

Interactions avec le matériel

Christophe BLAESS

christophe.blaess@logilin.fr

<https://www.blaess.fr/christophe/>

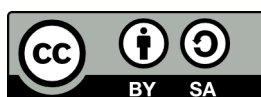


Ingénierie et formations sur Linux et les logiciels libres
<https://www.logilin.fr>

Entrées-sorties.....	3
Entrées-sorties par ports dédiés.....	3
Entrées-sorties par projection en adresses virtuelles.....	4
Entrées-sorties par GPIO.....	5
À vous : interactions par GPIO.....	8
Les contextes du noyau.....	9
Contexte d'appel-système.....	9
Contextes d'interruption.....	10
Contexte de thread du noyau.....	11
Gestion des interruptions.....	12
Principes.....	12
Activation et désactivation des interruptions.....	13
Routine de service personnelle.....	14
À vous : déclenchement d'interruptions par GPIO.....	15
Traitement d'interruption différé.....	16
Exécution différée par tasklet.....	18
Exécution différée par workqueue.....	19
Threaded interrupts.....	20
Travaux pratiques : driver virtuel en mode caractère.....	21
Protection des variables globales.....	22
Protections par spinlock.....	23
Travaux pratiques : synchronisation par mutex et spinlock.....	24
Attentes d'événements.....	25

Principe.....	25
Utilisation d'une waitqueue.....	26
Opérations bloquantes et non-bloquantes.....	27
Travaux pratiques : attentes d'événements.....	28
Projection de mémoire.....	29
Appel-système mmap().....	29
À vous : implémentation d'une méthode <i>mmap()</i>	30
Transferts de données par DMA.....	31

Ce support de formation est distribué sous licence **Creative Commons 4.0**



(Attribution - Partage dans les mêmes conditions).

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

v. 9.11

Entrées-sorties

Entrées-sorties par ports dédiés

Réserver une plage d'entrées-sortie :

```
<asm/io.h>
```

```
void * request_region (int base, int nb, const char *name);  
void   release_region (int base, int nb);
```

Lecture sur un port :

```
unsigned char inb (unsigned int port);  
unsigned short inw (unsigned int port);  
unsigned long inl (unsigned int port);
```

Écriture sur un port :

```
void outb (unsigned char value, unsigned int port);  
void outw (unsigned short value, unsigned int port);  
void outl (unsigned long value, unsigned int port);
```

Sur les architectures PC classiques, les périphériques sont accessibles par des opérations d'*entrées/sorties* utilisant des instructions spécifiques. Sur d'autres architectures, les périphériques sont projetés dans l'espace mémoire à des adresses réservées. Nous étudierons ce principe plus loin.

La fonction `request_region()` renvoie un pointeur NULL en cas d'erreur. Les plages d'entrées-sorties réservées sont visibles dans `/proc/ioprots`.

Entrées-sorties par projection en adresses virtuelles

Projection dans l'espace d'adressage du kernel :

```
void __iomem * ioremap (phys_addr_t base_addr,  
                        unsigned long length);  
  
void iounmap (void __iomem *addr);
```

Accès par octets, mots, longs :

```
unsigned char ioread8 (void __iomem *addr);  
unsigned short ioread16 (void __iomem *addr);  
unsigned int ioread32 (void __iomem *addr);  
  
void iowrite8 (unsigned char value, void __iomem *addr);  
void iowrite16 (unsigned short value, void __iomem *addr);  
void iowrite32 (unsigned long value, void __iomem *addr);
```

Accès par blocs :

```
void memcpy_fromio (void *kbuffer, void __iomem *addr,  
                    unsigned long length);  
void memcpy_toio (void __iomem *addr, void *kbuffer,  
                  unsigned long length);  
void memset_io (void __iomem *addr, unsigned char c,  
                unsigned long length);
```

Les méthodes `ioread()`, `iowrite()` et `mem*io()` contiennent des barrières mémoire évitant les optimisations inappropriées du compilateur.

Les fichiers `exemple-fpga.c` et `exemple-fpga.h` présents dans les sources des exemples montrent un extrait d'un projet réel de drivers utilisant les projections mémoire.

Entrées-sorties par GPIO

Réservation et visibilité dans /sys :

```
int  gpio_request (unsigned int gpio, const char * ident);  
void gpio_free    (unsigned int gpio);  
  
int  gpio_export  (unsigned int gpio);  
int  gpio_unexport (unsigned int gpio);
```

Sens d'accès :

```
int  gpio_direction_input  (unsigned int gpio);  
int  gpio_direction_output (unsigned int gpio, int value);
```

Lecture et écriture :

```
int  gpio_get_value (unsigned int gpio);  
int  gpio_set_value (unsigned int gpio, int value);
```

Numéro d'interruption associée :

```
int  gpio_to_irq (unsigned int gpio);  
int  irq_to_gpio (unsigned int irq);
```

La plupart des processeurs conçus pour les systèmes embarqués disposent de broches d'entrées-sorties génériques (*General Purpose Input-Output*).

Le système de fichiers virtuel /sys permet d'accéder aux GPIO dans /sys/class/gpio.

