Formation au Noyau Linux

Jérôme Pouiller <j.pouiller@sysmic.org>

Quatrième partie IV

L'API

Les frameworks

- Le noyau est divisé en frameworks.
- Pour bien utiliser un framework, il est nécessaire de comprendre le fonctionnement du noyau et de la technologie décrite par le framework.
- Un framework est généralement décrit dans son fichier `.h` (exemple : `linux/usb.h`).
- Un framework contient généralement une paire de fonctions `register`/`unregister` permettant au driver de se déclarer auprès du framework (Exemple : `usb_register_driver`)
- La fonction `register` prend souvent en paramètre une structure contenant des pointeurs sur des callbacks appelées lors des divers évènements pouvant se produire (exemple : la structure `usb_driver`)
- Parmi les callbacks, il y a souvent une fonction `probe`

Les frameworks

- Les extensions de gcc sont utilisées pour initialiser ces structures avec les attributs nommés
- Un framework fourni aussi d'autres fonctions permettant de lancer des actions (exemple : `usb_submit_urb`)
- Un framework peut raffiner un framwork plus générique (en le "proxifiant") (exemple : les framework usb_serial et le framework usb)
- Une driver peu appartenir à plusieurs frameworks. Par exemple, un driver de carte réseau peut utiliser le framework pci, le framework, network_device, sysfs, debugfs, etc...
- Il existe un framework pour les modules (`linux/modules.h`) accessible avec `THIS_MODULE`
- Un exemple de framework simple enrobant les *char devices* : `include/linux/cdev.h`

Fonctions standards de `string.h`

- Taille de chaines `strlen`, `strnlen`
- Comparaisons : `strcmp`, `strncmp`, `strcasecmp`, `strncasecmp`, `strnicmp`, `memcmp`, `strstarts`
- Recherches : `strchr`, `strnchr`, `memchr`, `memscan`, `strrchr`, `memchr_inv`, `strspn`, `strcpsn`, `strstr`, `strnstr`, `strpbrk`
- Copies : `strcpy`, `strncpy`, `memcpy`, `strcat`, `strncat`, `strsep`
- `strlcpy` et `strlcat` (De meilleures implémentations de `strncat` et `strncpy`). Toujours utiliser ces fonctions.
- `skip_spaces`, `strim`, `strstrip` : Supprime les blanc en début et fin de chaîne

Fonctions standards de `string.h`

- Convertions de nombres et de booléens : `strtobool`, `kstrto{int,uint,l,ul}` `kstrto{s64,u64,s32,u32,s16,u16,s8,u8}` `kstrto{s64,u64,s32,u32,s16,u16,s8,u8}_from_user` (existe aussi préfixé avec `simple_`, mais destiné à mourrir)
- Duplication avec allocation : `kmemdup`, `kstrdup`, `kstrndup`, `memdup_user`, `strndup_user` (nous verrons l'argument `gfp` un peu plus loin)
- Formatage : `sprintf`, `snprintf`, `sscanf`
- Utilitaire sur les nombres : `max`, `max3`, `min`, `min3`, `clamp`, `swap`
- Notez que beaucoup de ces fonctions possèdent des implémentations génériques mais peuvent être surchargées par architecture
- Référence : `linux/string.h` `linux/kernel.h`

Au sujet des accès aux E/S

La méthode d'accès aux E/S dépend de l'architecture :

- *Memory Mapped Input Output (MMIO)* : Registres mappés en mémoire. Il suffit de lire et écrire à une adresse particulière pour écrire dans les registres du périphérique. La méthode la plus répandue
- *Port Input Output (PIO)* : Registres accessible par un bus dédié. Instructions assembleurs spécifiques (`in`, `out`). Cas notable de l'architecture x86.

On déclare rarement les adresses des périphériques en absolu. On préférera définir les offsets à partir d'une base :

```
outb(base_device + REGISTER, 1);
```

ou

```
writeb(base_device->register, 1);
```

Fonctionnement en PIO

- On doit d'abord informer le noyau que l'on utilise une plage de ports avec

```
struct ressource *request_region(unsigned long
    start, unsigned long len, char *name);
void release_region(unsigned long start,
    unsigned long len);
```

- Référence : `linux/ioport.h`
- Permet d'éviter que deux drivers essayent de référencer la même plage de ports.
- Il est possible d'avoir des information sur les ports actuellement utilisés en lisant le fichiers `/proc/ioports`
- Une fois réservé, il est possible de lire/écrire sur les ports avec :

```
u\{8,16,32\} in\{b,w,l\}(int port);
void out\{b,w,l\}(u\{8,16,32\} value, int port)
    ;
```

