Informatique embarquée 15/16

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

4 avril 2017

Accès mémoire par le noyau

- Liste des ressources occupées

```
# cat /proc/iomem
...
01c20800-01c20bff : /soc@01c00000/pinctrl@01c20800
```

- Requête de ressource par un pilote

```
#define PWM_BASE 0x01c20c00

int hello_start()  // init_module(void)
{sta=request_mem_region(PWM_BASE,12,"PWM test");
 if (sta==NULL)
    printk(KERN_ALERT "mem request failed");
 else
   {jmf_gpio = (u32)ioremap(PWM_BASE, 12);
    writel((1<<4),(void*)(jmf_gpio)); // 24000/120=200 kHz
    writel((100<<16)+50,(void*)(jmf_gpio+4));
   }
 return 0;
}
```

```
# cat /proc/iomem
01c20800-01c20bff : /soc@01c00000/pinctrl@01c20800
...
01c20c00-01c20c0b : PWM test
```

Accès aux GPIO par le noyau : gpiolib

Configuration du noyau Linux pour intégrer gpiolib

```
CONFIG_GPIO_SUNXI:

This option enables support for GPIOs on the Allwinner
SOCs (sun4i/sun5i/sun7i). The GPIOs must be
defined in [gpio_para] section of sysconfig.fex
file (gpio_used/gpio_num/gpio_pin_x variables)


Symbol: GPIO_SUNXI [=y]
Type  : tristate
Prompt: GPIO Support for sunxi platform
  Defined at drivers/gpio/Kconfig:181
  Depends on: GPIOLIB [=y] && (ARCH_SUN4I [=n] || ARCH_SUN5I [=y] || ARCH_S
  Location:
    -> Device Drivers
      -> GPIO Support (GPIOLIB [=y])
```

Pour utiliser GPIO : (port-'A')×32+broche

Accès aux GPIO par le noyau : gpiolib

Le noyau fournit des méthodes pour accéder aux GPIO :

```
#include <linux/gpio.h>
int my_gpio=('G'-'A')*32+9; // GPIO ID
gpio_is_valid(my_gpio);
gpio_request_one(my_gpio, GPIOF_OUT_INIT_LOW, "jmf_gpio");
gpio_set_value(my_gpio,jmf_stat);
gpio_free(my_gpio);
```

1. Vérifier si le GPIO existe sur cette architecture (port*32+pin)
2. Requérir la broche (direction, état par défaut)
3. Manipuler l'état de la broche
4. Relacher la ressource

```
MODULE_LICENSE("GPL");
```

sinon

```
[  525.704550] 4mymod_version_gpiolib: module license 'unspecified' taints kern.
[  525.711983] Disabling lock debugging due to kernel taint
[  525.717944] 4mymod_version_gpiolib: Unknown symbol gpiod_set_raw_value (err )
[  525.725256] 4mymod_version_gpiolib: Unknown symbol gpio_free (err 0)
[  525.731627] 4mymod_version_gpiolib: Unknown symbol gpio_request_one (err 0)
[  525.738610] 4mymod_version_gpiolib: Unknown symbol gpio_to_desc (err 0)
```

Module noyau : gestion des interruptions

Seul un module noyau peut accéder aux interruptions du processeur (exemple : interruption 7 est associée à la broche 10 du port parallèle).

```
jmfriedt@(none):~$ cat /proc/interrupts | grep parp
  7:          0    IO-APIC-edge  parport0
```

qui se gère par [1]

```
#define PARALLEL_PORT_INTERRUPT 7
#define BASEPORT                0x378
int s=0;
static int int_handler(void) {printk(">>> PARALLEL PORT INT\n");s=1;}
static int __init jmfpport_init_module (void) {
  ret = request_irq(PARALLEL_PORT_INTERRUPT, &int_handler,
                    SA_INTERRUPT, "parallelport", NULL);
  enable_irq(7);
  outb_p(0x10, BASEPORT + 2);
}
static ssize_t skeleton_read (struct file *file, char *buf,
                size_t count, loff_t *ppos) {
 [...]
 do {} while (sortir==0);    // interruptible_sleep_on(&skeleton_wait);
 err = copy_to_user(buf,string,counter);
 ...
}
```

1. www.captain.at/howto-linux-device-driver-template-skeleton.php

Interruptions et gpiolib

```
my_gpio =('B'-'A')*32+2;  // PB2
err=gpio_is_valid(my_gpio);
err=gpio_request_one(my_gpio, GPIOF_IN, "jmf_irq");
irq = gpio_to_irq(my_gpio);
irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf→
    ↪", &dev_id);
...
free_irq(irq,&dev_id);
gpio_free(my_gpio);  // libere GPIO
```

appellera `irq_handler` chaque fois que l'interruption se déclenche.

