Systèmes et réseaux : rendu de projet

PiKern, un noyau minimaliste pour Raspberry Pi

Théophile Bastian, Nathanaël Courant

29 mai 2016

Résumé

Au cours de ce projet, nous nous sommes intéressés à l'écriture en C++ d'un noyau minimaliste bootable pour Raspberry Pi. Nous avons réussi à implémenter une gestion de processus distincts avec ordonnanceur, une couche réseau complète gérant le ping et l'UDP, un système de fichiers en mémoire pouvant contenir des fichiers exécutables, un shell distant minimaliste, ainsi que quelques jeux : un serveur snake, et une implantation basique de la Z-machine, permettant de jouer à un grand nombre de jeux d'*interactive fiction*, en particulier une bonne partie des jeux publiés par Infocom.

Code source

Table des matières

1 Vue d'ensemble

1.1 Organisation globale

La partie principale du projet est le noyau lui-même. Celui-ci (dans `kernel/`) se décompose lui-même en plusieurs modules de fonctions différentes, qui seront décrits en détail plus tard dans le rapport. Notons tout de même que certaines parties traitent de communication avec le matériel, d'autres de réseau, d'autres encore de gestion des processus, ...

Afin de gérer l'USB — et donc la carte réseau, branchée au contrôleur USB —, nous avons utilisé USPi, un driver *bare-metal* de bonne qualité trouvé sur Internet. En effet, l'USB des Raspberry Pi est très mal documenté, très difficile à interfacer et encore plus difficile à maintenir, d'après le développeur du driver Linux ; ainsi avons-nous préféré utiliser du code déjà existant (le seul que nous ayons utilisé dans ce projet) pour cela.

Enfin, notre système supporte l'exécution de fichiers exécutables (compilés avec des options particulières et décorés) chargés depuis un système de fichiers en mémoire. Ceux-ci représentent une partie importante du projet en termes de fonctionnalités : le shell distant, par exemple, est un programme indépendant du noyau (interfacé avec celui-ci *via* une bibliothèque).

1.2 Organisation du code

Tout le code source du noyau peut être trouvé sur Github ici, versionné.
Le code du noyau (dossier `kernel/src`) est organisé dans les fichiers suivants :

- `arp` : gestion des paquets ARP (découverte d'adresse MAC) ;
- `assert` : gestion des assertions ;
- `atomic` : opérations atomiques (mutex, ...) ;
- `barriers` : barrières mémoire et d'instructions ;
- `Bytes` : classe munie d'opérateurs << et >> pour gérer des octets en big endian, utilisé essentiellement pour le réseau ;
- `cinit` : fonction sur laquelle saute le point d'entrée (écrit en assembleur) après avoir déplacé le stack pointer ;
- `common` : déclarations de types et de fonctions usuelles ;
- `exec_context` : contexte d'exécution d'un programme (stdin, stdout, working directory, ...) ;
- `expArray` : équivalent à `std::vector` ;
- `filesystem` : un RAMFS ;
- `format` : formatteur sur le modèle de sprintf, pour des `Bytes` ;
- `fs_populator` : remplissage initial du RAMFS ;
- `genericSocket` : classe de base pour les sockets (entre applications et réseau) ;
- `gpio` : gestion des GPIOs (pins logiques matériels) ;
- `hardware_constants` : constantes dépendant de la version de Raspberry Pi utilisée ;
- `hashTable` : une table de hachage ;
- `icmp` : gestion des paquets ICMP (actuellement, uniquement réponse au ping) ;
- `init.s` : point d'entrée (assembleur) du programme ;
- `interrupts` : gestion des interruptions ;
- `ipv4` : gestion de la couche IPv4 ;
- `kernel` : fonction principale (main) ;
- `logger` : interface de logs via le réseau ;
- `mailbox` : gestion du protocole *mailbox*, permettant de dialoguer avec le matériel (récupération de l'adresse MAC, de la température des composants, de gérer l'alimentation des composants, ...) ;
- `malloc` : malloc naïf (`sbrk`) ;
- `networkCore` : cœur de gestion du réseau ;
- `pair` : équivalent à `std::pair` ;
- `process` : gestion des processus et des SVC ;
- `queue` : équivalent à `std::queue` ;

— `sleep` : gestion des `sleep`, et plus généralement du temps (timers, ...) ;
— `start_message` : contient une chaîne de caractères envoyée sur le réseau au boot ;
— `startup` : fonctions standard d'initialisation des programmes indépendants, inclus dans `libsys.a` ;
— `svc` : appels SVC (appels système) ;
— `syslib` : header pour les programmes indépendants ;
— `udp` : gestion des paquets UDP ;
— `udpSocket` : socket (héritant de `GenericSocket`) spécialisé pour l'UDP ;
— `udpSysWrite` : structure pour écrire de l'UDP sur le réseau depuis un programme indépendant ;
— `uspi_interface` : interfaçage avec USPi (driver USB).

