Interruptions

M2 SETI B4 / MS SE SE758

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

Plan

Introduction

- De très nombreux périphériques utilisent les interruptions pour signaler la survenue d'événements
  - Données disponibles, opération terminée, etc.
- Le pilote de périphérique doit pouvoir réagir lors du déclenchement d'une interruption en provenance d'un périphérique qu'il gère
- Un pilote peut donc mettre en place un gestionnaire d'interruption (voire même plusieurs), qui sera appelé automatiquement lorsque l'interruption survient

Enregistrement

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

```
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name,
                void *dev)

int devm_request_irq(struct device *dev, unsigned int irq,
                     irq_handler_t handler, unsigned long irqflags,
                     const char *devname, void *dev_id);
```

- Enregistrent un gestionnaire d'interruption
- `struct device *dev` : le périphérique concerné
- `unsigned int irq` : le numéro de l'interruption (souvent récupéré depuis l'arbre des périphériques, par exemple via le champ `irq` de la structure `struct i2c_client`)
- `irq_handler_t handler` : pointeur vers le gestionnaire d'interruption, c'est-à-dire la fonction appelée lorsque l'interruption survient (voir plus loin)

Enregistrement

#include <linux/interrupt.h>

```
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name,
                void *dev)

int devm_request_irq(struct device *dev, unsigned int irq,
                     irq_handler_t handler, unsigned long irqflags,
                     const char *devname, void *dev_id);
```

- `unsigned long irqflags` : drapeaux (voir liste dans `<linux/interrupt.h>`). Le plus courant est `IRQF_SHARED` qui indique que la ligne d'interruption est partagée entre plusieurs périphériques
- `const char *devname` : nom du périphérique (apparaîtra dans `/proc/interrupts`)

Enregistrement

#include <linux/interrupt.h>

```
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name,
                void *dev)

int devm_request_irq(struct device *dev, unsigned int irq,
                     irq_handler_t handler, unsigned long irqflags,
                     const char *devname, void *dev_id);
```

- `void *dev_id` : pointeur arbitraire qui sera passé en argument au gestionnaire d'interruption
  - Peut être `NULL` seulement si la ligne n'est pas partagée. Sinon, il doit être unique
  - Souvent pointeur vers la structure stockant les données liées au périphérique (`foo_device`) pour pouvoir facilement y accéder depuis le gestionnaire d'interruption

Enregistrement

#include <linux/interrupt.h>

```
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name,
                void *dev)

int devm_request_irq(struct device *dev, unsigned int irq,
                     irq_handler_t handler, unsigned long irqflags,
                     const char *devname, void *dev_id);
```

- Renvoient `0` si tout s'est bien passé, une valeur négative en cas d'erreur
- La variante `devm` permet de supprimer le gestionnaire d'interruption automatiquement lorsque le périphérique est déconnecté

Suppression

#include <linux/interrupt.h>

```
void free_irq(unsigned int irq, void *dev_id);

void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id);
```

- Supprime le gestionnaire d'interruption

Le gestionnaire d'interruption

```
irqreturn_t foo_handler(int irq, void *dev_id);
```

- Arguments
  - `int irq` : le numéro de l'interruption
  - `void *dev_id` : le pointeur passé à la fonction `request_irq`
- Valeur de retour
  - `IRQ_HANDLED` si l'interruption concernait ce périphérique et a été traitée (si la ligne d'interruption est partagée, les autres gestionnaires enregistrés ne seront pas appelés)
  - `IRQ_NONE` si l'interruption ne concernait pas ce périphérique (cas par exemple des lignes d'interruptions partagées entre plusieurs périphériques). Le noyau va donc exécuter les autres gestionnaires enregistrés pour cette interruption

Le gestionnaire d'interruption

Rôles classiques

- Acquitter l'interruption auprès du matériel (sinon, en général, elle se déclenche de nouveau tout de suite après)
- Si nécessaire transférer les données depuis le périphérique vers un tampon dans l'espace noyau (ou vice-versa)
- Si nécessaire, réveiller les tâches qui pourraient être en sommeil en attente de cet événement (appel à `wake_up*`)

Le gestionnaire d'interruption

Contraintes

- L'interruption peut survenir n'importe quand sur n'importe quel processeur
  - Le gestionnaire d'interruption s'exécute donc dans le contexte d'une tâche arbitraire
  - Cela n'a donc pas de sens de vouloir copier des données vers ou depuis l'espace utilisateur
- Pendant l'exécution du gestionnaire d'interruption, toutes les interruptions sont désactivées sur le processeur sur lequel s'exécute le gestionnaire
  - Il est donc important qu'il s'exécute rapidement sous peine d'une dégradation importante des performances et du temps de réponse du système

