Système et réseaux : Rendu de projet

Pluvinage Lucas, Fehr Mathieu

May 16, 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 finalement pas été implémenté, faute de temps. Le noyau implémenté dispose cependant de multiples fonctionnalités similaires à un noyau UNIX. Ainsi nous disposons d'un système d'exploitation préemptif, avec notamment un mode utilisateur, un système de mémoire virtuelle et un système de fichier.

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 Doxygen a été compilé avec le projet, pour le rendre plus lisible. Il se situe dans le dossier doc/.

1.2 Modules utilisés et dépendances

Le dépôt contient un sous-module qemu-fvm, qui est un fork de qemu permettant d'émuler le raspberry pi 2.

D'autres dépendances sont nécessaire pour pouvoir faire tourner l'OS:

- genext2fs, qui permet de créer un système de fichier qui sera chargé en RAM.
- minicom, qui permet de lire le port série.
- arm-none-eabi-gcc, qui est le compilateur croisé nous permettant de compiler pour l'architecture ARM.

1.3 Compilation du projet

Pour récupérer les submodules, il faut utiliser la commande `git submodule init` puis`git submodule update`. Pour configurer QEMU, il faut utiliser les commandes `./configure --target-list=arm-softmmu` et `make` dans qemu-fvm.

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 port série pour envoyer des données à un ordinateur, l'utilisation du timer ARM, l'utilisation du MMU pour la gestion de la mémoire virtuelle, une abstraction de système de fichier (VFS) sur lequel deux systèmes de fichiers sont montés, et une mailbox permettant de communiquer avec le VideoCore.

2.1 Serial

Le Raspberry Pi dispose de plusieurs UART (Universal Asynchronous Receiver Transmitter). Nous utilisons le Mini-UART qui est une version simplifiée implémentant UART avec moins d'options à contrôler. Avec un baudrate fixé à 115200 bps, cela nous suffit largement pour le développement d'un terminal. Le mini-UART dispose d'un buffer de lecture et d'écriture de 8 byte, ce qui permet de ne pas avoir à suivre précisément les interruptions levées par ce dernier: on peut se contenter de récupérer les données reçues à chaque tick de timer.

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.

Le timer permet de retarder l'exécution d'une fonction, et surtout de générer les interruptions de contrôle indispensables dans le cadre d'un OS préemptif.

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

Après une tentative infructueuse d'implémenter FAT, c'est EXT2 qui a été utilisé pour représenter les fichiers en mémoire. Au lieu de brancher directement Ext2 aux appels systèmes et autres fonctionnalités noyau, une interface a été développée afin de faire abstraction entre le driver et le reste du kernel.
Ainsi chaque système de fichiers développé implemente des fonctions définies par le VFS (Virtual File System) ce qui permet au VFS de traiter indifféremment chaque système de fichier. Cela permet en plus de pouvoir monter un système de fichier sur un autre, et ainsi accéder à plusieurs systèmes fichiers à partir de l'arborescence.
Deux drivers de systèmes de fichiers ont été implémentés:

- EXT2: Ce système de fichiers, utilisé dans l'écosystème Unix avant d'être remplacé par EXT3/4, utilise une structure à base d'i-node et de blocks.
- DEV: Pour représenter les différentes interfaces disponibles dans l'OS, (inspiré de /dev chez Unix), un pseudo-système de fichier a été implémenté. Il permet par exemple d'accéder au port série (/dev/serial).

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. Un timer déclenche des interruptions à intervalle régulier, permettant au noyau de reprendre la main sur les processus et d'exécuter l'ordonnanceur.

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.
Le mapping choisi est le suivant:

- 0x00000000 - 0x80000000: Espace utilisateur.
- 0x80000000 - 0xbf000000: Mapping linéaire vers la mémoire physique.
- 0xbf000000 - 0xc0000000: Mapping vers les addresses de périphériques.
- 0xc0000000 - 0xf0000000: Tas (heap) du noyau alloué dynamiquement.
- 0xf0000000 - 0xffffffff: Code et segment de données noyau.