- Ces fonctions s'occupent de convertir les données dans le bon

Fonctionnement en MMIO

- Il existe des fonctions identique pour les MMIO.

```
struct ressource *request_mem_region(unsigned
    long start, unsigned long len, char *name)
void release_mem_region(unsigned long start,
    unsigned long len)
```

- Il est possible de voir le mapping actuel dans le fichier `/proc/iomem`
- Il est ensuite nécessaire de faire appel au MMU pour mapper les IO dans l'espace d'adressage du noyau

```
void *ioremap(unsigned long phys_add, unsigned
    long size)
void iounmap(unsigned long phys_addr)
```

- Ces fonctions s'occupent de la cohérence avec le cache.

Fonctionnement en MMIO

- Sur les architectures simples, il possible de directement déréférencer le résultat de `ioremap`. Il existe néanmoins des fonctions apportant une couche d'abstraction (barrières mémoire, etc...).

```
u{8,16,32,64} read{b,w,l,q}(void *addr)
void write{b,w,l,q}(u{8,16,32,64}, void *addr)
```

- Vous devez utilisez ces fonctions :
  - les adresses ne sont pas volatiles
  - `write` et `read` s'occupent de placer des barrières et ou d'avoir des accès "volatiles"
  - il peut y avoir des problèmes de SMP ou de préemption gérés par `write` et `read`
  - certaines architectures ne supportent pas l'accès direct à la mémoire
  - il est plus facile pour sparse de détecter des erreur
  - Cas notable des accès PCI (little endian obligatoire)

Accèder aux IO en userspace

- `/dev/mem` est un fichier fichier device permettant d'accèder aux adresses physiques.
- Il est possible d'utiliser `mmap` sur ce fichier et d'écrire aux adresses occupées par des périphériques
- Sur les architectures utilisant PIO, il est possible de demander au noyau de laisser un processus utilisateur utiliser les instruction assembleurs `out*` et `in*` pour qu'ils puissent accéder aux périphérique avec `ioperm` et `iopl`.
- C'est ainsi que fonctionne les serveurs graphiques XFree86 ou xorg

Quelques notions

- Adresses physiques : Adresses physiques de la mémoire
- Adresses virtuelles : Adresses converties par le MMU
- Adresses logiques : Adresses dans l'espace du noyau mappée linéairement sur les adresses physiques
  - Elles peuvent être converties en adresses physiques en appliquant un simple masque de bits
  - Les adresses logiques sont des adresses virtuelles
  - Sur les architectures 32bits, les adresses logiques sont placées entre `0xC0000000` (3Go) et `0xFFFFFFFF`
  - Il est toutefois possible de modifier ce parametrage
  - Il est possible de convertir une adresse logique en adresse physique avec la macro `__pa`
  - ... et faire l'inverse avec la macro `__va`

Quelques notions

- Mémoire basse (*Low Memory*) : Partie de l'adressage virtuel contenant les adresses logiques. (voir `/proc/kallsyms` en root)
- Mémoire haute (*High Memory*) : Le reste de l'adressage virtuel
- *Out Of Memory (OOM) killer* : Mécanisme déclenché lorsque le système ne peut plus fournir de mémoire. Dans ce cas, un processus est désignée pour être *killé*. Il s'agit d'un algorithme heuristique

Les pages

Les pages sont l'unité de gestion du MMU.

- `PAGE_SIZE` indique la taille des page sur l'architecture courante
- Une page fait 4 à 8Ko.
- *Page Frame Number (PFN)* Le numéro de la page. Ce sont en fait les bits de poids fort des adresses.
- Il est facile de convertir les pages en *pfn* et inversement avec `page_to_pfn` et `pfn_to_page` (correspond à `page - CST` et `pfn + CST`)
- Il est possible d'obtenir l'adresse virtuelle d'une page avec la macro `page_address` (si elle est mappée dans l'espace d'adressage du noyau)

Les pages

- Il est possible d'allouer et de désallouer des pages avec :

```
struct page *alloc_pages(unsigned int flags,
    unsigned int order);
struct page *alloc_page(unsigned int flags);
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned
    int order);
```