1.3 Dépendances

Pour fonctionner, le projet dans son état actuel nécessite :
— les outils de cross-compilation `arm-none-eabi` (comprenant `g++`, `gcc`, `ld`, `as`, `ar`, `objdump`, `objcopy` et autres) ;
— Python 3 ;
— une Raspberry Pi B version 1.

1.4 Compilation

La compilation se déroule en plusieurs étapes :

1. compilation de `libsys.a` ;
2. compilation de chaque programme indépendant, décoration de ce fichier comme exécutable pour notre noyau et copie dans le dossier du RAMFS initial ;
3. génération du fichier C++ définissant le contenu du RAMFS initial ;
4. compilation des objets qui ne sont pas encore compilés pour le noyau ;
5. assemblage dans `_build/output.elf` ;
6. copie de l'objet compilé avec les décorations nécessaires en `kernel.img` ;
7. éventuellement (`make install`), copie sur la carte SD.

L'étape d'assemblage utilise un script de linker essentiellement repris de `arm-none-eabi-ld`, à une ou deux modifications près. Chaque compilation vers un objet nécessite un certain nombre d'options, afin de compiler pour la bonne architecture et sans le code d'initialisation, entre autres.

La compilation des programmes indépendants est un peu compliquée également : comme nous n'avons pas utilisé de MMU, nous avons dû produire de l'assembleur *position-independant*, stockant ses variables globales à des endroits adaptés (et non en plein milieu de la pile système, comme ce qui est fait par défaut...), etc.

2 Bas niveau et matériel

2.1 Matériel

Notre noyau a été conçu et testé sur une Raspberry Pi B version 1. Nous avons de plus soudé sur une plaque de prototypage quelques headers de pins auxquels sont reliées huit LEDs identiques pour afficher un octet, une LED d'état et une LED de crash, connectables par GPIO à la Raspberry Pi ; ainsi qu'un interrupteur d'alimentation et les ports USB allant avec, afin d'éviter d'user les connecteurs à brancher et débrancher le câble USB d'alimentation (seule manière prévue d'allumer et d'éteindre la Raspberry Pi).

Le chargement du noyau se faisait en écrivant directement sur la carte SD (qui sert de stockage et de disque de démarrage à la Raspberry Pi) à place du `kernel.img` installé par la distribution standard Linux pour Raspberry Pi, à l'aide de `make install`.

2.2 Séquence de boot

Nous avons conservé le bootloader par défaut de Raspbian pour démarrer notre `kernel.img`. La séquence de boot standard (que nous n'avons donc pas implémentée) démarre tout d'abord le GPU, qui va exécuter le bootloader, charger (par simple `memcpy`) l'image système en RAM, puis démarrer le CPU et lui donner la main.

Celui-ci déplace son PC (Program Counter) à l'adresse `0×8000`, où doit se trouver la fonction de démarrage du noyau, correspondant pour nous au contenu de `init.s`, qui déplace le SP à `0×8000` (la stack croît vers le bas), puis saute sur le `_c_init` (dans `cinit.cpp`) qui écrit le BSS et donne la main à `kernel_main`.

2.3 GPIO

Les pins GPIO se contrôlent aisément à l'aide de constantes à écrire à un endroit précis en mémoire pour en gérer le sens (input ou output) et en écrire l'état.

2.4 Mailbox

Ce protocole permet d'interagir avec les différents composants matériels, comme le GPU (non-implémenté), mais aussi la récupération de l'adresse MAC de la carte réseau, la mise sous tension des composants, les capteurs de température, ...

3 Processus

Pour gérer plusieurs processus s'exécutant en parallèle, on effectue de la simulation séquentielle du parallélisme, en utilisant un seul cœur du Raspberry Pi, qui exécute séquentiellement des morceaux de chaque processus.

Pour faire la transition entre chaque processus, il s'agit de sauvegarder son état (*ie.* l'état de tous ses registres et de son mode uniquement, puisqu'il n'y a pas de MMU). On garde ces informations, ainsi que quelques autres (état du processus, nom, répertoire courant...) dans une table globale.

Les appels système se font par l'intermédiaire des intructions SVC, qui font un appel au superviseur (et en particulier, changent le mode du processeur ; c'est la seule manière de sortir du mode *user*). Il y a un type d'appel SVC pour chaque appel système souhaité ; chacun d'entre eux ayant du code associé dans `process.cpp`.

Les informations retenues pour chaque processus sont :

— l'état des registres et le mode,
— les processus suivants et précédents dans la liste circulaire doublement chaînée des processus,
— le répertoire courant,
— l'état du processus : actif, en train d'attendre de l'entrée-sortie, d'attendre la terminaison d'un autre processus, de dormir, ou zombi, ainsi qu'un entier 64 bits indiquant une information d'état (le moment de réveil pour un processus endormi, par exemple).

