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 avons 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 10 000 lignes de codes de C++, 700 lignes de C et 150 lignes d'assembleur ARM64. 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, `libdevice-tree` 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`. La table 1 décrit l'origanisation de cette plage d'adresse au sein de notre noyau.

1 :1 *mapping* Code du noyau, *Device Tree*	`0xffff 0000 0000 0000`
Mémoire Vidéo	`0xffff 1000 0000 0000`
Mémoire périphériques *MMIO*	`0xffff 2000 0000 0000`
Pages Utilisateurs programme, tas et pile des programmes utilisateurs	`0xffff 3000 0000 0000`
Buffers DMA	`0xffff 4000 0000 0000`
Tas du Noyau grandissant vers le bas	`0xffff 5000 0000 0000`
Système de Fichier en RAM	`0xffff a000 0000 0000`
Pile du Noyau grandissant vers le haut	`0xffff f000 0000 0000` `0xffff ffff ffff ffff`

Table 1 – Vue de la mémoire au sein de l'espace noyau.

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.
  Chaque nœud est composé d'un bit de donné indiquant si au moins une page (=une feuille) est libre parmis tout ses descendant, ou si la page est libre s'il s'agit d'une feuille. Ainsi on peut accéder à la disponibilité de chaque page en temps constant. On peut également trouver une page disponible, sous réserve d'existence, en $\bigcirc(\log n)$ temps, étant donné qu'on élimine la moitié des page existantes à chaque étape de descente dans l'arbre, le tout en gardant constament une page libre au moins parmis les descendants envisagés.
  La libération d'une page doit se faire en modifiant les ancêtre nécessaires pour prendre en compte la disponibilité de la page, donc en au plus $\bigcirc(\log n)$ temps également.
- `contiguous_page_alloc.hpp` travaille sur une petite portion de la mémoire physique (≈ 100 Mio) et s'occupe d'allouer une liste de pages contiguës. Cette allocation est nécessaire pour la création des *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 noyau, un *heap* est simulé à partir des adresses `0xffff500000000000`. En utilisant une structure de

liste doublement chainée de *MetaBlock*, on alloue la mémoire requise linéairement dans le heap, ainsi que les données suivantes :

— La taille *size* du bloc mémoire, sans compter le décalage permettant l'alignement du début du block si celui-ci est alloué.
— Les block suiants et précédent *next* et *previous*
— L'adresse de début de la mémoire allouée, sous la forme d'un pointeur et d'une adresse vers un tableau de charactères.

Ces block permettent de manipuler plus facilement les blocks mémoire, notamment pour les séparer en deux si le block alloué est trop grand ou pour fusionner deux block libres consécutifs.

2.4 Système de fichier

Nous utilisons les *bootloader* des *Raspberry Pi* afin de charger une image disque en mémoire à l'adresse physique `0x0x18000000`. Après configuration du *MMU*, nous sommes en mesure d'accéder à cette image depuis l'adresse virtuelle `0xffffa00000000000`.

Cette image disque est celle d'une partition FAT contenant les programmes utilisées et disponible dans notre système d'exploitation. La bibliothèque permettant de lire cette partition n'a pas été développé par nos soins, elle est disponible à l'adresse suivante : `http://elm-chan.org/fsw/ff/`. Les quelques fonctions nécessaire à son bon fonctionnement sont trouvables dans le fichier `ramdisk.cpp`.

3 *Drivers*

Plusieurs *drivers* ont été développé au cours de ce projet afin d'interagir avec les différents périphériques présent sur les *Raspberry Pi* supportés. Ils ont tous été développé à l'aide des documentations fournies par *Raspberry Pi* :

— *Raspberry Pi 3B+* : `https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf`
— *Raspberry Pi 4* : `https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf`

3.1 GPIO

Le premier pilote développé est celui qui est essentiel pour la communication à l'extérieur. Ce pilote s'occupe d'initialiser et de contrôler les 40 *pins* disponibles sur les *Raspberry Pi*. La figure 1 donne les fonctions de ces 40 connecteurs. Le pilote développé au sein du fichier `gpio.cpp` permet de lire et fixer les niveaux logiques, de configurer les résistances de *pull-up* ainsi que de traiter les interruptions générées par les changements de de niveau.

3.2 Mini UART

Un pilote afin d'utiliser l'implementation d'UART par *Broadcom*, Mini UART, a été développé au sein du fichier `miniuart.cpp`. Celui-ci permet d'établir une communication *serial*. Bien que fonctionnel, nous l'avons utilisé afin de fixer les problèmes du notre pilote UART, plus fiable. En effet, l'horloge de Mini UART est celle du GPU présent, ce qui limite la vitesse en baud de communication (et la rend impossible lors du changement d'horloge du GPU !).