- Les `flags` sont décrit un peu plus loin
- `order` indique le nombre de pages à allouer. Le noyau alloue 2^{order} pages
- `int get_order(unsigned long size)` retourne `order` nécessaire pour avoir `size` octets de mémoire
- Il existe des raccourcis pour allouer une nouvelle page et obtenir son adresse :

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags,
```

Le Slab

- Le Slab est l'allocateur de mémoire du noyau. C'est l'équivalent de l'allocateur de la libc.
- Il existe des alternatives telles que le Slub ou le Slob
- Il gère des *pool* d'espace mémoire.
- Il est possible d'obtenir des informations sur l'état du Slab en lisant le fichier `/proc/slabinfo`
- Il est possible d'allouer et de désallouer des espaces de mémoire avec le Slab en utilisant :

```
void *kmalloc(size_t size, gfp_t flags);
void kzalloc(size_t size, gfp_t flags);
void kzfree(const void *p);
void *kcalloc(size_t n, size_t size, gfp_t
    flags);
void kfree(const void *objp);
```

- On retrouve aussi l'équivalent de `realloc` :

```
void *krealloc(const void *p, size_t new_size,
    gfp_t flags);
```

Création de *pool* dans le Slab

Si vous avez de grande quantité d'objets identiques à gérer, ou si vous avez des besoins spécifiques, vous pouvez créer un pool spécialement pour vos besoins.

Le cas échéant, il est même possible d'utiliser vos propres fonctions d'allocation :

```
mempool_t *mempool_create(int min_nr,
    mempool_alloc_t *alloc_fn, mempool_free_t *
    free_fn, void *data);
void mempool_destroy(mempool_t *pool);
```

Vous pouvez garder la politique d'allocation par défaut en utilisant :

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

Référence : `linux/mempool.h`, `/proc/slabinfo`

Options d'allocation

Elles permettent de spécifier des contraintes sur l'allocateur. Les plus courantes sont :

- `GFP_KERNEL` : Standard
- `GFP_USER` : De la mémoire destinée à être donnée à un processus utilisateur. Elle est alors bien séparée des pages allouées pour le noyau.
- `GFP_HIGHUSER` : Indique en plus que l'allocation doit être faite dans la mémoire haute
- `GFP_ATOMIC` : L'allocateur ne peux pas bloquer durant l'allocation. Permet d'être utilisé dans des interruptions. Utilise le pool d'allocation d'urgence. Ne pas en abuser.
- `GFP_DMA` : Retourne un bloc contenu dans l'espace physique utilisable par les DMA
- Référence : `linux/gfp.h`

vmalloc

- Il est possible d'allouer de la mémoire non-contiguë avec `vmalloc` :

```
void *vmalloc(unsigned long size);
void *vzalloc(unsigned long size);
void vfree(void *addr);
```

- Alloue dans la mémoire haute
- ... ainsi la mémoire allouée peut être non contiguë dans la mémoire physique
- Il est ainsi possible d'allouer d'importante quantité de mémoire.
- Référence : `linux/vmalloc.h`, `mm/vmalloc.c`, `/proc/vmallocinfo`

Conversions

Il est possible de faire des conversions en utilisant :

- `phys_to_virt` et `virt_to_phys` (qui appellent `__pa` et `__va` si possible)
- `page_to_virt`, `virt_to_page`, `phys_to_page` et `page_to_phys` (qui appellent `page_address`)
- `pfn_to_page` et `page_to_pfn`

La mémoire des processus

- Copier des données depuis/vers l'espace d'adressage du processus courant :

```
int copy_from_user(kern_buf, user_buf, size);
int copy_to_user(user_buf, kern_buf, size);
```

- Retourne le nombre d'octets non copiés
- Lire/affecter une variable simple (jusqu'à 8 octets) de l'espace d'adressage du processus courant :

```
get_user(var, user_ptr);
put_user(var, user_ptr);
```

- La taille de `var` est automatiquement calculée

La mémoire des processus

- La variable `current` pointe sur le processus courant
- Chaque processus possède une table de mapping mémoire
- Le mapping du processus courant se trouve dans `current->mm`
  - `current->mm->rb_tree` contient des *virtual memory area (vma)* sous la forme d'un arbre rouge-noir
  - Il est possible d'utiliser la fonction `find_vma(mm, addr)` pour retrouver une VMA
  - allouer de la mémoire consiste à allouer une structure `vm_area_struct` et l'ajouter dans l'arbre rouge-noir
  - Le noyau parle ensuite *to pin* une page en mémoire lorsque celle-ci doit être réellement allouée
- Il est possible d'obtenir les mapping d'un processus en lisant les fichier `/proc/<PID>/map`, `/proc/<PID>/smap`, `/proc/<PID>/pagemap`
- Référence : `Documentation/vm/pagemap.txt`, `Documentation/filesystems/proc.txt`, `linux/mm.h`

Mapper des adresse physiques

Parmi les fonctions les plus utiles, `remap_pfn_range` permet de mapper des adresses physiques dans un espace utilisateur. Elle est particulièrement utile pour implémenter l'appel `mmap` :