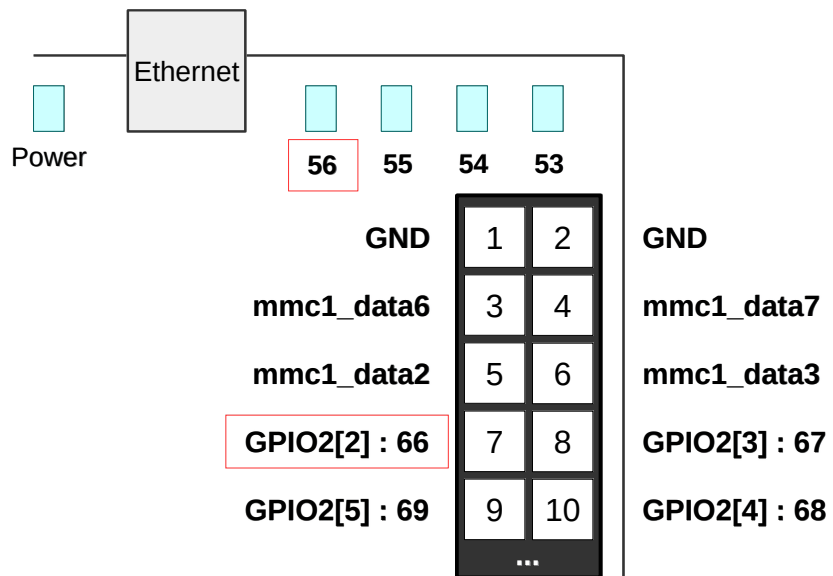
Les GPIO du Raspberry Pi

<i>Function</i>	<i>Pin</i>		<i>Pin</i>	<i>Function</i>
+3.3 V	1		2	+5 V
I2C SDA / GPIO 2	3		4	+5 V
I2C SCL / GPIO 3	5		6	GND
GPIO 4	7		8	UART 0 TX / GPIO 14
GND	9		10	UART 0 RX / GPIO 15
GPIO 17	11		12	PWM / GPIO 18
GPIO 27	13		14	GND
GPI O22	15		16	GPIO 23
+3.3 V	17		18	GPIO 24
SPI 0 MODSI / GPIO 10	19		20	GND
SPI 0 MISO / GPIO 9	21		22	GPIO 25
SPI 0 CLK / GPIO 11	23		24	SPI 0 CE 0 / GPIO 8
GND	25		26	SPI 0 CE 1 / GPIO 7
ID SD	27		28	ID SC
GPIO 5	29		30	GND
GPIO 6	31		32	GPIO 12
GPIO 13	33		34	GND
PCM FS / GPIO 19	35		36	UART0 CTS / GPIO 16
GPIO 26	37		38	PCM DIN / GPIO 20
GND	39		40	PCM DOUT / GPIO 21

Raspberry Pi Expansion Connector

Attention : les entrées GPIO doivent recevoir des signaux en [0, +3.3V] pas en [0, +5V] !

Les GPIO de la Beaglebone Black



À vous : interactions par GPIO

Utilisez un Raspberry Pi ou un BeagleBone Black avec une chaîne de compilation native (ou une chaîne de cross-compilation sur un PC).

Vérifiez la configuration du fichier `gpio-examples.h`.

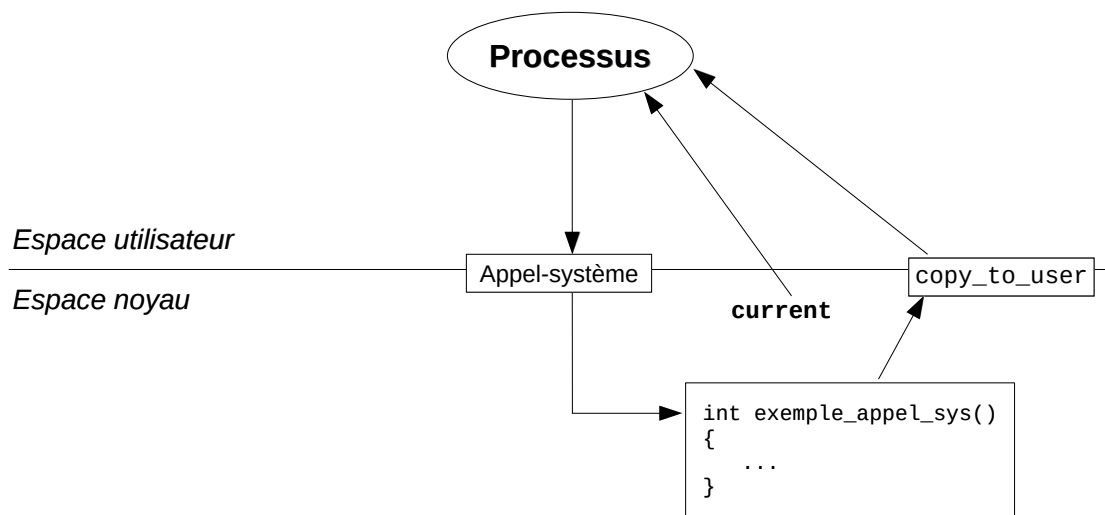
Examinez le code du driver `example-V-01.c` puis compilez-le.

Chargez le module dans le noyau et vérifiez les actions sur les GPIO concernées.

Les contextes du noyau

Il existe trois contextes d'exécution dans le noyau : appel-système, interruption et *threads* du noyau :

Contexte d'appel-système



Le code du noyau est exécuté pour le compte d'un processus.