Tout cela permet d'avoir des processus s'exécutant en même temps, et pouvant communiquer à travers les sockets existants. Cependant, il y a une difficulté pour exécuter des processus depuis un fichier.

En effet, exécuter un processus depuis un fichier requiert le chargement du fichier en mémoire, et l'exécution de celui-ci. Or, nous n'avons pas de MMU, le fichier ne peut donc pas être facilement placé à n'importe quel endroit de la mémoire. La solution pour cela est de compiler les programmes avec l'option `-fPIC`, qui génère du code indépendant de la position.

Cependant, ce code indépendant de la position a besoin d'une table globale indiquant les positions des données. Pour ce faire, nous ajoutons le décalage convenable aux variables dans la section `.got` (Global Offset Table), ce qui permet d'avoir les décalages voulus. Le format de l'en-tête d'un fichier exécutable est donc le suivant :

— constante magique `"\x7fELF"` (4 octets),

— fin de la section `.bss` (4 octets), permet de savoir la mémoire statique à allouer au processus,
— position de la section `.got` (4 octets),
— longueur (en mots de 4 octets) de la section `.got` (4 octets).

Malgré cela, il semblerait qu'il reste quelques problèmes avec la relocalisation des exécutables : ainsi, les exceptions et l'utilisation de chaînes de caractères globales semblent poser problème. Comme nous n'avions pas énormément de temps, et que le format des fichiers produits par `gcc` est complexe, nous n'avons pas cherché à corriger ces problèmes (ce qu'il faudrait bien sûr faire dans un vrai système d'exploitation !).

4 Réseau

Notre projet de noyau s'appuie fortement sur le réseau, puisque nous n'avons aucune interface (écran, clavier, souris, ...) branchée directement sur la Raspberry Pi. Nous l'utilisons entre autres pour envoyer des logs à d'autres machines et ouvrir un shell distant.

4.1 Protocoles supportés

Nous avons implémenté différents protocoles réseau afin d'aboutir à une architecture représentant un bon compromis de simplicité d'utilisation et d'implémentation.

— Couche 2 (liaison de données) : protocole **Ethernet**, en partie implémenté par USPi (enrobage des frames). Il restait à coder l'ajout d'adresses MAC (destination, origine) et un EtherType à la frame.
— Couche 3 (réseau) : protocole **IPv4** servant d'enrobage à d'autres protocoles.
— Couche 3 (réseau) : protocole **ARP** afin de découvrir les adresses MAC sur le réseau local (nécessaire pour le protocole Ethernet si on ne souhaite pas hardcoder les adresses MAC des machines).
— Couche 4 (transport) : protocole **ICMP** partiellement implémenté pour répondre au ping.
— Couche 4 (transport) : protocole **UDP** fortement utilisé pour toute la transmission de données. UDP était bien plus simple à implémenter que TCP, et semble fiable lorsqu'utilisé sur un réseau local (aucune perte de paquets notée pendant les tests).

Nous avons de plus implémenté un protocole ad-hoc pour le logger. Lors de son initialisation, le logger envoie sur le réseau un paquet spécial (sur couche UDP), reconnu par le client de logs. Celui-ci a alors 500 ms pour répondre un autre paquet spécial, afin que le logger l'enregistre comme client. Le logger envoie ensuite ses logs à tous les clients ayant demandé à les recevoir, que ce soit pendant la phase d'initialisation, ou plus tard (mais alors, le client aura raté tous les logs de démarrage).

5 Programmes indépendants

5.1 Shell et démon de shell réseau

Nous avons implémenté un shell (`ush`) et un démon de shell réseau (`ushd`). Ceci nous permet de nous connecter au port 22 de la Raspberry Pi en UDP et d'obtenir une console. Celle-ci étant totalement non-sécurisée, nous l'avons nommée Unsecure SHell (ou Unreliable SHell ou Micro Shell, au choix) : la connexion passe par UDP, n'est pas chiffrée, se fait sans mot de passe, ...

Le démon écoute les paquets réseau sur le port 22 et ouvre des `ush` pour chaque nouveau client (*ie.* nouveau couple adresse-port), dont il gère l'entrée et la sortie standard pour les interfacer avec le réseau.

5.2 Commandes usuelles de shell

Les commandes suivantes sont implémentées : `cat`, `echo`, `ls`, `ps`, `pwd`, `cd`.