Le gestionnaire d'interruption
Contraintes (suite)

- Un gestionnaire d'interruption n'a pas le droit de mettre en sommeil la tâche courante ni de bloquer
  - Par conséquence, il doit impérativement utiliser le drapeau `GFP_ATOMIC` s'il alloue de la mémoire pour éviter que la fonction d'allocation ne bloque
  - Il ne peut pas non plus bloquer en demandant un *mutex* par exemple

Plan

Top half vs. *bottom half*

- La plupart des systèmes d'exploitation modernes découpent les traitements effectués suite à une interruption en deux parties
  - Le *top half* : le gestionnaire d'interruption lui-même qui répond à l'interruption et qui doit être le plus rapide possible. Il se contente en général d'acquitter l'interruption, de transférer les données et de programmer l'exécution ultérieure de la seconde partie, le *bottom half*
  - Le *bottom half* : cette seconde partie, qui s'exécute ultérieurement dans un contexte moins contraint, effectue si nécessaire des opérations sur les données (par exemple décodage) ou tout autre opération nécessaire suite à l'interruption

Bottom half

- Les opérations effectuées *bottom half* le sont dans un contexte moins contraint (en fonction du mécanisme : interruptions partiellement ou totalement activées, possibilité de mise en sommeil, etc.)
- On peut donc se permettre de faire des opérations plus longues ou complexes
- Plusieurs mécanismes sont disponibles dans le noyau pour implémenter le *bottom half*, c'est-à-dire l'exécution différée de certains opérations suite à une interruption
  - *Softirq*, non utilisable directement par un pilote
  - *Tasklet*, contexte d'interruption
  - *Workqueue*, contexte thread
  - *Threaded IRQ*, contexte thread

Tasklet

Déclaration et initialisation

- Représentée par une structure `struct tasklet_struct`
- Initialisée
  - Statiquement : `DECLARE_TASKLET(t, func, data)`
  - Dynamiquement :
    ```
    void tasklet_init(struct tasklet_struct *t,
    void (*func)(unsigned long), unsigned long
    data);
    ```
- `func` est la fonction qui sera exécutée par la *tasklet* et `data` un entier arbitraire qui lui sera passé en argument

Tasklet

Programmation

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t); /* High priority */
```

- Lorsque `tasklet_schedule` est appelée, on a la garantie que la *tasklet* sera exécutée sur un processeur au moins une fois dans le futur
- Si la *tasklet* est déjà programmée pour l'exécution mais que son exécution n'a pas encore débutée, elle ne sera exécutée qu'une seule fois
- Si la *tasklet* est déjà en cours d'exécution, elle est reprogrammée pour une exécution ultérieure
- La même *tasklet* ne peut s'exécuter que sur un processeur à un instant donné (il faut qu'elle se termine complètement pour qu'elle soit de nouvea exécutée sur le même processeur ou sur un autre)

Tasklet

Exécution

```
void foo_tasklet_func(unsigned long data);
```

- Les *tasklets* sont exécutées alors que les interruptions sont activées
- Elles peuvent donc se permettre de faire des traitements plus long quittes à être interrompues
- Néanmoins, elles s'exécutent toujours dans le contexte d'interruption et donc elles ne doivent pas bloquer ni mettre en sommeil la tâche courante
- C'est le mécanisme qui permet un délai minimal entre l'interruption (*top half*) et l'exécution du *bottom half*

Workqueue
Introduction

- Le mécanisme de *workqueue* est un mécanisme générique permettant d'exécuter de façon asynchrone une fonction
- Peut être utilisé comme *bottom half* mais pas uniquement
- Des threads appartenant au noyau se chargent d'exécuter les différents travaux (*work items*) au fur et à mesure de leur arrivée
- Il est possible, pour des questions de latence par exemple, de créer explicitement un thread dédié à l'exécution de certains travaux plutôt que d'utiliser les threads par défaut
- Documentation :
  `Documentation/core-api/workqueue.rst`

Workqueue

Déclaration d'un travail

- Il existe deux structures pour représenter un travail
  - `struct work_struct`
  - `struct delayed_struct` (travaux qui doivent attendre l'expiration d'un délai avant leur exécution)
- Initialisation statique
  - `DECLARE_WORK(ws, f);`
  - `DECLARE_DELAYED_WORK(ds, f);`
- Initialisation dynamique
  - `INIT_WORK(ws, f);`
  - `INIT_DELAYED_WORK(ds, f);`
- Travail : `void f(struct work_struct *work);`

Workqueue
Programmation

