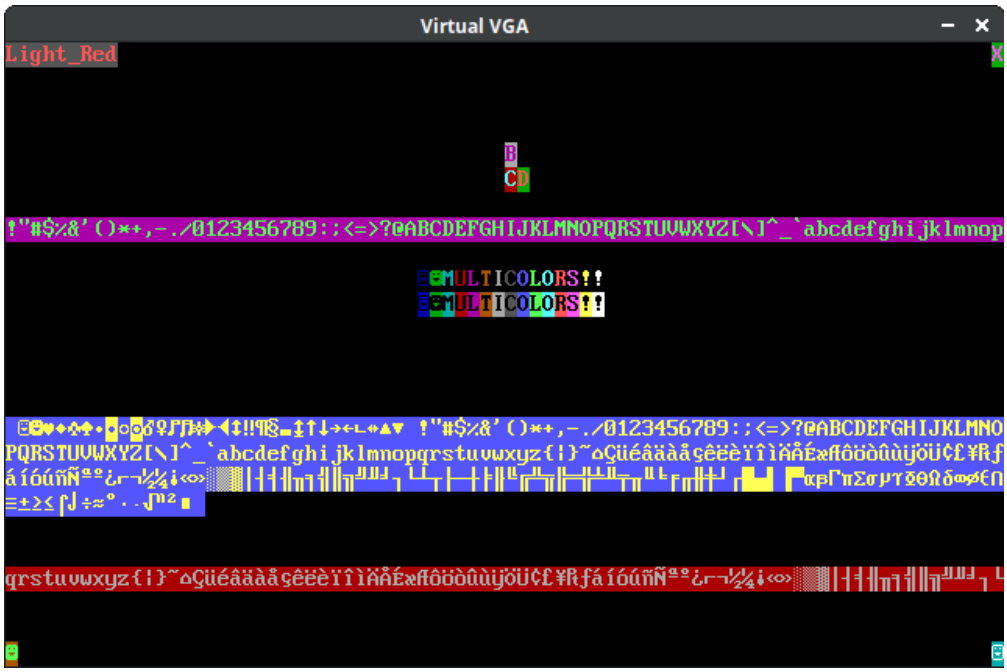

Figure 1: Affichage graphique attendu

Pour les curieu-x-ses et/ou avancé-e-s :

- Réalisez des mesures de performances (benchmark), p.ex. écritures de 10'000 secteurs, afin de comparez les performances obtenues entre l'écriture de secteur émulée et paravirtualisée.
  - ajoutez la ou les cibles permettant de réaliser ces benchmarks au `Makefile` racine afin que cela soit facilement testable
  - quel gain de performance obtenez-vous grâce à la paravirtualization (par rapport à l'émulation) ?
  - pensez à compiler avec l'option `-O3` de gcc afin d'activer les optimisations de code les plus agressives

Code source à disposition

Généralités

Afin de vous donner une base sur laquelle démarrer, le template ci-dessous vous est fourni sur le git du cours, dans le répertoire `template` :

```
|-- guest
|   |-- Makefile
|   |-- shared
|   |   |-- entrypoint_asm.s
|   |   |-- guest.ld
|   |   |-- ide_emul.c
|   |   |-- ide.h
|   |   |-- ide_pv.c
```

```
|   |   |-- pmio_asm.s
|   |   |-- pmio.h
|   |   |-- test_disk.c
|   |   |-- test_disk.h
|   |   |-- utils.c
|   |   |-- utils.h
|   |   |-- vga_emul.c
|   |   `-- vga.h
|   |-- test_disk_emul.c
|   |-- test_disk_pv.c
|   `-- test_vga_emul.c
|-- Makefile
|-- tests
|   `-- disk_ref.raw
`-- vmm
    |-- font.c
    |-- font.h
    |-- gfx.c
    |-- gfx.h
    |-- Makefile
    `-- vmm.c
```

Le code fourni comporte l'ébauche d'un hyperviseur dans le répertoire `vmm`, ainsi que trois OS guest différents, chacun réalisant une fonction bien précise :

- `test_vga_emul` : OS guest réalisant un affichage en mode texte VGA
- `test_disk_emul` : OS guest réalisant des écritures de secteurs à l'aide d'un driver pour un vrai contrôleur de disque ATA (IDE)
- `test_disk_pv` : OS guest réalisant des écritures de secteurs à l'aide d'un driver paravirtualisé pour un contrôleur de disque virtuel ("inexistant")

A la racine se trouve un `Makefile` définissant les cibles ci-dessous qui permettent de valider si l'hyperviseur émule correctement l'affichage et le contrôleur de disque :

```
Available targets:
  test_all       : builds and run all tests
  test_vga_emul  : builds and run the display tests on a guest VM featuring VGA
                   emulation
  test_disk_emul : builds and run regression tests on a guest VM featuring disk
                   emulation
  test_disk_pv   : builds and run regression tests on a guest VM featuring disk
                   paravirtualization
  clean          : deletes all generated files (not the disk though)
```

A savoir que la validation des écritures de secteurs se base sur l'image disque de référence `tests/disk_ref.raw`.