```
int remap_pfn_range(
  struct vm_area_struct *vma,
  unsigned long addr,
  unsigned long pfn,
  unsigned long size,
  pgprot_t flags);
```

- `vma` Espace d'adressage virtuel dans lequel le mapping doit s'effectuer. Dans le cadre de mmap, fournie par le kernel.
- `addr` L'adresse à l'intérieur de `vma` (=offset)
- `pfn` La page physique à mapper
- `size` Taille de l'intervalle à mapper
- `flags` Flags éventuels (reprendre `vma->vm_page_prot`)

Il est aussi possible de surcharger l'attribut `vm_operations_struct *vm_ops` de la `vm_area_struct` afin

Sparse

- Sparse est un outils spécifique au noyau
- Sparse parse le code et interprète certains attributs non définis lors de la compilation avec `gcc`
- `bitwise` indique que deux types ne peuvent pas être castés (exemple : `phys_addr_t` et **`long unsigned`**)
- `address_space(NUM)` précise le domaine d'un pointeur et empêche les pointeurs de différent domaines d'être affecté entre eux :
  - `address_space(0)` Kernel
  - `address_space(1)` User
  - `address_space(2)` MMIO
- Lancer vos compilation avec l'option `C=1` pour activer Sparse
- Référence : `linux/compiler.h`, *sparse(1)*

Interruptions

Pour demander la gestion d'une interruption :

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

- `irq` : Le numéro de l'IRQ
- `handler` La fonction à apeller.
- `flags` Principalement :
  - `IRQF_SHARED` : L'interruption peut être partagé
- `devname` : Un descriptif pour `/proc/interrupts`
- `dev_id` : Pointeur qui sera passé en paramètre de `handler`. On utilise généralement un pointeur sur une structure décrivant l'instance du device. Ce pointeur à aussi son importance lors de la libération de l'interruption.

Interruptions

- Il est possible d'obtenir la liste des interruptions enregistrées dans `/proc/interrupts`

On libère l'interruption avec :

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

- `irq` : Numéro d'irq
- `dev_id` : Instance à supprimer. `dev_id` doit donc être unique pour chaque IRQ

Le PIC

Une fois allouée, il est possible de demander au système (au *Programmable Interrupts Controller (PIC)* en fait) d'activer ou non l'interruption avec :

```
void enable_irq(unsigned int irq)
void disable_irq(unsigned int irq)
```

Note : Les PIC sont des périphériques comme les autres. Nous n'expliquons pas spécifiquement ici comment développer un driver pour un PIC.

Référence : `linux/interrupts.h`

Les handlers d'interruption

Les gestionnaires (*handlers*) d'interruption :

- Il est de type `irqreturn_t (*)(int irq, void *dev_id)`
- Il doit retourner (`irqreturn_t`)
  - `IRQ_HANDLED` si l'interruption a bien été gérée
  - `IRQ_NONE` dans le cas inverse. L'interruption est alors passée au handler enregistré suivant si l'interruption est partagée
- Il doit acquitter l'interruption sur le périphérique
- Il ne doit pas être bloquant :
  - pas de `wait_event`
  - pas de `sleep`
  - allocation de mémoire avec `GFP_ATOMIC`
  - attention aux sous-fonctions, vérifier qu'elles sont bien *interrupt compliant*
- Le maximum de traitement doit être reporté dans le *Bottom Half (bh)*

Softirq

- Les softirqs sont exécutés dans l'environnement des interruptions mais alors que les interruptions sont activées
- Elles n'empêchent pas le déclenchement des interruptions mais ne sont pas ordonnancées avec les tâches
- Par conséquent, quasiment les mêmes règles que les interruptions s'appliquent
- Les Softirq sont réservés aux tâche demandant une fréquence de traitement importante. Les développeurs du kernel gardent le nombre de Softirq en quantité limitée :