5.3 Interpréteur de Z-machine

La Z-machine est une machine virtuelle inventée en 1979 et spécialisée pour les jeux de fiction interactive : elle a été créée dans le but de rendre portables ce type de jeux publiés à cette époque, le nombre d'architectures existant étant très élevé ! Cette objectif de portabilité nous a permis d'en implanter également une version, qui peut éxécuter de manière minimaliste les fichiers de la version 5 de la Z-machine — une des plus utilisées de nos jours. Ainsi, ce petit programme ($\approx$ 1500 lignes) nous permet de jouer à un grand nombre de jeux (probablement plusieurs milliers !).

L'interpréteur se contente de lire le fichier donné en argument, et d'éxécuter son contenu en suivant la spécification (c.f. http://inform-fiction.org/zmachine/standards/z1point1/index.html) lorsque c'était possible et pas trop compliqué, ou en se débrouillant pour écrire quelque chose de minimaliste permettant aux jeux de fonctionner dans les autres cas. Le lecteur est invité à essayer des jeux comme *Balances* ou *All Things Devours* pour se représenter les jeux en question ; une partie importante des titres sortis par Infocom (dont *Trinity*, un des jeux du genre jugé parmi les meilleurs) devraient pouvoir être utilisés avec l'interpréteur fourni.

5.4 Serveur de Snake multijoueur

Nous avons également implanté un petit jeu de Snake multijoueur en Python, puis recodé le serveur en C++ en utilisant les appels système de notre noyau afin de pouvoir éxécuter celui-ci sur le Raspberry Pi. Il est ainsi possible de se connecter avec plusieurs joueurs, chacun contrôlant un serpent indépendamment.

6 Difficultés rencontrées

Recoder la bibliothèque standard

Nous avons eu des problèmes systématiquement dès lors que nous avons essayé d'utiliser les bibliothèques standard C et C++, qu'il s'agisse de problèmes de lien des objets, d'utiliser notre `malloc` et non le `malloc` standard, ...

Nous nous sommes donc résolus à recoder les parties de ces bibliothèques dont nous avons eu besoin : `printf`, quelques fonctions sur les chaînes de caractères C, un tableau redimensionnable, une table de hachage, ...

Écrasement de la table de vecteurs

La table de vecteurs se trouve à l'adresse mémoire 0×00. Une fonction de mailbox nécessitait un bloc de mémoire alloué sur le tas aligné sur 4 octets, et une inattention avait conduit à la ligne

```
buff = (uint32_t*) (mem + ((16 - ((Ptr)mem & 0xFFFFFFF0)) & (0xFFFFFFF0)));
```

au lieu de

```
buff = (uint32_t*) (mem + ((16 - ((Ptr)mem & 0xF)) & (0xFFFFFFF0)));
```

Le résultat est que `buff` se trouvait modulé — et non aligné — sur 4 octets, et toute écriture dans `buff` provoquait donc l'écrasement de la table de vecteurs, entraînant une erreur incompréhensible... surtout lorsqu'on doit la débugger avec des LEDs.

Liaison C et C++ à la fois de malloc

Pour faire marcher USPi, il est nécessaire d'implémenter les fonctions définies dans le header `uspi/include/uspios.h`, dont `malloc`, définie dans un bloc `extern "C"` dans ce header.

Or, `malloc` était également définie dans `malloc.h`, sans liaison C cette fois. Certains fichiers incluaient `uspios.h` sans inclure `malloc.h`, ainsi aucun warning de redéfinition n'était levé ; toutefois, si on appelait `malloc` depuis ces fichiers, les conventions d'appel différentes provoquaient une erreur brutale, et tout aussi difficile à trouver que la précédente.

`.text.startup` n'est pas au début

Pour s'assurer que l'assembleur contenu dans `init.s` serait placé par le linker script à l'adresse `0×8000`, nous avions manuellement indiqué que celui-ci devait se trouver dans la section `.text.startup`.

Cela marchait très bien, jusqu'au jour où un constructeur jamais appelé a été ajouté à une structure. Le compilateur a alors jugé bon de le placer dans la section `.text.unlikely`, placée *avant* `.text.startup`.

Ainsi, l'ajout d'un constructeur à une classe empêchait le démarrage (pas le moindre clignottement de LED !) de la Raspberry Pi, puisque le code exécuté au démarrage (adresse `0×8000`) était... un constructeur d'une structure inutile, puis un return.

Pile du mode SVC mal placée

La gestion des interrupts se fait par copie des registres du processus actuel sur la pile, afin de les sauver. Le code lançant le premier processus mettait ainsi le pointeur de pile, qui n'était plus utilisé par la suite au début de la table contenant les registres du processus à lancer, par souci de cohérence avec le reste du code.

Cependant, lors de l'ajout des appels SVC, le pointeur de pile se retrouvait alors là où on l'avait laissé, le Raspberry Pi démarrant en mode SVC. La pile écrasait alors la table des processus, et provoquant ainsi un crash difficile à expliquer.