3.2 Bibliothèque standard

Le système disposant d'un MMU, les programmes sont compilables avec un nombre minimal d'options, à condition d'utiliser arm-none-eabi-gcc. Cette toolchain est compilée avec Newlib, une bibliothèque de développement bare metal implémentant en partie la libc et demandant un nombre minimal d'appels systèmes.
Ainsi pour communiquer avec le noyau, les programmes disposent d'un certain nombre d'appels systèmes Linux, décrits dans include/syscalls.h. Des fonctionnalités de terminal furent aussi développées, elles sont décrites dans include/termfeatures.h.

3.3 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.

3.4 Appels systèmes

L'objectif initial était de faire un système s'apparentant à un noyau Linux. Ainsi les appels systèmes développés suivent les mêmes spécifications. L'encodage des erreurs se fait via la variable errno, que l'on transmet au processus en renvoyant -ernno, une valeur négative étant signe d'erreur, ce qui est décodé dans la bibliothèque utilisateur.

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 tâche 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 (firmware fourni par l'organisation Raspberry Pi), qui va lui-même lire le fichier start.elf, qui va s'occuper de lire le fichier CONFIG.TXT, pour ensuite trouver le fichier image à exécuter, qui est `kernel.img` dans notre cas. 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

- cat: concatène les fichiers passés en paramètres et affiche le résultat sur stdout.
- cp: copie un fichier ou un dossier vers une destination.
- init: initialise le premier shell, et récupère les processus zombies.
- ls: liste les entrées d'un dossier. (-i pour afficher les inodes, -a pour afficher les dossiers cachés, -l pour afficher en ligne)
- mkdir: crée un dossier dans le répertoire courant.
- pico: un éditeur de fichiers minimaliste.
- pwd: affiche le dossier courant.
- rm: supprime un fichier ou un dossier en mode récursif (-r)
- show: affiche un fichier au format bitmap (base résolution de préférence).
- touch: créer un fichier.
- wesh: WindOS Experimental Shell, l'invite de commande lancé au démarrage de l'OS.

6 Difficultés rencontrées

6.1 De la simulation à la réalité

La plus grande difficulté que nous avons rencontré est sans conteste le passage de la simulation à la réalité. Le kernel plante toujours de manière aléatoire sur le raspberry pi 2, et ne termine pas de loader sur raspberry pi 1.

De plus, QEMU n'émule pas exactement le raspberry pi, ce qui fait que certains programmes marchent sur QEMU, tandis qu'ils ne marche absolument pas sur le raspberry pi. Par exemple, QEMU démarre le raspberry pi en mode superviseur, alors que le raspberry pi démarre en mode hyperviseur (il nous a fallu beaucoup de temps pour remarquer ça, puisque c'était très mal documenté).

6.2 Un gros manque de documentation

Un autre problème était le manque flagrant de documentation pour raspberry pi 2. Tandis qu'il y a beaucoup de documentations sur la carte BCM2835 pour le raspberry pi 1, il n'y a que 20 pauvres pages pour le BCM2836. En général, les instructions processeurs étaient bien documentés, mais les périphériques l'étaient très mal (par exemple, nous n'avons toujours pas trouvé les pins du GPIO permettant d'alimenter la carte SD, via le controleur EMMC).

6.3 Des bugs non résolus sur le RPi2

Il reste encore deux grands bugs qui font que le kernel est inutilisable sur le raspberry pi 2, mais qui marche sur QEMU :

Le premier vient du fait que le port série décide parfois de ne pas s'initialiser pour une raison inconnue, et plante au démarrage. Relancer le noyau règle le problème.

Le second est plus cryptique. Le frame pointer (fp ou r12) possède parfois une valeur abérante, ce qui fait planter le programme utilisateur. On a remarqué en général que des registres prenaient parfois

des valeurs inconnues (comme par exemple, quand on a l'instruction `mov r0,r5`, mais que 3 instructions plus tard, alors qu'il n'y avait pas d'interruptions, r0 != r5).