# 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.

### 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 :

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

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

— `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 ($\approx 100$ 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`.

## 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.

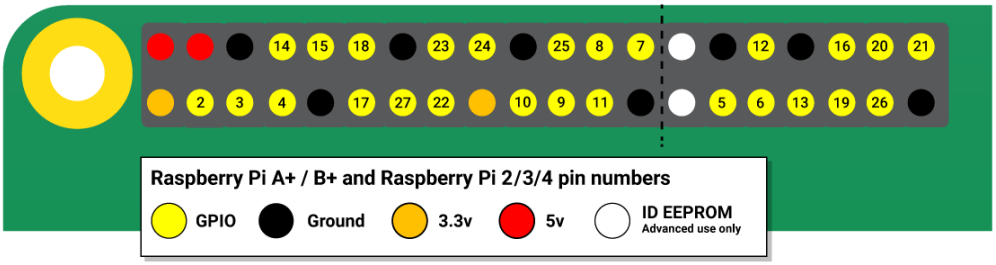


FIGURE 1 – Les 40 *pins* disponibles sur les *Raspberry Pi*.

## 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 !).

## 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.

## 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`.

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

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

## 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`.

## 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

## 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

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*.

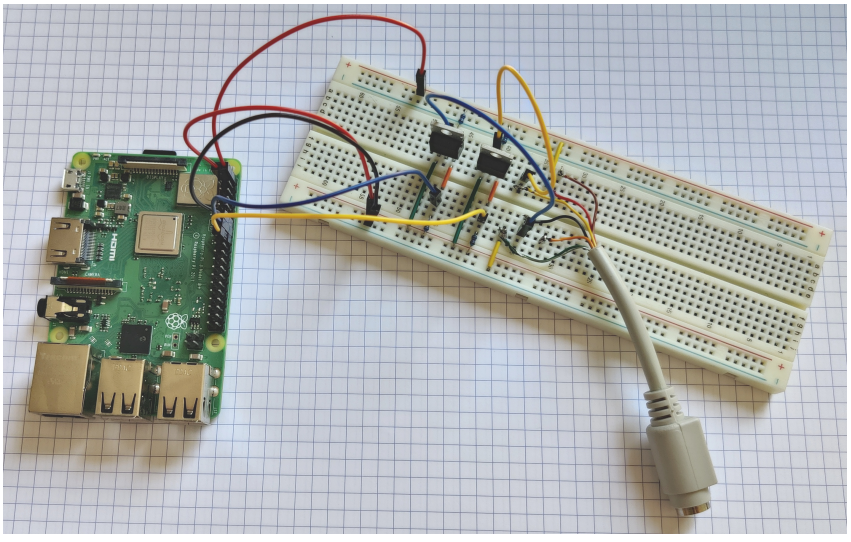

FIGURE 3 – Le circuit électronique utilisé pour interfacer un clavier PS/2 avec une *Raspberry Pi*.

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.