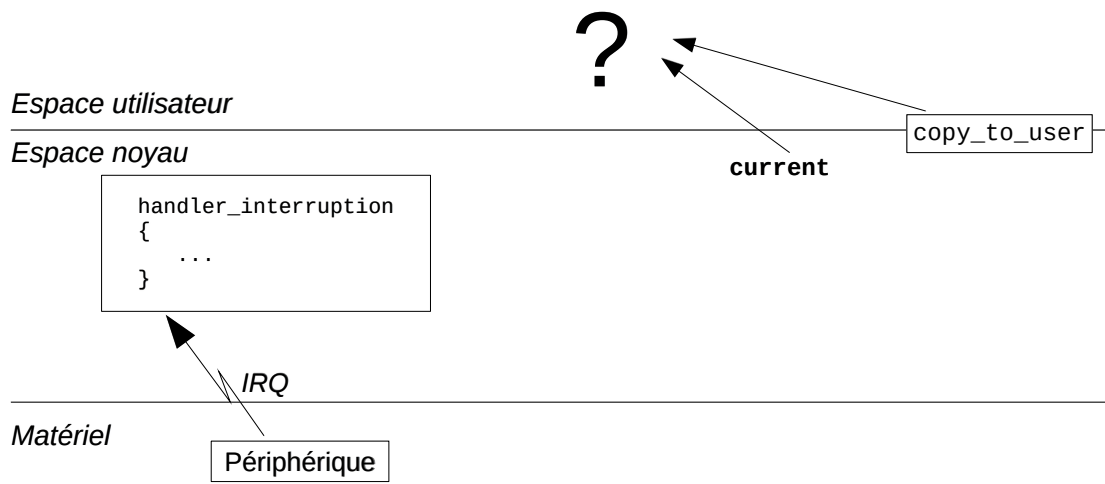
La variable `current` pointe sur une structure `task_struct` représentant la tâche en cours d'exécution.

Il est possible d'accéder à la mémoire virtuelle du processus correspondant à cette tâche avec les routines `copy_from_user()`, `copy_to_user()`, etc.

On peut endormir le processus en appelant `schedule()` ou une fonction bloquante (par ex. `mutex_lock_interruptible()` ou `wait_event_interruptible()`).

L'appel-système doit être **réentrant** (ne pas utiliser de variables globales sans protection) car il peut être invoqué à deux reprises en parallèle.

Contextes d'interruption



Le code du gestionnaire d'interruption est exécuté de manière imprévisible.

La variable `current` n'est pas utilisable, ni l'espace mémoire utilisateur.

On ne doit en aucun cas s'endormir !

Le gestionnaire est appelé avec sa propre interruption coupée sur tous les processeurs, ainsi il n'a pas besoin d'être réentrant.

Contexte de thread du noyau

```
#include <linux/kthread.h>

struct task_struct *kthread_run (int (*function)(void *),
                                void *param,
                                const char *name);

bool kthread_should_stop();

int kthread_stop(struct task_struct *thread);
```

Le code est exécuté dans l'espace mémoire du noyau, le processeur étant en mode superviseur.

Le déroulement du code est toutefois déclenché par l'ordonnanceur, au même titre que les processus utilisateur.

À vous...

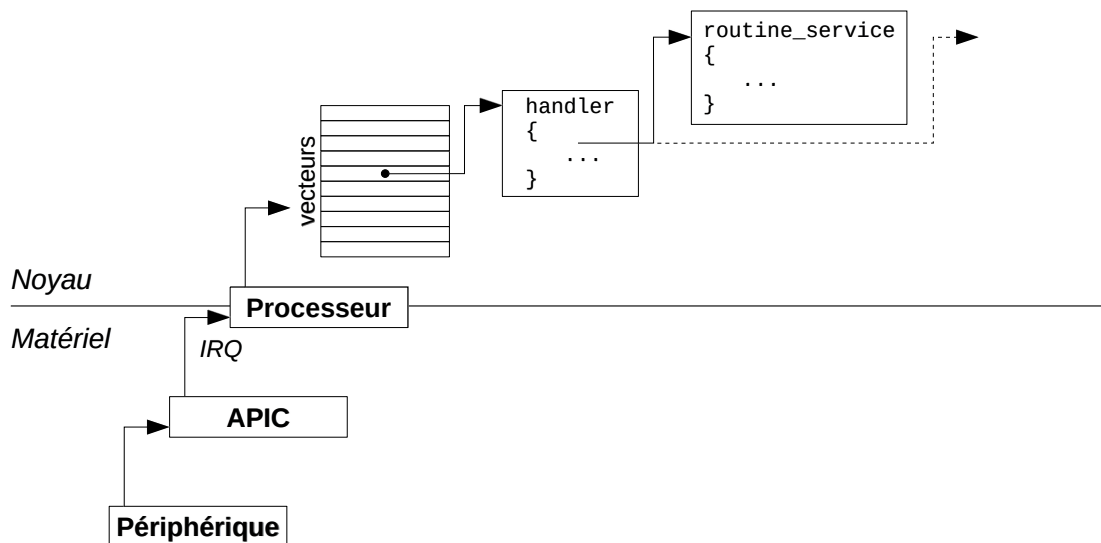
Chargez le module `example-V-02`.

Vérifiez la présence du thread noyau dans la liste des processus et des messages dans les traces du système.

En déchargeant le module, vérifiez que le thread se termine.

Gestion des interruptions

Principes



Lorsqu'un périphérique désire notifier le système de l'occurrence d'un événement externe, il change l'état d'une ligne d'IRQ (*Interrupt ReQuest*).

Le **contrôleur d'interruption** (anciennement du type 8259, puis *Southbridge*, et plus récemment un APIC - *Advanced Programmable Interrupt Controller*) transmet la demande d'interruption au processeur.

Le processeur sauvegarde l'état de ses registres, puis se branche à une adresse mémoire (vecteur) correspondant au numéro de l'interruption.

