Accès direct aux périphériques mappés en mémoire

M2 SETI B4 / MS SE SE758

Guillaume Duc
guillaume.duc@telecom-paris.fr
2020–2021

Périphériques MMIO

- Jusqu'à maintenant nous avons étudié les méthodes de communication avec des périphériques connectés à des bus tels que I^2C, SPI, USB, etc. pour lesquels le noyau offre une API de communication
- Or, certains périphériques (en général les périphériques internes à un SoC ou des IP chargées dans un FPGA) sont directement mappés dans l'espace d'adressage physique
- La communication avec ces périphériques passe par des lectures et des écritures mémoires classiques
- On parle de périphérique MMIO (*Memory Mapped IO*)

Réservation

```
struct resource * request_mem_region(resource_size_t start,
                                     resource_size_t n,
                                     const char *name)

struct resource * devm_request_mem_region(struct device *dev,
                                          resource_size_t start,
                                          resource_size_t n,
                                          const char *name)
```

- Ces deux fonctions informent le noyau que notre pilote souhaite réserver la plage mémoire débutant à l'adresse physique `start` et de longueur `n` octets
- Si cette zone est déjà réservée par un autre pilote, elles retournent `NULL`
- La liste des zones réservées est disponible dans le fichier `/proc/iomem`
- Cette étape n'est pas strictement nécessaire mais recommandée

Libération

```
void release_mem_region(resource_size_t start,
                        resource_size_t n);
```

- Cette fonction supprime la réservation

Mapping mémoire

- Le périphérique est mappé en mémoire physique
- Or, le noyau travaille en espace d'adressage virtuel
- Il est donc nécessaire de mettre en place une plage d'adresses virtuelles qui sera traduite vers la plage d'adresses physiques correspondant au périphérique pour pouvoir y accéder

Mapping mémoire

```
void __iomem *ioremap(resource_size_t offset, size_t size);
void __iomem *devm_ioremap(struct device *dev,
                           resource_size_t offset,
                           resource_size_t size);
```

- Ces fonctions mettent en place une plage d'adresses virtuelles qui sont traduites vers la plage d'adresses physiques `offset` à `offset + size`
- Elles renvoient un pointeur vers la première adresse virtuelle ou `NULL` si une erreur s'est produite
- Ces fonctions s'assurent que le cache est désactivé pour l'accès à ces zones

Suppression du mapping mémoire

```
void devm_iounmap(struct device *dev, void __iomem *addr);
void iounmap(volatile void __iomem *addr)
```

- Ces deux fonctions suppriment le mapping mémoire

Mapping mémoire

- En théorie, pour accéder au périphérique, il ne reste plus qu'à faire des lecture et des écritures sur la plage d'adresses virtuelles obtenue
- En pratique c'est plus compliqué
  - Il faut s'assurer que le cache est désactivé (c'est le cas grâce à `ioremap`)
  - Certaines architectures ne le permettent pas (il faut donc passer par des primitives particulières)
  - Il ne faut pas que ces accès soient réordonnés par le compilateur (utilité du mot clé `volatile` et éventuellement des barrières de compilation)
  - Il ne faut pas que ces accès soient réordonnés par le processeur (barrières mémoires)

Accès bas niveau

```
u8  __raw_readb(const volatile void __iomem *addr);
u16 __raw_readw(const volatile void __iomem *addr);
u32 __raw_readl(const volatile void __iomem *addr);

void __raw_writeb(u8  val, volatile void __iomem *addr);
void __raw_writew(u16 val, volatile void __iomem *addr);
void __raw_writel(u32 val, volatile void __iomem *addr);
```

- Ces primitives permettent des lectures et des écritures
- Pas de conversion d'endianness
- Pas de protection contre le réordonnancement à l'exécution (utiliser les barrières mémoires décrites plus loin)

Accès haut niveau

```
u8  readb(const volatile void __iomem *addr);
u16 readw(const volatile void __iomem *addr);
u32 readl(const volatile void __iomem *addr);

void writeb(u8  val, volatile void __iomem *addr);
void writew(u16 val, volatile void __iomem *addr);
void writel(u32 val, volatile void __iomem *addr);
```

- Ces primitives permettent des lectures et des écritures
- Les données lues sont interprétées comme étant *little endian* et éventuellement converties
- Protection contre le réordonnancement à l'exécution (des barrières mémoires sont utilisées par ces fonctions)

Barrières mémoires

- `rmb()` : Barrière en lecture. Toutes les lectures situées avant (dans le code) cette barrière doivent être effectuées avant la moindre lecture après cette barrière
- `wmb()` : Barrière en écriture. Toutes les écritures situées avant la barrière doivent être effectuées avant la moindre écriture après la barrière
- `mb()` : Barrière en lecture et écriture. Tous les accès mémoires (lecture et écriture) situés avant la barrière doivent être effectués avant le moindre accès mémoire situé après
- Explications détaillées dans `Documentation/memory-barriers.txt`