```
HI_SOFTIRQ,      BLOCK_IOPOLL_SOFTIRQ,
TIMER_SOFTIRQ,   TASKLET_SOFTIRQ,
NET_TX_SOFTIRQ,  SCHED_SOFTIRQ,
NET_RX_SOFTIRQ,  HRTIMER_SOFTIRQ,
BLOCK_SOFTIRQ,   RCU_SOFTIRQ,
```

- Les SoftIrq Hi et Tasklet sont des multiplexeurs permettant d'exécuter d'autres tâches
- Référence : `linux/interrupts.h`

Tasklets

- Exécutés par les SoftIrq Hi et Tasklet
- Hi est exécuté avec la priorité la plus haute parmi les SoftIrq, alors que les Tasklets ont quasiment la priorité la plus basse
- Pour initialiser et dés-enregistrer un tasklet :

```
DECLARE_TASKLET(name, void (*func)(unsigned
    long), unsigned long data);
void tasklet_init(struct tasklet_struct *t,
    void (*func)(unsigned long), unsigned long
    data);
void tasklet_kill(struct tasklet_struct *t);
```

Tasklets

- Pour demander l'exécution d'un Tasklet (dans le SoftIRQ Tasklet et dans le SoftIRQ Hi)

```
void tasklet_schedule(struct tasklet_struct *t)
void tasklet_hi_schedule(struct tasklet_struct
    *t)
```

- Si la tasklet était déjà prévue pour être exécutée, elle ne sera exécutée qu'une fois.
- Si la tasklet est déjà en cours d'exécution, elle sera ré-exécutée (mais pas simultanément, même sur un SMP).
- Référence : `linux/interrupts.h`

Les kthreads

- Equivalent à des threads en espace utilisateur
- Créer une kthreads :

```
struct task_struct *kthread_create(int (*
    threadfn)(void *data), void *data, const
    char namefmt[], ...)
```

- Marquer la tâche comme devant être être ordonnancée (Fonctionne avec toute les tâches, pas spécifique aux kthreads)

```
wake_up_process(struct task_struct *k)
```

- Il est possible de faire `kthread_create` et `wake_up` en un seul appel :

```
struct task_struct *kthread_run(int (*threadfn)
    (void *data), void *data, const char namefmt
    [], ...)
```

Les kthreads

- Affecter une kthread à un CPU

```
void kthread_bind(struct task_struct *k,
    unsigned int cpu)
```

- Tuer une kthread :

```
int kthread_stop(struct task_struct *k)
```

- Ne pas trop abuser des kthreads. Les calculs s'effectuent normalement dans les SoftIRQ et surtout dans les appel système des processus. Le cas échéant, préférez l'utilisation des Workqueue génériques. Enfin, posez-vous la question si l'action doit être effectuée dans le noyau ou dans l'espace utilisateur
- Référence : `linux/kthread.h`

Workqueues

- Mécanisme général pour repousser un calcul
- Permet d'ordonnancer des des tâches dans une kthread
- Exemple d'utilisation des Workqueues : flush des caches des disques, défragmentation en arrière-plan, etc...
- `INIT_WORK(work, func)` permet de déclarer une nouvelle structure `work` exécutant `func`
- `int schedule_work(struct work_struct *work)` permet de demander l'exécution de `work`
- `int schedule_work_on(int cpu, struct work_struct *work)` spécifie un CPU particulier sur lequel le `work` doit s'exécuter
- `int schedule_delayed_work(struct delayed_work * work, unsigned long delay)` démarre `work` après un certain délais. Permet de faire des tâches périodiques.
- `bool cancel_delayed_work(struct delayed_work * work)` Annule une tâche délayée

Workqueues

- Il est possible d'utiliser d'autre workqueue (= d'autres kthreads) que celle de `schedule_work`
- `int queue_work(struct workqueue_struct *wq, struct work_struct *work)` permet de demander l'exécution de work en spécifiant la workqueue.
- Il est aussi possible de créer ses propres workqueues avec `alloc_workqueue(name, flags, max_active)`
- Référence : `linux/workqueue.h`

Les Wait Queues

- Permet d'attendre de manière passive un évènement.
- Peut-être utilisé dans le contexte d'une tâche ou dans une kthread.
- Cela permet de faire passer la tâche associée de l'état *run* à l'état *wait*
- Fonctionne de manière similaire à des `pthread_cond` ou des `rt_event` sous Xenomai ou les `OSFlags` sous μC/OS-II
- Pour déclarer :
  - `DECLARE_WAIT_QUEUE_HEAD(my_queue)` : initialisation lors de la déclaration
  - `wait_queue_head_t my_queue;` `init_waitqueue_head(&my_queue)` : Déclaration et initialisation séparés

Les Wait Queues

Pour attendre un évènement :

- Attend un évènement sur la queue ET que la condition soit vérifiée