A cette adresse se trouve une routine de gestion (*handler*) qui acquittera l'interruption pour le contrôleur et invoquera une ou plusieurs routines de service installées par les drivers.

Pour en savoir plus

« IRQs: the Hard, the Soft, the Threaded and the Preemptible »

Alison Chaiken - ELCE 2016 – YouTube channel « *The Linux Foundation* »

<https://www.youtube.com/watch?v=-pehAzaP1eg>

Activation et désactivation des interruptions

Activer/désactiver **une** interruption sur **tous** les cœurs :

```
void disable_irq      (int irq);  
void disable_irq_nosync (int irq);  
void enable_irq      (int irq);
```

`disable_irq_nosync()` n'attend pas la fin de l'exécution d'un éventuel *handler* associé à cette *irq* qui serait en train de s'exécuter sur un autre CPU.

Activer/désactiver **toutes** les interruptions sur **le** cœur appelant :

```
void local_irq_disable (void);  
void local_irq_enable (void);
```

Masquer **toutes** les interruptions, puis restaurer celles qui étaient actives :

```
void local_irq_save    (unsigned long msq);  
void local_irq_restore (unsigned long msq);
```

La macro `local_irq_save()` utilise bien un argument `unsigned long`, et non un pointeur, même si elle stocke dedans l'état des interruptions actives.

A titre d'exemple, on trouve dans le répertoire `drivers/` du noyau 5.10 :

- 381 appels à `local_irq_save()`,
- 923 appels à `disable_irq()`,
- 91 appels à `local_irq_disable()`,

Toutefois ces mécanismes de protection contre les interruptions sont plutôt rares par rapport aux :

- 14.067 appels à `spin_lock_irqsave()`,
- 2.893 appels à `spin_lock_bh()`.

que nous verrons ci-après.

Routine de service personnelle

```
<linux/interrupt.h>
```

```
int request_irq (int irq,  
                irqreturn_t (*handler)(int, void *),  
                int attr, char *name, void *ident);  
  
void free_irq (int irq, void *ident);
```

attributs : IRQF_SHARED (partagée), IRQF_TRIGGER_RISING / IRQF_TRIGGER_FALLING (fronts montant ou descendant), IRQF_TRIGGER_HIGH / IRQF_TRIGGER_LOW (niveaux) etc.

```
irqreturn_t handler (int irq, void *ident);
```

Le gestionnaire doit renvoyer IRQ_HANDLED ou IRQ_NONE.

```
int devm_request_irq (struct device *dev, int irq,  
                    irqreturn_t (*handler)(int, void *),  
                    int attr, char *name, void *ident);  
  
void devm_free_irq (struct device *dev,  
                  int irq, void *ident);
```

Attention, `request_irq()` peut endormir l'appelant, ne pas l'utiliser dans un contexte d'interruption.

Les attributs correspondant à `IRQF_SHARED`, `IRQF_DISABLED`, et `IRQF_SAMPLE_RANDOM` étaient respectivement `SA_SHIRQ`, `SA_INTERRUPT` et `SA_SAMPLE_RANDOM` avant le noyau 2.6.17 inclus.

Lorsqu'une interruption est partagée entre plusieurs gestionnaires, ceux-ci seront appelés successivement. Chacun devra déterminer (en interrogeant son matériel) si l'interruption lui était effectivement destinée et renvoyer `IRQ_HANDLED` dans ce cas, ou `IRQ_NONE` sinon.

L'argument *ident* de `request_irq()` est un pointeur qui devra être fourni de manière identique à `free_irq()` pour identifier le gestionnaire à retirer dans le cas d'interruption partagée.

À vous : déclenchement d'interruptions par GPIO

Examinez, compilez puis chargez le module `exemple-v-03` sur le Raspberry Pi.

Vérifiez qu'à chaque front montant du GPIO d'entrée, la valeur sur le GPIO de sortie change d'état.

On peut utiliser un oscillateur externe ou un générateur de signal basse fréquence pour envoyer un signal périodique sur la broche GPIO d'entrée (ou sur la broche 10 du port parallèle).

Lorsqu'un front montant sera présent sur cette broche, une interruption sera levée. En utilisant le module de l'exemple ci-dessus, cette interruption fera basculer la broche (GPIO ou port parallèle) de sortie.

Si l'on visualise les deux signaux (entrée et sortie) sur un oscilloscope, on peut "zoomer" sur leurs fronts montants et mesurer le temps de prise en charge des interruptions.