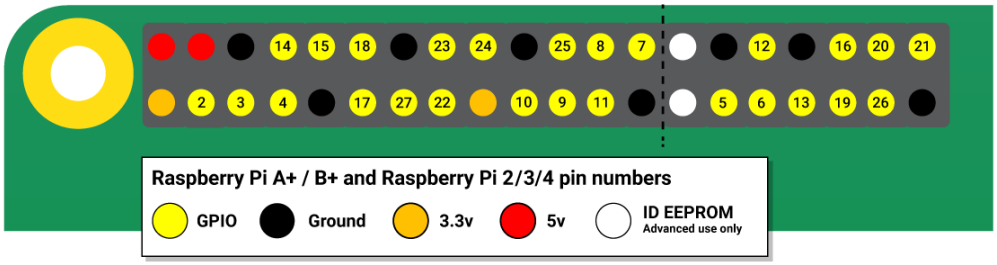

FIGURE 1 – Les 40 *pins* disponibles sur les *Raspberry Pi*.

3.3 UART

Un pilote UART qui est essentiel pour le débogage est présent au sein du noyau (dans le fichier `uart.cpp`). Celui-ci permet d'interagir avec les PL011 présent sur les cartes. La communication *serial* se fait, comme pour Mini UART, via les *pins* 14 et 15 du GPIO. Nous avons utilisé le *Raspberry Pi Debug Probe*, en figure 2, afin d'établir une communication *serial* entre les *Raspberry Pi* et nos ordinateurs.

FIGURE 2 – Le *Raspberry Pi Debug Probe*, l'adaptateur *serial* vers *USB* que nous avons utilisé.

3.4 *Mailbox* du *VideoCore*

La *Mailbox* du *VideoCore*[1] permet d'interagir facilement avec certains périphériques présent sur le SoC. Elle permet par exemple l'allocation des *Framebuffer* pour la video, la configuration des horologes ou le contrôle des LED d'activité. Un pilote permettant de communiquer via le protocole *Mailbox* au GPU est donc présent donc notre noyau, au sein du fichier `mailbox.cpp`.

3.5 Contrôleurs IRQ

Afin de gérer les interruptions générées par les périphériques, nous avons du développer des contrôleurs IRQ. Cependant, les deux SoC sur lesquels nous avons travaillés le BCM2711 et le BCM2835 ne possèdent pas les mêmes contrôleurs IRQ. Deux pilotes ont donc été intégrés au noyau, l'un propre au BCM2711, l'autre pour le GIC-400, présent sur les *Raspberry Pi 4*. Ces pilotes sont disponibles au sein du dossier `kernel/hardware/irq`.

1. Nom des GPU présent sur les *Raspberry Pi* développé par *Broadcom*.

3.6 Contrôleur DMA

Un pilote DMA a été développé afin de procéder à des copies entres des segments de la mémoire. Sur les *Raspberry Pi*, ce contrôleur est directement relié au bus utilisés par tous les périphériques pour communiquer, ce qui permet d'interagir directement avec eux. Ce pilote supporte par exemple, la copie de l'horloge du système directement vers l'UART, sans passer par le CPU. La connection du contrôleur directement au bus implique cependant que toutes les copies interagissant sur la SDRAM doivent être linéairement selon les adresses physiques. Cela a motivé le développement des *Buffer* DMA.

Ce pilote semble fonctionnel, cependant des problèmes sont apparu lors de l'utilisation du DMA pour de copier les fenêtres des processus sur le *framebuffer*. Plus de tests sont nécessaire pour s'assurer de son bon fonctionnement.

3.7 Horloge Système

Un pilote pour l'horloge du système, partagée entre les cœurs du CPU est présent au sein du noyau. Celui-ci permet de configurer le lancement d'IRQ de manière récurrente ou non. Le pilote est trouvable au sein du fichier `system_hardware.cpp`.

3.8 Timer ARM

Un pilote pour l'horloge de chaque cœur ARM est également au sein du noyau. Cependant, celui-ci ne supporte pas la configuration du lancement d'IRQ. Il reste très utile afin de récupérer à faible coût le temps écoulé depuis le démarrage du noyau.

4 Processus

Une implémentation de processus et threads est proposé dans notre OS. Celle-ci bien que imparfaite, propose tout de même de multiples fonctionnalités et reste flexible pour de futurs améliorations. Elle est lourdement inspiré de l'implémentation de Linux.

4.1 Hiérarchie

Au niveau de notre noyau, la notion de processus n'existe pas, ni celle de thread. Tout est une tâche (représentée par la classe C++ `Task`). Une tâche est de façon générale une opération qui peut être préempté ou ordonnancé : cela désigne donc tout objet qui sera manipulé par le scheduler.

