Architecture des systèmes d'exploitation n7OS

Evann DREUMONT

Département SN
Deuxième année - ASR

2024 - 2025

Table des matières

1 Introduction

Ce document a pour but de présenter l'implémentation de mon kernel au cours du projet d'Architecture des systèmes d'exploitation.

2 Console

Dans un premier temps, nous avons implémenté la console. Pour ce faire, nous avons décidé d'utiliser une console qui s'appuie sur la position du curseur. Celle-ci étant sauvegardé dans une seule variable curseur et nous avons utilisé une division / modulo pour retrouver la ligne et la colonne.

Afin de tester cette implémentation, nous avons réalisé des affichages sur la console à l'aide la fonction `console_putbytes` puis `printf`.

3 Pagination

Pour réaliser la pagination, nous avons dans un premier temps implémenté `mem.c`. Dans ce premier module, nous avons décidé d'utiliser une bitmap pour référencer quels plages mémoire étaient utilisé. Puis, nous avons pu implémenter les méthodes `init_paging` et `alloc_page_entry` qui nous permet d'initialiser le paging et d'allouer l'espace mémoire kernel.

Pour tester, nous avons utiliser la méthode `print_mem` du module mem.c qui permet d'afficher l'état des pages mémoire à un instant t. Puis, nous avons tenté d'allouer une variable à une très grande adresse :

```
alloc_page_entry(0xa000FFF8, 1, 1);
uint32_t *ptr = (uint32_t *)0xa000FFFc;
uint32_t test = *ptr;
test++;
print_mem();
```

Le fait que nous n'ayons pas de pagefault lorsque la première instruction `alloc_page_entry` est présente, nous a permis de dire que notre implémentation était fonctionnelle.

4 Interruptions

Ensuite, nous nous sommes attelés à l'implémentation du support des interruptions. Ici, pas de choix spécifique mis à part que nous manipulons des structs plutot que la variable en binaire directement pour simplifier et augmenter la lisibilité du code. Autrement, nous avons suivi les instructions fournies.

Afin de tester notre implémentation, nous avons implémenté une interruption de test (Interruption 50). Puis tentons de l'exécuter, en utilisant l'instruction `__asm__("int $50");`.

```
init_syscall();

__asm__("int $50");

if (example() == 1)
{
    printf("Appel systeme example ok \n");
}
```

Comme nous observons l'affichage `Appel systeme example ok` sur la console, nous pouvons donc dire que notre implémentation des interruptions est fonctionnel.

5 Timer

Une fois les interruptions implémentée, nous avons décidé de nous intéresser au timer. Ici, pas d'implémentation particulière. Nous avons implémenté l'interruption et configuré le PIT via le PIC pour appeler notre fonction `tick()`.

Nous avons également modifié la console pour pouvoir afficher l'heure sur le header. Pour ce faire, nous avons implémenté la fonction `update_clock` qui update le header, que nous appelons dans la fonction tick pour mettre à jour la valeur de la date au rafraichissement au changement du timestamp.

Enfin, pour pimenter le tout, nous avons ajouté une fonction `rtc_sync` (en s'appuyant de la page wiki d'OSDev), ce qui permet de synchroniser le timestamp avec la valeur de la RTC permettant d'afficher l'heure réel. dans notre header.

Ici pas de test particulier, l'affichage dans le header étant suffisant pour valider le fonctionnement de notre timer.

6 Appels systèmes

Pour les appels système, nous avons implémenté le syscall `shutdown`. Ici rien de particulier au niveau de l'implémentation.

Pour les tests, nous avons fait un appel à la fonction `shutdown(1)` et vérifier que la VM s'éteignait bien !

7 Clavier

Pour la mise en place du clavier, nous avons dans un premier temps mis en place l'appel système qui va bien. Puis, nous avons activé l'interruption du clavier via le PIC. Pour tester, nous avons utilisé le bloc suivant :

```
while(1)
{
    char c = kgetch();
    printf("%c", c);
}
```

Celui-ci nous permettait de taper au clavier et de voir le résultat s'afficher sur l'écran.

8 Processus

Pour finir, nous avons implémenté les processus. Ici encore, nous avons choisi d'utiliser un struct à la place de la valeur binaire directement pour simplifier la lisibilité.

Pour l'implémentation des processus, nous avons fait le choix d'utiliser une table de processus de taille fixé. Le PID de chacun des processus correspond à l'index du processus dans la table. Afin de déterminer si un processus existe ou non, nous avons utilisé une bitmap. Ce choix, bien que non optimal, nous permet une implémentation rapide et simple de la détection des process dans la table des processus.

Pour ce qui est de l'ordonnancement, nous avons mis en place un round robin avec des quanta de temps de 10ms. Pour déterminer le prochain processus, nous nous bâson sur la bitmap pour savoir si le processus existe, et nous regardons son état pour voir si le processus est pret. Une fois que nous avons trouvé un processus, nous faisons un context switch. Cette implémentation ne prend pas en compte les priorité, et ne permet pas d'intercaller des processus. Encore une fois, ce choix a été fait pour simplifier l'implémentation aux dépens des performances du système.

L'ordonnancement est appelé via la fonction `schedule` tous les 10 ticks.

Pour tester les processus, nous avons utilisé la fonction `spawn_process` qui permet de lancer un processus. Ces processus de test affichaient un message différent, ce qui nous a permis de voir que l'ordonnancement rand-robin est bien fonctionnelle.

9 Améliorations personnelles

Ensuite, nous avons décidé d'implémenter un minishell. Celui-ci permet d'accéder aux différentes fonctions que nous avons définies au préalable. Il permet, entre autre, d'utiliser le clavier,

d'éteindre l'ordinateur. Nous pouvons aussi interagir avec les processus : les bloquers, les débloquers, les lister et les terminer (le kill est un kill -9).

10 Conclusion

Ce fut un projet fort sympathique, et qui nous a permis d'apprendre énormément sur le kernel et le fonctionnement d'unix. Merci pour ce projet.