Traitement d'interruption différé

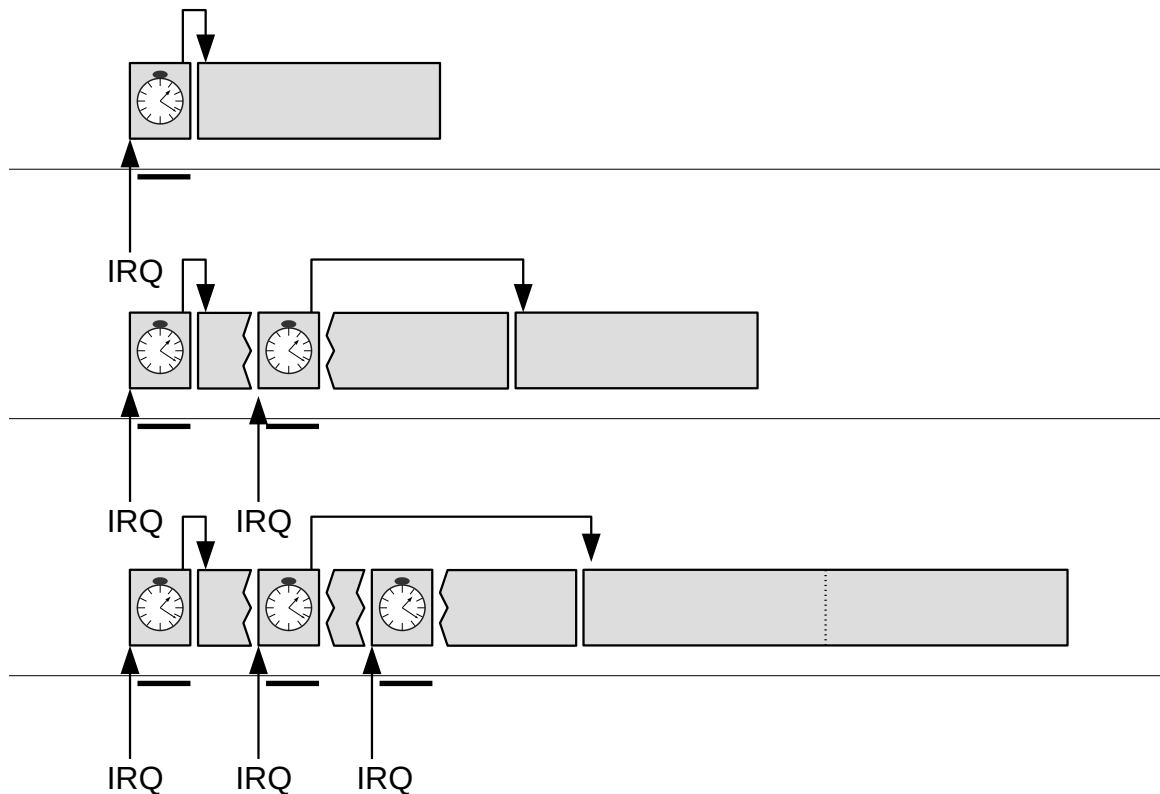
Problèmes avec les handlers d'interruptions monolithiques



Lorsque plusieurs occurrences de la même interruption se présentent en rafale rapide :

- La première occurrence est traitée correctement.
- La seconde occurrence n'est traitée que lorsque le *handler* se termine. Si elle doit faire une action urgente (par exemple un horodatage), celle-ci est réalisée en retard.
- La troisième occurrence est perdue.

Traitements différés par *Top-Half* / *Bottom-Half*



La partie urgente (horodatage) est réalisée dans la *Top-Half* immédiatement à la réception de l'interruption. Le traitement moins urgent est effectué dans la *Bottom-Half* alors que l'interruption peut à nouveau se déclencher.

Si plusieurs *Top-Halves* se déclenchent alors que la première *Bottom Half* ne s'est pas terminée, une seule *Bottom-Half* ultérieure sera exécutée. Il est du ressort du développeur de compter les occurrences des interruptions et réaliser le traitement correspondant autant de fois que nécessaire.

Il existe plusieurs mécanismes d'implémentation des *Bottom-Halves* : nous observerons les *tasklets*, les *workqueues* et les *threaded interrupts*.

Pour en savoir plus :

- « *Interruptions et tasklets* » <https://www.blaess.fr/christophe/2017/06/06/>

Exécution différée par tasklet

Travail différé dans une fonction du type :

```
void tasklet_function (unsigned long arg);
```

Initialisation d'une structure `tasklet_struct` associant la fonction et un argument :

```
<linux/interrupt.h>

DECLARE_TASKLET(name, function, argument);

void tasklet_init (struct tasklet_struct *tasklet,
                  void (*fonction) (unsigned long),
                  unsigned long data);
```

Programmation et annulation :

```
void tasklet_schedule (struct tasklet_struct *tasklet);

void tasklet_kill      (struct tasklet_struct *tasklet);
```

La fonction programmée avec **tasklet_schedule()** sera exécutée dès le retour du gestionnaire d'interruption (avec toutes les interruptions re-validées).

Si une *tasklet* est programmée à plusieurs reprises avant d'être exécutée, la fonction ne sera appelée qu'une seule fois. La fonction sera exécutée obligatoirement sur le processeur qui l'a programmée. Une *tasklet* s'exécute dans un contexte noyau semblable à celui d'un gestionnaire d'interruption. Il n'y a pas nécessairement de processus `current` valide, et elle ne doit pas dormir.

À vous...

Chargez le module `examp le-V-04` et vérifiez que le comportement soit identique au précédent.

Exécution différée par workqueue

Travail différé dans une fonction du type :

```
void wq_function (struct work_struct *);
```

Initialisation avec :

```
<linux/workqueue.h>  
  
DECLARE_WORK(name, function_wq);
```

Programmation et annulation avec :

```
void schedule_work (struct work_struct *name);  
  
void schedule_delayed_work (struct work_struct *name,  
                             unsigned long delay);  
  
void cancel_delayed_work (struct work_struct *name);  
  
void flush_scheduled_work (void);
```

La structure `work_struct` contient un champ `data` que l'on peut utiliser pour passer un argument à la fonction de travail différé. La *workqueue* est exécutée par un *thread* du noyau (`events`). Elle est donc ordonnancée, et peut appeler des fonctions bloquantes (sommeil autorisé).

En général, on préfère les *threaded interrupts* aux *workqueues* pour les nouveaux drivers.

À vous...