```
void wait_event(queue, condition);
```

- Idem mais retourne une erreur si exécuté dans un appel système et le processus est tué avec `SIGKILL`

```
int wait_event_killable(queue, condition);
```

- Idem mais retourne une erreur si exécuté dans un appel système et le processus reçoit un signal

```
int wait_event_interruptible(queue, condition);
```

- Idem `wait_event` mais avec un timeout

```
void wait_event_timeout(queue, condition);
```

- `wait_event_timeout` + `wait_event_interruptible` :

```
void wait_event_interruptible_timeout(queue
```

Les Wait Queues

Pour réveiller les tâches en attentes

- Réveil tous les processus en attente sur la queue

```
wait_up(queue);
```

- Idem mais, seulement les interruptibles

```
wait_up_interruptible(queue);
```

Coherent mapping

- Gère un mapping “cohérent” accessible depuis le périphérique et le CPU

```
void *dma_alloc_coherent(
  struct device *dev, /* Device sur lequel est
      mappé le DMA. Fourni par le framework */
  size_t size,    /* Taille du mapping */
  dma_addr_t *handle, /* Adresse physique à
      fournir au device */
  gfp_t gfp)      /* Flags habituels */
```

- `dma_alloc_coherent` retourne une adresse virtuelle pour accéder au mapping par le CPU.
- La libération du buffer se fait par :

```
void dma_free_coherent(struct device *dev,
    size_t size, void *cpuaddr, dma_handle_t
    bus_addr);
```

Streaming mapping

- Dans ce mode, le driver doit allouer lui-même la mémoire (utilisation de `GFP_DMA`).
- Il faut configurer le cache du CPU que la zone mémoire puisse être utilisé pour un DMA :

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

- `direction` peut être `DMA_TO_DEVICE`, `DMA_FROM_DEVICE`, `DMA_BIDIRECTIONAL`, `DMA_NONE`
- Le cache n'est pas cohérent. Pour accéder au buffer, il faut apeller :

```
void dma_unmap_single(struct device *dev,
    dma_addr_t bus_addr, size_t size, enum
    dma_data_direction direction)
```

- Cet appel doit être fait après que le périphérique ait terminé ces accès au buffer

Les listes

- Implémentation assez singulière de listes génériques en C.
- utilisation de la macro `container_of` qui permet de retourner un pointeur sur l'objet contenant en connaissant l'adresse d'un attribut et et le type :

```
#define container_of(ptr, type, member) ({ \
  const typeof( ((type *)0)->member ) *__mptr =
       (ptr); \
  (type *)( (char *)__mptr - offsetof(type,
     member) );})
```

- Pour déclarer une liste :
  - Déclarer un type pour les noeuds de sa liste
  - Ajouter un attribut de type `struct list_head` à son type
- Une liste vide est alors composé d'une simple `struct list_head` à déclarer avec `LIST_HEAD(my_var)`
- Pour les fonction d'ajout, suppression, etc... tout est géré avec les `list_head`
- Pour accéder au données liées à un `list_head`, il faut utiliser la macro `list_entry` (qui est en fait la même chose que

Les listes

Exemple :

```
#include <linux/list.h>
stuct my_node_t {
  struct list_head node;
  /* other attributes */
}
void f() {
   LIST_HEAD(my_list);
   struct my_node_t new_node;
   struct my_node_t *i;
   printf("%d\n", list_empty(my_list));
   list_add(&new_node->node, &my_list);
   printf("%d\n", list_size(my_list));
   list_for_each_entry(i, &my_list, node) {
   }
}
```

Les arbres

- Il existe deux implémentations principales d'arbre dans le noyau
- Les *rbtree* (arbres rouges-noirs)
- Fonctionnent aussi à base de `container_of` mais nécessite une plus grande part de personnalisation que les listes.
- En particulier, nécessite l'implémentation des fonction de recherche et d'ajout.
- Cette architecture permet de définir sa propre fonction de comparaison sans impacte sur les performance et en maintenant une certaine généricité
- Références : `Documentation/rbtree.txt` et `linux/rbtree.h`
- Les *radix priority search tree* (arbres préfixes ou `prio_tree`)
- Uniquement utilisé pour référencer les structures `vma`
- Références : `Documentation/prio_tree.txt` et `linux/prio_tree.h`

Les fifo

- Apellé aussi *Buffer Circulaire*
- L'une des structure les plus utilisée dans le noyau
- Permet la communication avec les interruptions (pas de pause lors des accès) et entre les processus
- Pendant longtemps, chaque driver avait son implémentation de fifo
- Il existait un début d'interface commune dans `linux/circ_buf.h`
- Il existe maintenant une implémentation de référence : *kfifo*

kfifo

- Déclarer et initialiser une kfifo :

