Gestion des périphériques

MI074 - 8

Abstraction des périphériques

- On souhaites abstraire les périphériques (device), c'est à dire unifier leur gestion malgré leur diversité.
- On va définir un device comme un objet générique qui va être spécialisé en fonction du device réel
- On va définir une collection de fonctions (une API) pour chaque type de device dont:
  - le prototype est imposé par le noyau mais dont
  - l'implémentation est spécifique à chaque type de device

  L'ensemble de ces fonctions constitue le driver du device

Device générique

```
struct device_s {
    // public
    spinlock_t              lock;   // en cas d'usage par + threads
    void *                  base;   // adresse en mémoire physique
    uint_t                  irq;    // numéro de ligne INT
    uint_t                  type;   // type du device
    driver_s                op;     // opérations suportées
dépend                                          // du type
    struct irq_action_s   action; // handler d'interruption

    // private
    void *                  data;   // dépend du driver
}
```

action

Associe un device et un handler d'interruption (ISR)

```
typedef void (irq_handler_t) (struct device_s *dev);

struct irq_action_s {
    struct device_s *dev;
    irq_handler_t * irq_handler;
}
```

Nous allons voir, plus loin, comment associer une ligne d'interruption avec une irq_action

Drivers génériques

ICU

```
void set_mask(struct device_s *dev, uint_t mask);
uint_t get_mask(struct device_s *dev);
uint_t get_highest_irq(struct device_s *dev);
```

TIMER

```
void set_period(struct device_s *dev, uint_t period);
uint_t get_value(struct device_s *dev);
```

GENERIC

```
ssize_t read(struct device_s *dev, struct request_s *req);
ssize_t write(struct device_s *dev, struct request_s *req);
```

Requête générique

Une requête décrit un travail à réaliser pour tout device

```
struct request_s {
    // public
    void *    src;
    void *    dst;
    size_t    size;

    // private
    list_t    list;
    int       error;
    void *    data;
}
```

Cas d'étude : ICU

Fonctions publiques

soclib_icu_init(struct device_s *icu, void * base, uint_t irq);

- initialise : lock base, irq, type, op (driver), irq_action
- initialise la structure de travail : data

soclib_icu_bind(struct device_s *icu, struct device_s *dev);

- fait un lien entre le device de l'icu et un autre device

Fonctions privées

soclib_icu_set_mask(...)
soclib_icu_get_mask(...)
soclib_icu_get_highest_irq(...)
soclib_icu_irq_handler(...)

Cas d'étude : ICU

- pour l'icu le champ data sera simplement un tableau de pointeur irq_action_s avec autant de cases qu'il y a de d'interruption reçues par l'ICU.

- L'opération bind a pour effet d'initialiser la case de ce tableau dont le numero se trouve dans le device lui-même

```
struct irq_action_s **action = icu->data;
action[dev->irq] = &(dev->action);
```

Routage des interruptions

Le cpu peut être considéré comme un device
au sens où il reçoit des interruptions.

Fonctions publiques

cpu_init(struct sched_s *cpu);

- initialise la structure de travail (data) de la structure sched

cpu_bind(struct sched_s *cpu, struct device_s *dev);

- fait un lien entre le cpu et un device

Routage des interruptions

Un device crée un événement et lève une interruption.
l'interruption est routée physiquement jusqu'au CPU en passant par l'ICU. On suppose qu'elle n'est pas masquée.

Le processeur est interrompu, il exécute :

- kentry
- __do_interrupt qui agit comment le handler irq du CPU
  - l'argument de do_interrupt est égal à l'état des lignes IRQ
  - on va chercher dans le tableau data l'action à exécuter et on l'exécute :

```
action = sched->data[highest(irqstate)];
action->irq_handler(action->dev);
```

Initialisation de l'architecture

__do_init va appeler une fonction arch_init définie dans le répertoire arch/soclib

cette fonction va demander l'initialisation des devices et du cpu puis faire des liens pour les handler avec les fonctions bind

handler du timer

Au plus simple on va supposer que le timer est programmé pour provoquer une interruption périodique, et qu'à chaque période, le système demande une commutation de thread.

donc de manière périodique: on appelera
soclib_timer_irq_handler(struct device_s * timer)

où l'on fait deux choses:
reset_irq
sched_yield

Appel d'une fonction de driver

Si on veut appeler une fonction d'un driver de device,
il faut connaitre le device et la fonction du driver.
Par exemple pour écrire une chaine de sur un tty

```
void tty_puts(uint_t tty_id, char *buffer)
{
    struct device_s *tty = &tty_tbl[tty_id];
    struct dev_request_s req;
    uint_t size, written_char=0;
    for(size=0;buffer[size];size++);
    while (size != written_char) {
        req.src = buffer + written_char;
        req.size = size - written_char;
        written_char += tty->op.generic.write(tty, &req);
    }
}
```