Chargez le module `example-V-05` et vérifiez le fonctionnement. Vérifiez qu'en exécutant simultanément une boucle temps-réel de haute priorité, le traitement en *workqueue* est différé jusqu'à la fin de la boucle.

Threaded interrupts

Il est possible d'installer un *handler* qui sera exécuté en tant que *thread* du noyau. Il sera alors possible de gérer des priorité temps-réel entre tâches et *handlers*.

```
int request_threaded_irq (unsigned int irq,
                          irq_handler_t top_half_handler,
                          irq_handler_t bottom_half_handler,
                          unsigned long flags,
                          const char *name, void *dev);
```

La fonction *top_half_handler* sera appelée en contexte d'interruption, elle doit déterminer si l'IRQ était destiné au *driver*, puis renvoyer :

- IRQ_NONE si l'IRQ n'était pas destinée au *driver* ;
- IRQ_HANDLED si l'IRQ a été complètement traitée par le *handler* en interruption ;
- IRQ_WAKE_THREAD s'il faut lancer le second *handler* en contexte de *thread* noyau.

La fonction *bottom_half_handler* doit renvoyer IRQ_NONE ou IRQ_HANDLED.

```
int devm_request_threaded_irq (struct device *dev, int irq,
                              irq_handler_t top_half,
                              irq_handler_t bottom_half,
                              unsigned long flags,
                              const char *name, void *dev);
```

Les paramètres *top_half_handler* et *bottom_half_handler* peuvent être NULL si on ne veut pas les implémenter.

À vous

Chargez le module `example-v-06`.

Le comportement est-il identique à celui des handlers précédents ?

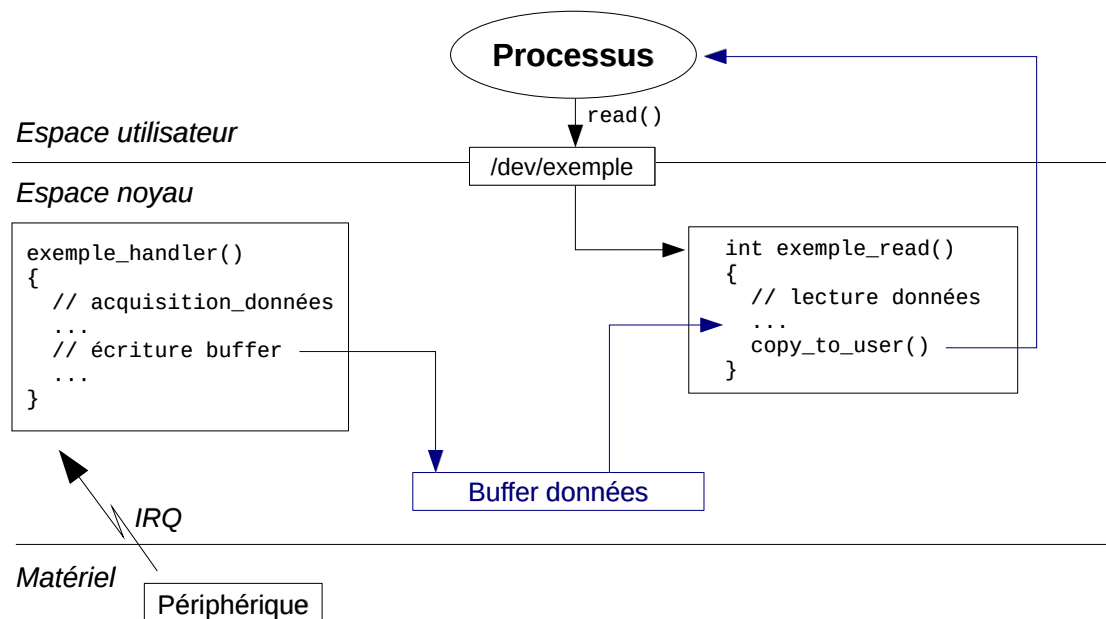
Voyez-vous le *thread* noyau avec la commande `ps` ?

Travaux pratiques : driver virtuel en mode caractère

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice V-1 : driver virtuel de file de messages

Protection des variables globales



Dans de nombreux drivers, des données sont acquises ou diffusées vers le périphérique dans une routine d'interruption (ou une fonction différée *bottom-half*). Pour transférer les données depuis ou vers les applications, on emploie des appels-système (`read()`, `write()`, `ioctl()`...).

Le transfert des données entre l'appel-système et le gestionnaire d'interruption se fait généralement par des variables globales du module. La synchronisation des accès est indispensable car l'interruption peut survenir n'importe quand – y compris durant l'appel-système.

L'utilisation des sémaphores ou mutex vus précédemment n'est pas possible car on ne peut pas dormir dans un contexte d'interruption. On emploie alors les *spinlocks*.

Protections par spinlock

```
<linux/spinlock.h>

int spin_lock_init (spinlock_t *spl);
```

Dans le gestionnaire d'interruption on emploie :

```
void spin_lock (spinlock_t *spl);
void spin_unlock (spinlock_t *spl);
```

Dans un appel-système on utilise :

```
void spin_lock_irqsave (spinlock_t *spl,
                        unsigned long mask);
void spin_unlock_irqrestore (spinlock_t *spl,
                             unsigned long mask);
```

ou si les variables sont utilisées en *bottom-half* :

```
void spin_lock_bh (spinlock_t *spl);
void spin_unlock_bh (spinlock_t *spl);
```

Le code de gestion des *spinlocks* est compilé différemment suivant la configuration du noyau : multi-processeurs, uni-processeur, préemptible ou non, etc.