```
DEFINE_KFIFO(fifo, type, size)
```

- `fifo` : Nom de la fifo
- `type` : Type des objets à contenir
- `size` ; Taille de la fifo
- Pour pousser, tirer et lire des éléments :

```
int kfifo_put(fifo, val)
int kfifo_get(fifo, val)
int kfifo_peek(fifo, val)
```

kfifo

- Pleins d'autres fonctions utiles :

```
kfifo_recsize(fifo)
kfifo_reset(fifo)
kfifo_size(fifo)
kfifo_len(fifo)
kfifo_is_full(fifo)
kfifo_is_empty(fifo)
```

- Il est possible de coupler une kfifo avec `wait_queue`
- Il est possible de déclarer une kfifo avec un espace mémoire déporté de la structure et ainsi utiliser des espace mémoire spéciaux.

Mutex

Précautions classiques à l'utilisation des sections critiques :

- Acquérir le mutex le plus tard possible, Relâcher le plus tôt
- Etudier la granularité nécessaires aux sections critiques
- Attention aux bug classiques : latences, dead locks, inversements de priorités

Mutex

Méthodes assez classiques des mutex :

- Déclarer et initialiser un mutex :

```
DEFINE_MUTEX(name);
void mutex_init(struct *mutex);
```

- Acquérir le mutex. Attention, si appelé depuis un processus, celui-ci ne peut plus être tué

```
mutex_lock(struct mutex *lock);
```

- Idem, mais le processus peut-être tué

```
mutex_lock_killable(struct mutex *lock);
```

- Idem, mais le processus peut être interrompu avec un signal

```
mutex_lock_interruptible(struct mutex *lock);
```

Mutex

- Idem, mais retourne immédiatement si le mutex est déjà locké

```
mutex_trylock(struct mutex *lock);
```

- Retourne 1 si le mutex est locké

```
mutex_is_locked(struct mutex *lock);
```

- Unlock le mutex

```
mutex_unlock(struct mutex *lock);
```

Référence : `linux/mutex.h`

Autre mécanismes

Dans la même branche, le noyau propose aussi :

- Des sémaphores (référence : `linux/semaphore.h`)
- Des *Read-Write Lock* (référence : `linux/rw_semaphore.h`)
- Mutex avec protocole d'héritage de priorités et détection de dead lock (aka `rt_mutex`) (référence : `Documentation/rt-mutex-design.txt`, `linux/rtmutex.h`)
- Les `rt_mutex` peuvent être utilisé par la `libpthread`
- référence : `Documentation/pi-futex.txt` `Documentation/rt-mutex.txt`

Désactivation des interruption et de la préemption

- Lors de certains accès, les mécanismes de protections de ressources inter-tâches ne sont plus suffisants.
- Cela peut concerner des accès concurrent avec des interruptions.
- Il est alors nécessaire de désactiver temporairement les interruptions afin de garantir que l'on ne sera pas interrompu.
- Ces fonctions sont à utiliser avec parcimonie

Désactivation des interruption et de la préemption

- Sauve/restaure l'état des interruptions dans/de flags et désactive les interruptions :

```
local_irq_save(unsigned long flags);
local_irq_restore(unsigned long flags);
```

- Désactive/Active les interruptions (à priori, toujours utiliser les versions `_save` et `_restore`)

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

- Retourne vrai si les interruption du CPU local sont désactivées :

```
local_irq_disabled();
```

Désactivation des interruption et de la préemption

- Désactive/Réactive les SoftIrq (bottom halves) :

```
local_bh_disable();
local_bh_enable();
```

- Désactive/Réactive la préemption