`irq_handler` finit par `return IRQ_HANDLED;`

```
[ 4446.692969] gpio_request 34=0
[ 4446.696059] gpio_to_irq=47
[ 4446.698827] finished IRQ: error=0

# cat /proc/interrupts
...
 47:          1  sunxi_pio_edge  16 Edge      GPIO jmf
```

Les signaux : prévenir d'un évènement

Équivalent logiciel : les signaux

```
#include <signal.h>

void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);}

int main ( void ) {
    signal (SIGINT, my_handler);
    while (1) {}
}
```

qui donne chaque fois qu'on appuie sur CTRL-C (tuer par `kill -9 PID`)

```
jmfriedt@(none):~$ ./sigint
I got SIGINT, number 2
I got SIGINT, number 2
```

Distribution du signal d'interruption

Distribution d'interruption par signaux

1. un unique point de gestion d'une interruption matérielle (module noyau)
2. une multitude de processus utilisateurs réagissent à l'interruption
3. chaque processus s'enregistre auprès du module
4. le module réagit au plus vite à l'interruption ...
5. ... puis, quand il a le temps, prévient tous les processus identifés par leur ID.

Génération du signal

Depuis le noyau : identifier le processus par son PID et envoyer le signal

```
struct siginfo sinfo;
struct task_struct *task;

// on cherche PID au cas ou' le process aurait disparu
memset(&sinfo, 0, sizeof(struct siginfo));
sinfo.si_signo= SIGUSR1;     // depuis son enregistrement
sinfo.si_code = SI_USER;
task = pid_task(find_vpid(pid), PIDTYPE_PID);
if (task == NULL) {pr_info("Cannot find PID\r\n");}
   else send_sig_info(SIGUSR1, &sinfo, task);
```

Processus potentiellement lent ⇒ exécuter dans une sous-tâche

```
static void do_something(unsigned long data) {... }

int hello_start()  // init_module(void)
{ INIT_WORK(&irq_work, do_something);
  request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &irq_id);
...
}

static irqreturn_t irq_handler(int irq, void *dev_id)
{schedule_work(&irq_work);  // on le fera quand on aura le temps
 return IRQ_HANDLED;
}
```

Distribution du signal d'interruption

Solution alternative (*thread*)

1. un processus désire être informé de l'interruption au cours de son exécution
2. génère un thread qui bloque en lecture sur un point d'entrée de `/sys/class` ou `/dev`
3. quand l'interruption se déclenche, le module débloque le processus en lecture ... [2] [3]
4. ... et le thread prévient son père de l'évènement.

2. `http://www.makelinux.net/ldd3/chp-6-sect-2`
3. `http://www.makelinux.net/ldd3/chp-5-sect-3` pour les sémaphores en espace noyau

Sémaphore

... pour débloquer la lecture, ici sur un timer.

```
static int dev_read(struct file *fil,char *buff,size_t len→
    ↪,loff_t *off)
{
 down (&mysem);
...
 return len;
}

static void do_something(unsigned long data)
{up (&mysem);}

int hello_start()  // init_module(void)
{sema_init(&mysem, 1); /* init the semaphore as full */
// simule decl. interruption pour debloquer semaphore
 init_timer_on_stack(&exp_timer);
 exp_timer.expires = jiffies + delay * HZ;
 exp_timer.data = 0;
 exp_timer.function = do_something;
 add_timer(&exp_timer);
 return t;
}

module_init(hello_start);
```