Le *spinlock* lui-même est un verrou (comme un *mutex*), mais utilisant une attente active, sans sommeil. Son intérêt est d'offrir une gestion portable, correcte sur un système multi-processeurs de portions critiques non-interruptibles. Dans l'appel-système on utilise `spin_lock_irqsave()` qui verrouille le *mutex* et coupe toutes les interruptions sur le processeur local. Sur un système uni-processeur, le gestionnaire ne peut donc pas être appelé avant le `spin_unlock_irqrestore()`.

Sur un système multi-processeurs, le gestionnaire peut se déclencher sur un autre processeur que l'appel-système. Il restera en attente sur `spin_lock()` jusqu'à la fin de la portion critique de l'appel-système. Lorsque l'exclusion mutuelle désirée se situe entre un appel-système et une fonction différée (*tasklet*) on peut employer `spin_lock_bh()` qui interdit l'exécution des *bottom halves*.

À vous...

Chargez le module `example-V-07`. Déclenchez quelques interruptions et lisez le contenu du fichier spécial avec `cat`.

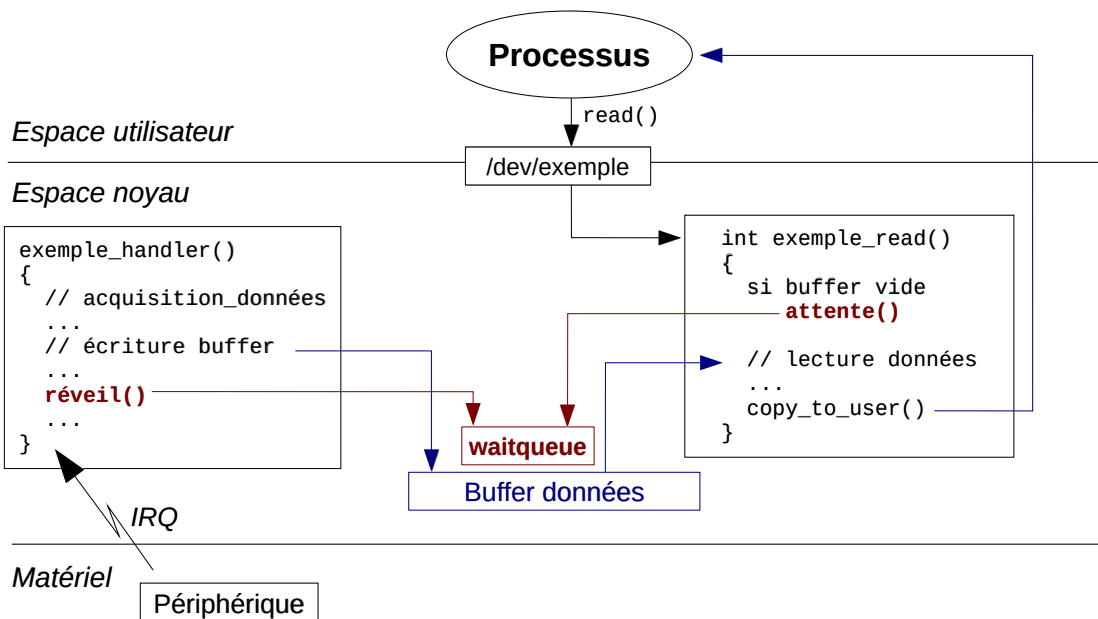
Travaux pratiques : synchronisation par mutex et spinlock

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice V-2 : synchronisation des accès à une file de messages

Attentes d'événements

Principe



Pour attendre passivement un événement externe on utilise le mécanisme des *waitqueue*.

On associe une *waitqueue* à une condition, par exemple "il y a des données dans le buffer de réception".

L'appel-système de lecture des données peut endormir volontairement le processus en attente sur la *waitqueue*. La tâche passe dans l'état *Sleeping* et l'ordonnanceur en active une autre.

Le gestionnaire d'interruption qui remplit le *buffer* peut demander le réveil d'une tâche (ou plusieurs) en attente sur cette *waitqueue*. La tâche est alors réveillée (mise dans l'état *Runnable*) et sera activée dès que l'ordonnanceur le décidera.

Utilisation d'une waitqueue

Déclaration de la *waitqueue* :

```
<linux/wait.h>

DECLARE_WAIT_QUEUE_HEAD(name);
```

Mise en sommeil :

```
void wait_event                (name, condition)
int  wait_event_interruptible (name, condition)

int wait_event_timeout          (name, cond, delay)
int wait_event_interruptible_timeout (name, cond, delay)
```

Réveil des tâches en attente :

```
void wake_up                    (wait_queue_head_t *name)
void wake_up_all                (wait_queue_head_t *name)
void wake_up_interruptible      (wait_queue_head_t *name)
void wake_up_interruptible_all (wait_queue_head_t *name)
```

La *condition* utilisée dans les macros `wait_event()` et `wait_event_interruptible()` est une expression qui sera évaluée avant de sortir de la boucle d'attente.

La fonction `wait_event()` attend en sommeil ininterrompue, généralement déconseillé.

Si `wait_event_interruptible()` est interrompue par un signal, elle renvoie une valeur non-nulle.

Les fonctions `wake_up_interruptible()` et `wake_up_interruptible_all()` ne réveillent que les tâches endormies dans des sommeils interrompibles. Les fonctions `wake_up()` et `wake_up_interruptible()` ne réveillent qu'une seule tâche parmi celles en attente, tandis que `wake_up_all()` et `wake_up_interruptible_all()` les réveillent toutes.

