Système et réseaux : Rendu de projet

Pluvinage Lucas, Fehr Mathieu

May 15, 2017

Abstract

Pour ce projet, nous nous sommes intéressés à l'écriture d'un noyau en C pour Raspberry Pi (1 et 2). L'objectif était à la base de pouvoir accéder à l'OS en réseau, ce qui n'a pas été implémenté au final, faute de temps. Un kernel à quand même été implémenté, possédant de multiples fonctionnalités, comme un user mode, un système de fichiers, et un shell.

Contents

1 Téléchargement et compilation du projet

1.1 Sources et documentation

Les sources du projet sont clonables via github, à l'addresse https://github.com/Nappy-san/WindOS.git.

Une documentation doxygène a été compilé avec le projet, pour le rendre plus lisible.

1.2 Modules utilisés et dépendances

Le git contient deux sous-modules :

- qemu-fvm, qui est un fork de qemu permettant d'émuler le raspberry pi 2
- USPi, qui est une librairie permettant d'utiliser les ports USB du raspberry pi

D'autres dépendances sont nécessaire pour pouvoir faire tourner le kernel :

- genext2fs, qui permet de créer un système de fichier dans notre mémoire ram
- minicom, qui permet de lire le port série
- arm-none-eabi, qui est le compilateur nous permettant de compiler pour le raspberry pi

1.3 Compilation du projet

Pour compiler le projet, il faut lancer "make all", qui compilera le kernel, et les programmes utilisateurs. Ensuite, pour lancer le kernel sur qemu, il suffit d'executer "make runs".

Pour lancer le kernel sur raspberry pi 2 (le kernel ne marchant pas sur raspberry pi 1), il faut copier les fichiers contenus dans le dossier "img", et les copier dans la première partition fat de la carte SD inséré dans le raspberry pi. Pour pouvoir afficher dans un terminal la sortie du raspberry pi, il faut brancher sur les pins connectés au controller uart un cable USBTTL, puis lancer minicom sur un terminal (avec la commande "sudo minicom -c on -b 115200 -D dev/ttyUSB0" si la sortie est située dans ttyUSB0).

2 Modules du kernel

Le kernel contient plusieurs modules intéressants, comme la gestion du serial pour envoyer des données à un ordinateur, l'utilisation du timer ARM, l'utilisation du MMu pour la gestion de la mémoire virtuelle, un système de fichier ext2, et une mailbox permettant de communiquer avec le VideoCore.

2.1 Serial

2.2 Timer

Le timer est très utile pour l'ordonnancement des processus, puisqu'il permet de lancer des interruptions à une fréquence donnée. Nous utilisons le timer de la carte BCM2835/2836. Nous l'avons implémenté à partir de la documentation du BCM2835 ARM Peripherals, à la page 196.

Nous permettons à partir du timer d'éxecuter une fonction de manière décalée (ce qui est nécessaire pour USPi) ou encore de générer des interruptions de manière régulière pour appeler l'ordonnanceur

2.3 Mémoire virtuelle

Le MMU (Memory Management Unit, décrit dans la partie B4 de l'arm ARM (https://www.scss.tcd.ie/ waldroj/3d1/arm_arm.pdf)) est utilisé pour permettre de rediriger des pages virtuelles vers des pages physiques.

Pour cela, nous avons implémenté dans mmu.c des fonctions bas niveau permettant de rediriger des pages de toute tailles via les translation table base (dont le système est très bien expliqué dans l'arm

ARM partie B4.7).

Ces fonctions sont ensuites utilisées par memalloc.c pour avoir une gestion de la mémoire virtuelle, en retenant les pages physiques déjà allouées.

2.4 Système de fichiers

2.5 Mailbox

La mailbox est ce qui permet la communication entre le VideoCore (le GPU) et le processeur. Cela permet au processeur de demander des informations au GPU, ou de demander au GPU d'effectuer des actions, comme alimenter en énergie un périphérique. La mailbox est bien expliqué sur la page https://github.com/raspberrypi/firmware/wiki/Mailboxes, mais reste très mal documentée sinon.

