Pi-kachULM_OS

1 Présentation

1.1 Objectif

Nous nous sommes fixés comme objectif premier la réalisation d'un système d'exploitation en C++ s'exécutant sur une *Raspberry Pi 3B+*. Ce choix était principalement motivé par le support de cette plateforme par l'émulateur *QEMU*.

Finalement, par curiosité et pour le fun™, nous av ons finalement choisi de supporter plusieurs modèle de *Raspberry Pi* :

— *Raspberry Pi 3B+*
— *Raspberry Pi 4 Model B*
— *Raspberry Pi Compute Module 4*

La totalité de ce qu'il suit à donc été testé et conçu afin d'être exécuté sur l'un de ces modèles. Ce projet compte environ XXX lignes de codes de C++, xx lignes de C et xx lignes d'assembleur. La totalité du code est disponible à l'adresse suivante : `https://github.com/hgruniaux/Pi-kachULM_OS`

1.2 Reconnaissance dynamique du matériel

Afin de supporter plusieurs machines, qui ne partagent pas toutes les mêmes caractéristiques et spécificités nous avons besoin d'un moyen de les déterminer dynamiquement, lors du démarrage du noyau. Pour ce faire, le noyau utilise et le *DeviceTree* donné par le *bootloader*. Nous avons donc développé une bibliothèque, `libdevicetree` permettant d'interpreter cette structure.

Un *DeviceTree* est, comme son nom l'indique, un arbre, compilé sous la forme d'un fichier binaire, qui est chargé par le *bootloader* lors du démarrage. Cet arbre est rempli par ce dernier, et informe le noyau du matériel présent. Une documentation est disponible sur cette page.

2 Modèle mémoire

2.1 Vue de la mémoire

Lors du démarrage du système d'exploitation, ce dernier récupère la taille de la mémoire disponible et s'occupe d'initialiser le *Memory Management Unit (MMU)* pour que le noyau puisse s'abstraire de la mémoire physique. Cette configuration est effectuée au sein du fichier `mmu_init.cpp`. Les pages ont une taille fixée à 4 kio.

On utilise ici à bon escient les spécificités du *MMU* de la plateforme ARM. Cette dernière nous permet d'avoir deux espaces d'adressage complètement distincts :

— La mémoire des processus est dans l'espace d'adresse `0x0000000000000000` à `0x0000ffffffffffff`.
— La mémoire du noyau est, elle, dans l'espace d'adresse `0xffff000000000000` à `0xffffffffffffffff`.

2.2 Allocateurs de pages

Une fois que le noyau est passé en mémoire virtuelle, il nous faut un moyen de garder une trace des pages physiques libres ou utilisées par le noyau. Pour ce faire, nous avons implémenté deux allocateur de pages :

— `page_alloc.hpp` s'occupe de l'allocation de la grande majorité des pages physiques de la mémoire. Cette allocation est réalisée de manière optimisée par un arbre binaire.
— `contiguous_page_alloc.hpp` travaille sur une petite portion de la mémoire physique et s'occupe d'allouée une liste de pages contiguës. Cette allocation est utilisée pour les *buffers* utilisés lors des transferts DMA.

2.3 Allocateurs de mémoire

À l'aide de l'allocateur de page principal et en modifiant les tables de translation du *MMU* du noyaux

2.4 Système de fichier

Le RamFs et les fonctionnalités (FAT)

3 *Drivers*

3.1 GPIO

3.2 Uart

3.3 Mini Uart

3.4 Mailbox

3.5 Contrôleur IRQ

3.6 Contrôleur DMA

3.7 System Timer

3.8 Timer ARM

4 Processus

4.1 Hiérarchie

Support des processus utilisateurs et noyaux leur `libc`. Pas de diff entre process et thread.

4.2 Appels systèmes

Liste des appels systèmes possibles (sycalls de `libsyscall`)

5 Ordonnanceur

5.1 Algorithme et Priorités

Comment l'ordonnateur fonctionne, qu'est-ce qu'il est capable de faire ?

5.2 Mise en pause de Processus

Comment et quand les processus font dodo ?

6 Écran et Fenêtres

6.1 *Framebuffer*

Rapidement comment on affiche des choses à l'écran.

6.2 Gestionnaire de fenêtres

Fonctionnalités et choix d'implementation (Processus Noyaux).

7 Clavier PS/2

Clavier PS2.

8 Conclusion

Quoi qu'on dit là ?