À vous...

Chargez le module `example-V-08`.

Vérifiez que `cat` soit bien bloquant en attente d'interruption. Dans la commande `ps`, on peut voir notre processus `cat` en sommeil interrompible (état S).

Opérations bloquantes et non-bloquantes

Dans l'espace utilisateur, l'attribut `O_NONBLOCK` placé sur un descripteur de fichier rend les opérations d'entrées-sorties non bloquantes.

Une telle opération échoue immédiatement avec l'erreur `EAGAIN` si son action est impossible.

```
int fd = open(filename, O_RDWR | O_NONBLOCK);  
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);  
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) & ~O_NONBLOCK);
```

Dans l'appel-système, on peut tester cet attribut dans le champ `f_flags` de la structure `file` transmise en premier argument.

À vous...

Chargez le module `example-V-09`.

Vérifiez si `cat` est toujours bloquant. Pourquoi ?

Utilisez le programme de test `cat-non-block.c` pour vérifier le fonctionnement correct du module.

Vous pouvez examiner également le module `example-V-10` qui réalise le même travail mais dont la structure est mieux adaptée aux bonnes pratiques du *kernel*.

Travaux pratiques : attentes d'événements

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

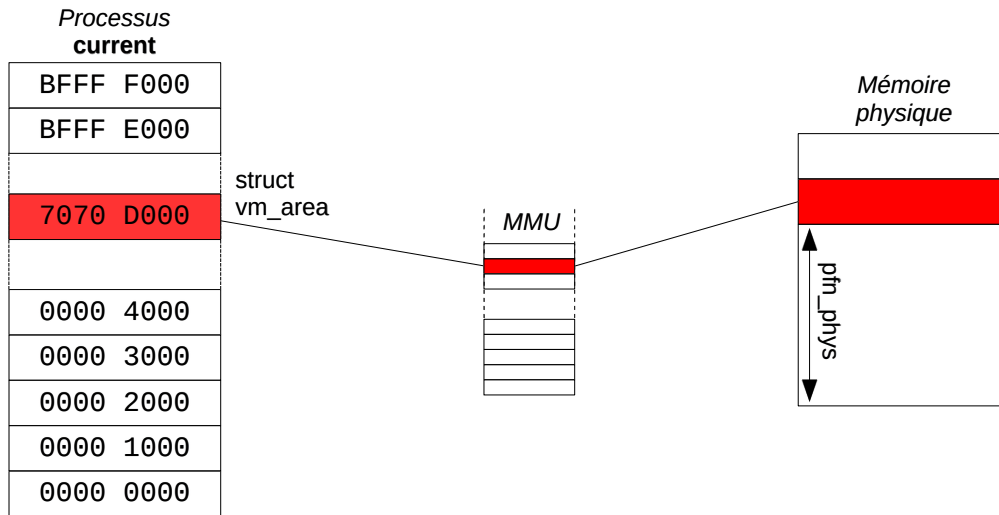
Exercice V-3 : méthode de lecture bloquante

Exercice V-4 : méthode d'écriture bloquante

Projection de mémoire

Appel-système mmap()

```
int remap_pfn_range (struct vm_area_struct *vma
                    unsigned_long virt_addr,
                    unsigned long pfn_phys,
                    unsigned long size,
                    pgprot_t permissions);
```



La fonction `remap_pfn_range()` projette une zone de mémoire physique dans une zone de mémoire virtuelle se trouvant dans l'espace *utilisateur* du processus *current*.

La structure `vm_area` décrit la zone d'adresses virtuelles dans laquelle établir la projection. Ses champs les plus utiles pour nous sont `vm_start` et `vm_end`.

À vous : implémentation d'une méthode *mmap()*

Chargez le module `example-V-11` qui projette un *buffer* dans la mémoire du processus qui invoque `mmap()`.

Utilisez l'outil `test-mmap.c` pour vérifier son fonctionnement.

Transferts de données par DMA

Le principe du DMA (*Direct Memory Access*) permet à un périphérique externe de lire ou d'écrire des données directement dans la mémoire physique parallèlement au CPU.

```
dma_addr_t dma_map_single (struct device *dev,
                           void *buffer,
                           size_t size, int direction);

void dma_unmap_single (struct device *dev,
                      dma_addr_t bus_addr,
                      size_t size, int direction);
```

direction : DMA_TO_DEVICE, DMA_FROM_DEVICE ou DMA_BIDIRECTIONAL.

```
void dma_sync_single_for_cpu (struct device *dev,
                              dma_addr_t bus_addr,
                              size_t size,
                              int direction);

void dma_sync_single_for_device (struct device *dev,
                                 dma_addr_t bus_addr,
                                 size_t size,
                                 int direction);
```

La fonction `dma_map_single()` autorise l'accès par le périphérique externe, suivant la *direction* indiquée au *buffer* visible à l'adresse physique (*bus*) renvoyée par la fonction.

Sur plusieurs architectures, l'utilisation de `DMA_BIDIRECTIONAL` en guise de *direction* est pénalisante en performances. Cette valeur est généralement déconseillée.

Le processeur ne doit pas accéder au *buffer* tant que la fonction `dma_unmap_single()` n'a pas été appelée. Si un accès temporaire est nécessaire, il faut l'encadrer par les appels `dma_sync_single_for_cpu()` et `dma_sync_single_for_device()`.

Il existe également des transferts en mode éparpillé (*scatter/gather*) que nous ne traiterons pas ici.