Parmis les fichiers sources fourni, seuls les fichiers sources ci-dessous nécessitent d'être modifiés :

```
Makefile
vmm/Makefile
vmm/vmm.c
guest/shared/ide_pv.c
guest/shared/utils.c
guest/shared/utils.h
```

Le code fourni étant incomplet, il vous faudra très probablement ajouter de nouveaux fichiers sources. Vous êtes 100% libres de faire comme bon vous semble, mais essayez d'être cohérent et de modulariser votre code pour le rendre plus lisible et maintenable.

Code hyperviseur

Le répertoire `vmm` contient l'ébauche d'un hyperviseur :

- `vmm.c` contient l'ébauche de l'hyperviseur. La plupart du code qui permet de créer une VM vous est déjà donné.
- `font.c` et `font.h` table de caractères ASCII étendus correspondant à la table 8-bit Code Page 437 de l'IBM PC d'origine. Chaque caractère est représenté par 16 valeurs de 8 bits où chaque valeur représente une ligne du caractère et chaque bit représente 1 pixel du caractère. Les 16 premières valeurs de la table représentent donc le 1er caractère, les 16 suivantes le deuxième, etc.
- `gfx.c` et `gfx.h` *wrapper* au dessus de la librairie SDL2 permettant de créer une fenêtre et d'y afficher des pixels. Vous pouvez trouver un exemple d'utilisation, `gfx_example.c`, sur le projet gfxlib disponible sur github ici.

Code OS guests

Le répertoire `guest` contient le code utilisés par les différents OS guests faisant chacun appel à des fonctionnalités différentes de l'hyperviseur, à savoir l'affichage VGA en mode texte, l'écriture de secteurs via un contrôleur ATA (IDE) réel et l'écriture de secteurs via un périphérique virtuel paravirtualisé :

- `entrypoint_asm.s` code assembleur du point d'entrée du noyau ; celui-ci initialise le MMU afin de mapper les adresses virtuelles sur les adresses physiques et d'appeler la fonction `guest_main` qui est le point d'entrée de l'OS guest en C (en réalité un noyau extrêmement primitif).
- `guest.ld` fichier *linker* permettant de générer un exécutable "plat" (à contrario d'un fichier ELF) où on s'assure que le début du binaire commence par la section `entrypoint` définie dans `entrypoint_asm.s`.
- `vga_emul.c` code d'affichage d'un caractère en mode texte VGA pour une carte VGA réelle.
- `vga.h` constantes liées à l'affichage en mode texte VGA et prototypes des fonctions d'affichage de caractère.
- `ide_emul.c` code d'écriture d'un secteur pour un contrôleur IDE réel.
- `ide_pv.c` code d'écriture d'un secteur paravirtualisé.
- `ide.h` taille d'un secteur et prototypes des fonctions d'écriture de secteur.
- `test_disk.c` code réalisant les tests d'écriture de secteurs.
- `test_disk.h` header pour `test_disk.c`
- `pmio_asm.s` et `pmio.h` code assembleur, appelable depuis C, permettant d'écrire sur les ports.
- `utils.c` et `utils.h` routines utilitaires, pour l'instant la fonction `memset` uniquement.
- `test_vga_emul.c` OS guest qui réalise le test d'émulation de l'affichage VGA.
- `test_disk_emul.c` OS guest qui réalise le test d'émulation d'écritures de secteurs.
- `test_disk_pv.c` OS guest qui réalise le test d'écritures de secteurs paravirtualisées.

Validation de l'affichage texte VGA avec QEMU

Nous vous fournissons également le code source d'un mini noyau *bare-metal* pour Intel IA-32 qui réalise exactement le même affichage en mode texte VGA que celui réalisé dans les tests de l'OS guest plus haut. La différence est qu'ici le noyau est exécuté par QEMU qui émule un vrai PC et vous pouvez ainsi voir à quoi devrait ressembler l'affichage par votre hyperviseur.

Le code source du noyau *bare-metal* réalisant l'affichage de test se trouve dans `resources/bare_metal_kernel`. Tapez `make run` pour le compiler et l'exécuter dans QEMU. Voici l'arborescence des sources du noyau :

```
resources/
`-- bare_metal_kernel
    |-- grub
    |   `-- grub.cfg
    |-- kernel
    |   |-- bootstrap_asm.s
    |   |-- kernel.c
    |   |-- kernel.ld
```

```
|   |-- Makefile
|   |-- types.h
|   `-- vga.h
`-- Makefile
```

Ressources

Les slides `resources/VGA_text_mode.pdf` décrivent la structure mémoire et la programmation du mode texte VGA.

Echéance

Ce travail pratique est à terminer pour le dimanche 2 avril.