Une tâche a systématiquement son propre stack et peut avoir optionnellement sa propre mémoire virtuelle. Il peut tourner à la fois en espace utilisateur (niveau EL0) ou en espace noyau avec un stack distinct (niveau EL1). Chaque tâche maintient aussi sa propre liste de fenêtre (pour le window manager) et de fichier ouverts.

Un processus est alors simplement une tâche qui tourne en espace utilisateur avec sa propre mémoire virtuelle. Un thread est une tâche qui partage les mêmes fichiers/fenêtres et la même mémoire virtuelle qu'avec une autre tâche (qui joue le rôle de processus parent).

4.2 Appels systèmes

Afin qu'une tâche puisse communiquer avec le noyau, un ensemble d'appels systèmes est supporté. Ces derniers se font avec la même convention que Linux : les arguments sont passés dans les registres `r0` à `r7`, l'identifiant de l'appel est passé dans `w8` et l'appel lui-même est effectué par l'instruction `svc #0`.

L'interprétation de l'identifiant et des arguments est fait par la table des syscalls. Cette dernière est spécifique à chaque tâche (aussi bien noyau qu'espace utilisateur). Toutefois, une seule table des syscalls a été implémenté dans notre noyau pour le moment. Elle est donnée dans 4.2.

Tous les syscalls ne peuvent pas forcèment être appelé par des tâches noyaux.

5 Ordonnanceur

5.1 Algorithme et Priorités

Le noyau utilise un scheduler préemptif pour déterminer la prochaine tâche à exécuter. Son algorithme exact est donné ici.

Toutes les tâches qui peuvent être actuellement exécutées sont réparties dans 32 queues en fonction de leur priorité. À chaque tick de l'horloge, s'il y a une tâche dans une queue de priorité plus élevée que la tâche actuelle, alors cette tâche est choisie pour être exécutée (même si le budget de temps n'est pas encore dépassé). Autrement, si le budget de temps de la tâche actuelle est dépassé (10 ms par défaut, dépend de la priorité de la tâche), alors la prochaine tâche est choisie dans la queue de priorité actuelle ou plus basse. La tâche précédente étant à nouveau ajoutée à la fin de la queue correspondant à sa priorité (algorithme round-robinn).

5.2 Mise en pause de Processus

Lorsqu'une tâche est mise en pause, on la retire des queues du scheduler. Pour la réveiller, il suffit de l'insérer un nouveau dans la queue correspondant à sa priorité.

Dans le cas d'un endormissement pour une certaine durée, la tâche est mise dans une delta queue. Dans cette structure de donnée, on préserve seulement le temps restant (avant le réveil) **relativement** à la tâche précédente dans la delta queue. De cette façon, la mise à jour de la queue est faite en temps constant (il suffit de mettre à jour la tête de la queue). Une fois le temps écoulé, on replace la tâche dans une des queues d'exécution du scheduler.

6 Écran et Fenêtres

6.1 *Framebuffer*

Au démarrage du noyau, une requête mailbox est effectuée pour allouer un framebuffer pour l'écran auprès du VideoCore (carte graphique intégrée de la Raspberry PI). Ce framebuffer n'est jamais réalloué plus tard et est directement modifié pour mettre à jour l'écran.

Normalement, un mécanisme de détection de la taille de l'écran est en place, mais ne marche pas en pratique (renvoie des tailles d'écran qui ne correspondant pas à la résolution native). De plus, un système de double buffering est aussi implémenté en utilisant les virtuals offsets du VideoCore, mais il ne fonctionne pas non plus sur le hardware.

6.2 Gestionnaire de fenêtres

Chaque tâche a la possibilité de créer une ou plusieurs fenêtres. Pour chaque fenêtre, un framebuffer est alloué en espace noyau pour stocker son contenu et une queue des messages est créé. La tâche utilisateur peut alors dessiner dans le framebuffer via des syscalls ou recevoir des messages du noyau sur les faits et gestes de la fenêtre (entrés claviers, déplacement, redimensionnement, fermeture demandée, etc.).

Le window manager en lui-même est une tâche noyau qui vérifie régulièrement s'il y a eu des modifications (déplacement de fenêtre, changement de focus, fenêtre mise à jour, etc.) et compose