```
bool schedule_work(struct work_struct *work);
bool schedule_delayed_work(struct delayed_work *dwork,
                           unsigned long delay)
```

- Programment l'exécution d'un travail (i.e. l'exécution de la fonction pointée par la structure `work_struct`) sur un thread noyau
- S'il est déjà programmé, elles renvoient `false`
- Dans le cas de la seconde fonction, le travail ne débutera pas avant l'expiration du délai `delay` exprimé en *jiffies*
- Une variante `xxx_on(int cpu, ...)` de ces fonctions existe pour laquelle il est possible de choisir le processeur sur lequel le travail sera exécuté (optimisations)

Workqueue
Autres fonctions

```
bool cancel_work_sync(struct work_struct *work);
bool cancel_delayed_work(struct delayed_work *dwork);
```

- Annulent un travail dont l'exécution n'a pas encore débutée

```
work_pending(ws);
delayed_work_pending(ds);
```

- Indiquent si un travail est en attente d'exécution

Workqueue
Conclusion

- Les fonctions permettant de créer une *workqueue*, c'est-à-dire un thread dédié, ne seront pas détaillées, voir la documentation
- Les travaux, qui s'exécutent dans le contexte d'un thread noyau
  - Peuvent prendre du temps
  - Peuvent être mis en sommeil (s'ils sont exécutés sur un thread noyau dédié)

Threaded IRQ

Présentation

- Le mécanisme de *Threaded IRQ* est similaire à celui des *workqueues* mais spécialisé pour le traitement des interruptions
- Le *bottom half* est automatiquement programmé pour exécution suite à l'appel du *top half*
- Son exécution se fait au sein d'un thread noyau dédié pour le pilote

Threaded IRQ

Enregistrement

```
int devm_request_threaded_irq(struct device *dev, unsigned int irq,
                              irq_handler_t handler, irq_handler_t thread_fn,
                              unsigned long irqflags, const char *devname,
                              void *dev_id);
```

- Les arguments sont les mêmes que pour la fonction `devm_request_irq`, sauf
- `irq_handler_t handler` : la fonction gérant le *top half*. Deux cas peuvent se présenter (voir slides suivants)

Threaded IRQ

`irq_handler_t handler` (*top half*)

- Si la ligne d'interruption est partagée entre plusieurs périphériques
  - Ce *top half* est obligatoire
  - Il doit vérifier que l'interruption provient bien du bon périphérique
    - Si non, il doit renvoyer `IRQ_NONE`
  - Puis, si le *bottom half* doit être exécuté, il doit désactiver l'interruption au niveau du périphérique et renvoyer la valeur `IRQ_WAKE_THREAD` (pour éviter que le périphérique ne déclenche une autre interruption avant l'exécution du *bottom half*)
  - Si l'exécution du *bottom half* n'est pas nécessaire, il renvoie simplement `IRQ_HANDLED`

Threaded IRQ

`irq_handler_t handler` (*top half*)

- Si la ligne d'interruption est n'est pas partagée
  - Ce *top half* est n'est pas obligatoire
  - S'il existe, il doit se comporter comme décrit précédemment (renvoyer `IRQ_HANDLED` ou désactiver l'interruption et renvoyer `IRQ_WAKE_THREAD`)
  - Il est également possible de s'en passer (et donc de fournir la valeur `NULL` à l'argument `handler`)
    - Dans ce cas, le drapeau `IRQF_ONESHOT` doit être utilisé dans le champ `irqflags` pour que l'interruption reste désactivée jusqu'à la fin de l'exécution du *bottom half*
    - Quand l'interruption survient, le noyau la désactive automatiquement et programme l'exécution du *bottom half*

Threaded IRQ

`irq_handler_t thread_fn` (*bottom half*)

- L'argument `thread_fn` pointe vers la fonction contenant le code du *bottom half*
- Cette fonction sera exécutée par un thread noyau crée explicitement pour votre pilote
  - Elle peut donc prendre du temps
  - Elle peut donc effectuer des opérations qui mettront le thread en sommeil
- Elle doit renvoyer `IRQ_HANDLED` si son exécution s'est bien déroulée
- Si le *top half* a explicitement désactivé l'interruption au niveau du périphérique, le *bottom half* doit la réactiver à la fin de son exécution

Plan

Conclusion

- Les traitements à effectuer suite à une interruption sont divisés en deux parties
  - *Top half* : le gestionnaire d'interruption à proprement parler, contraintes importantes (doit prendre le minimum de temps, pas le droit de bloquer), donc doit faire le strict minimum
  - *Bottom half* : le reste des traitements, exécutés ultérieurement, avec des contraintes moins importantes
- Trois mécanismes utilisables par un pilote pour implémenter le *bottom half*
  - *Tasklet*, contexte d'interruption
  - *Workqueue*, contexte thread noyau (partagé ou exclusif)
  - *Threaded IRQ*, contexte thread noyau (exclusif)