Pour communiquer avec la mailbox, il faut lui envoyer un pointeur sur l' instruction demandée, qui est une structure contenant le tag de l'instruction, ainsi que l'espace mémoire nécessaire pour stocker les données de retour ou d'entrée. Il est possible d'envoyer jusqu'a 8 d'instructions en même temps, mais pour faciliter l'implémentation, et puisque nous utilisons peu la mailbox, nous n'envoyons qu'une instruction à la fois.

3 Gestion de l'User mode

L'user mode est correctement géré par notre noyau, sous un modèle préemptif.

3.1 Gestion de la mémoire virtuelle

La mémoire virtuelle permet d'isoler les processus les uns des autres, ainsi que de leur faire croire qu'ils ont accès à toute la RAM. Nous avons placé le kernel dans la partie haute de la mémoire virtuelle (de 0x80000000 à 0xFFFFFFFF), pour y avoir accès directement lors des interruptions, sans avoir à changer la tables de translations. Grace aux fonctions définies dans memalloc, le kernel peut récupérer facilement les pages physiques libres disponibles, pour les affecter aux processus demandant de la mémoire.

3.2 Ordonnancement des processus

L'ordonnancement des processus se fait de manière assez simple. On fixe le nombre maximum de processus pouvant être éxecutés en parallèle. Les ID que l'on donne aux processus existants correspond à leur addresse dans un tableau contenant tout les processus (existants ou "libres"). On a de plus un tableau listant tout les id libres, et un contenant tout les id correspondant à des processus actifs (et qui de plus ne sont pas dans un wait), et un tableau contenant les processus zombies.

Pour récupérer les processus à executer, il suffit donc de parcourir la liste des processus actif de gauche à droite, puis recommencer.

3.3 Appels systèmes

4 Séquence de boot

La séquence de boot d'un raspberry pi n'est pas la même que pour un ordinateur classique. C'est le GPU qui démarre le CPU, et qui facilite la tache du boot.

4.1 Lancement du code via le GPU

Pour booter le raspberry pi, le GPU cherche, dans la première partition FAT de la carte SD, un fichier nommé bootcode.bin (que l'on à copié de la distribution linux), qui va lui-même lire le fichier start.elf (copié aussi de la distribution linux), qui va s'occuper de lire le fichier CONFIG.TXT, pour ensuite trouver le fichier image à éxecuter, qui est "kernel.img" dans notre noyau. C'est donc l'image kernel.img que nous devons compiler pour le noyau. La première instruction qui va être executée est à l'addresse 0x8000 pour le raspberry pi, et 0x10000 pour QEMU.

4.2 Mise en place des interruptions et du MMU

La première chose que nous faisons lors du boot est de vérifier que nous somme bien en mode superviseur (car le raspberry pi 2 démarre en hyperviseur, alors que QEMU démarre en superviseur). Ensuite, nous enregistrons la table d'interruptions à l'addresse 0, puis nous initialisons les stacks pour les différents modes, puis le MMU. Nous laissons la première page virtuelle pointer sur la première page physique, afin de pouvoir sauter vers le début du code C du kernel, situé dans la moitié haute de la RAM.

4.3 Initialisation des modules et du mode utilisateur

Une fois dans le code C, la première étape est d'initialiser les modules (comme le timer, ou les interruptions, ou encore le système de fichiers). Ensuite, on s'occupe de charger en mémoire le programme init. On lance alors le mode utilisateur, en démarrant dans ce programme.

5 Programmes fournis avec l'OS

Une sous section par programme ?

6 Difficultés rencontrées

On plante sur une instruction, 3 instruction avant on a r0=r5, mais quand ça plante on a plus r0=r5
Port série qui dit nope parfois fp wtf parfois hypervisor mode lol bug