Fonction C	Description
sys_exit	Tue la tâche avec un le code erreur donnée
sys_print	Affiche du texte dans le log noyau
sys_getpid	Renvoie l'identifiant de la tâche
sys_debug	Affiche un entier dans le log noyau
sys_spawn	Creé un nouveau processus enfant à partir du fichier ELF donnée
sys_sleep	Mets en pause la tâche pour le temps donné au minimum
sys_yield	Passes le contrôle à d'autres tâches
sys_sched_get_priority	Recupère la priorité actuelle de la tâche
sys_schet_set_priority	Définit la nouvelle priorité de la tâche
sys_sbrk	Change la position du heap (allocation de mémoire)
sys_open_file	Ouvre un fichier
sys_close_file	Ferme un fichier
sys_read_file	Lit des octects à partir d'un fichier ouvert
sys_get_file_size	Renvoie la taille d'un fichier ouvert
sys_open_dir	Ouvre un répertoire
sys_close_dir	Ferme un répertoire
sys_read_dir	Lit une entrée du répertoire
sys_poll_message	Récupère un message d'une fenêtre si disponible
sys_wait_message	Comme sys_poll_message mais bloque si plus de message
sys_window_create	Crée une fenêtre
sys_window_destroy	Détruit une fenêtre
sys_window_set_title	Mets à jour le titre d'une fenêtre
sys_window_get_visibility	Récupère la visibilité actuelle d'une fenêtre
sys_window_set_visibility	Affiche ou cache une fenêtre
sys_window_get_geometry	Récupéré la position et la taille d'une fenêtre
sys_window_set_geometry	Déplace ou redimensionne une fenêtre
sys_window_present	Redessine la fenêtre à l'écran
sys_gfx_clear	Efface le contenu d'une fenêtre par une couleur
sys_gfx_draw_line	Affiche une ligne dans une fenêtre
sys_gfx_draw_rect	Affiche un rectangle dans une fenêtre
sys_gfx_fill_rect	Remplit un rectangle dans une fenêtre
sys_gfx_draw_text	Affiche du texte dans une fenêtre
sys_gfx_blit	Copie une image 2D dans une fenêtre

Table 2 – Table des syscalls

les fenêtres à l'écran. C'est également ce dernier qui est responsable de dessiner l'arrière-plan (une image chargée à partir du système de fichier). Les décorations d'une fenêtre sont dessinées par le window manager dans le framebuffer de la fenêtre juste avant le blit vers le framebuffer global de l'écran.

7 Clavier PS/2

Afin d'interagir facilement avec l'interface de notre noyau, nous avons développé une interface ainsi qu'un pilote afin d'utiliser un clavier PS/2 sur les *Raspberry Pi*. Le protocole PS/2 n'étant pas supporté nativement par ces ordinateurs nous avons du construire un petit circuit électronique, visible en figure 3, afin de connecter un clavier aux *pins* du *GPIO*.

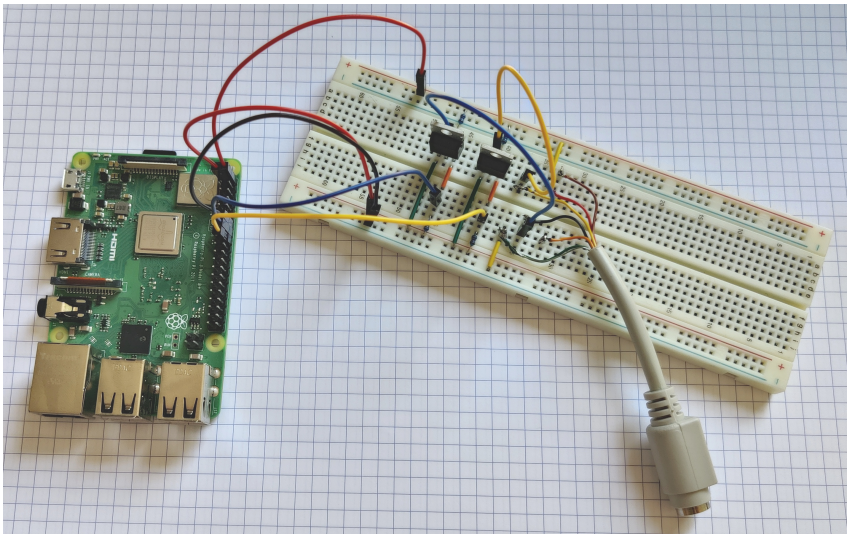

FIGURE 3 – Le circuit électronique utilisé pour interfacer un clavier PS/2 avec une *Raspberry Pi*.

Ce circuit d'alimenter et d'abaisser la tension de sortie des clavier PS/2 de 5 V au 3,3 V supportés par le *GPIO* des *Raspberry Pi*. Le pilote PS/2, visible dans le fichier `ps2_keyboard.cpp` est fait pour qu'à chaque front descendant de l'horloge (en bleu sur le circuit de la figure 3), une interruption est générée et permet de lire un bit d'information sur le fil pour les données (en jaune sur le circuit).

Après réception des 11 bits constituant un message du protocole PS/2, ce pilote s'occupe de traiter ce message afin d'en générer des événements claviers classiques. Ces derniers sont ensuite dispatché au *handler* configuré au sein du noyau.