```
preempt_disable();
preempt_enable();
```

Spin Lock

Dans certains cas, une tâche peut avoir besoin d'un accès exclusif à une ressource potentiellement utilisée dans une interruption. Dans ce cas, il est nécessaire de désactiver les interruptions lors de l'accès à la ressource. Néanmoins, l'interruption peut se produire sur un autre CPU. Il est alors nécessaire de se protéger contre cette éventualité

- Il s'agit d'une attente active sur une section critique
- Ne traite pas les problèmes de concurrence mais de parallélisme uniquement. Par conséquent, uniquement en environnement SMP
- Principalement utilisé lors de en complément de la désactivation des interruption
- La préemption est aussi désactivé durant un spin lock
- Le noyau propose aussi des *read/write spin locks* : `rwlock_t`
- Référence `linux/rwlock.h`

Spin Lock

API :

- Déclarer et initialiser un spin lock :

```
DEFINE_SPINLOCK(lock);
spin_lock_init(spinlock_t *lock);
```

- Idem que les fonctions `mutex_*` :

```
spin_lock(spinlock_t *lock);
spin_trylock(spinlock_t *lock);
spin_unlock(spinlock_t *lock);
```

Spin Lock

- Lock et désactive/réactive les interruptions sur le processeur courant. Les flags contiennent l'état du masque d'interruption :

```
spin_lock_irqsave(spinlock_t *lock, unsigned
    long flags);
spin_unlock_irqrestore(spinlock_t *lock,
    unsigned long flags);
```

- Spinlock et désactive/réactive les SoftIrq :

```
spin_lock_bh(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
```

Référence : `linux/spinlock.h`

Read-Copy-Update (RCU)

Type d'algorithme non bloquant :

- La lecture n'est pas bloquante
- On note le nombre de lecteurs
- Les modification s'effectuent sur une copie de l'objet
- Les lecture suivante se font sur la nouvelle version de l'objet
- Lorsque le dernier lecteur a terminé, l'objet d'origine est détruit. Seul subsiste la nouvelle version.
- Les RCU sont un design d'architecture. Les RCU ne possèdent que peu d'API pouvant être réutilisé.
- Référence `Documentation/RCU/whatisRCU.txt` `Documentation/RCU/*`

Exemple : Manipulation de listes

```
typedef struct {
  struct a_t a;
  int count_usage = 0;
  bool obsolete = false;
} rcu_t;
rcu_t *a = malloc(sizeof(rcu_t));
```

```
void read_a() {
 // lock:
 rcu_t *ptr = a;
 ptr->count_usage++;
 // do something with ptr;
 // unlock:
 ptr->count_usage--;
 if (ptr->obsolete && !ptr->
      count_usage)
   free(ptr);
}
```

```
void write_a() {
  struct rcu_t *a3 = a;
  struct rcu_t *a2 = malloc(
       sizeof(rcu_t));
  memcpy(a2, a);
  // modify a2;
  a = a2;
  a3->obsolete = true;
  if (!a3->count_usage)
    free(ptr);
}
```

les barrières mémoire

- Le compilateur et le CPU peuvent optimiser du code en réordonnancant les instructions.
- Le compilateur et le CPU ne réordonnance que les instructions indépendantes (donc, a priori, sans danger)
- Ce comportement peut néanmoins introduire des erreurs sur les système SMP ou lors d'accès à certains périphériques.
- Il est assez complexe de savoir ou doivent se placer les barrières. Elle interviennent néanmoins dans un certains nombre de structures de données ou d'algorithme
- Les barrière sont implémentées en utilisant des instructions assembleur particulières
- `barrier()` est un barrière pour le compilateur. Toutes les instruction avant la barrière seront placées avant les instructions apès la barrière
- Toutes les barrières CPU impliquent une barrière compiler. Du coup `barrier()` est peu utilisé.

les barrières mémoire

- Utilisation de barrières :
  - `mmiowb()` Empêche le réordonnancement des accès en écriture à la mémoire mappée sur IO
  - `rmb()` Barrière en lecture. Toutes les lecture demandées avant la barrière sont terminée avant `rmb()` et aucun lecture demandées après la barrière n'est commencée avant `rmb()`
  - `wmb()` Barrière en écriture
  - `mb()` Lecture et écriture
  - `smp_wmb()`, `smp_rmb()` et `smp_mb()` ont un comportement identique, mais ne sont présentent que si le système est compilé en SMP
  - Il existe d'autre types de barrières très spécifiques
- Tous les mécanismes de protections de ressources partagé contiennent déjà des barrières correctement placées (ça tombe bien, c'est la que c'est le plus complexe)
- Référence : `Documentation/memory-barriers.txt`