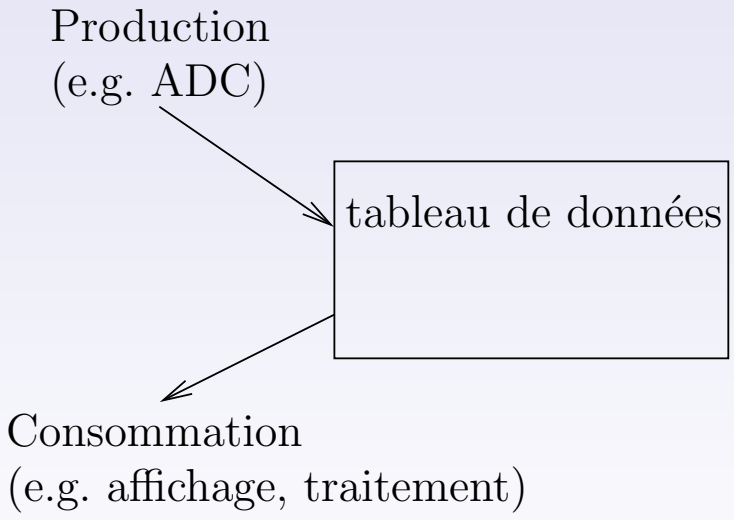
Protection des données

Plusieurs tâches gèrent une même donnée ⇒ risque d'incohérence
Trois méthodes pour protéger une donnée :

- sémaphore (compteur)
- mutex (binaire avec passage en mode veille de la tâche)
- spin-lock (binaire avec test du vérou)

Producteur-consommateur :

1. une tâche produit des données
2. un utilisateur demande ces données
3. la lecture est **bloquée** tant que les données n'ont pas été produites



Protection des données : sémaphore

- Abaisser le sémaphore bloque s'il est nul
- Relever le sémaphore débloque le processus en attent
- Le compteur peut croître pour débloquer plusieurs processus

```
#include <linux/semaphore.h>
struct semaphore mysem;

sema_init(&mysem, 0);       // init the semaphore as empty

down (&mysem); // bloque le consommateur
...
up (&mysem); // producteur debloque
```

Protection des données : mutex

Bloque-débloque une zone de code qui accède à une ressource commune :
une seule instance d'un mutex peut accéder à ce segment de code à un instant donné

```
#include <linux/mutex.h>
struct mutex mymutex;

mutex_init(&mymutex);

mutex_lock(&mymutex);   // bloque
...
mutex_unlock(&mymutex); // debloque
```

Protection des données : spinlock

Le mutex met la tâche en veille ⇒ procédure lourde et lente
Pour des opérations rapides (interruptions), sonder continuellement le vérou : `spinlock`

```
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(myspin);

spin_lock_init(&myspin);

spin_lock(&myspin);
...
spin_unlock(&myspin);

MODULE_LICENSE("GPL");
```

Problème de *deadlock* : deux processus faisant appel à un mécanisme de bloquage sont imbriqués.

Protection des données : spinlock

Le mutex met la tâche en veille ⇒ procédure lourde et lente
Pour des opérations rapides (interruptions), sonder continuellement le vérou : `spinlock`

```
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(myspin);

spin_lock_init(&myspin);

spin_lock(&myspin);
...
spin_unlock(&myspin);

MODULE_LICENSE("GPL");
```

Consommateur	Producteur
Mutex protège	Mutex protège
Sémaphore bloque	...
...	Mutex libère
Mutex libère	Sémaphore débloque

Problème de *deadlock* : deux processus faisant appel à un mécanisme de bloquage sont imbriqués.

Mise en pratique

1. Mise en pratique de `gpiolib` pour exploiter une broche
2. ... et s'en attribuer l'interruption matérielle (complexe à la lecture de la *datasheet*)
3. License GNU pour pouvoir exploiter les fonctionnalités d'autres modules noyau (abstraction du matériel)
4. Exploitation des fonctionnalités du noyau pour échanger avec l'espace utilisateur : sémaphore et tasklet/timer.

Exercices :

1. requérir une broche et la commander par `gpiolib`
2. faire clignoter cette broche sous contrôle d'un *timer* du noyau
3. associer un gestionnaire d'interruption à PB2
4. clignoter la LED sur déclenchement de cette interruption *avec attente* avant de changer à nouveau d'état (*debounce*)