Construcciones del Kernel y Funciones de ayuda

Mg. Ing. Gonzalo E. Sanchez
MSE - 2022

Implementación de Manejadores de Dispositivos

Construcciones del Kernel y Funciones de ayuda

- Macro container_of
- Listas enlazadas
- Mecanismo sleep
- Administración de delay y timers
- Kernel locking
- Work queue

Macro container_of

Macro container_of

- El kernel es una porción de código que no depende de ningún otro módulo, biblioteca C o software de terceros.
- Implementa muchos mecanismos que se encuentran en otras bibliotecas para poder lograr funcionalidades.
- Entre ellas existen funcionalidades para cadena de caracteres, compresión e impresión en pantalla entre otras.
- Existen muchas macros y funciones, una muy utilizada es la macro **container_of**

Macro container_of

- Cuando se tienen muchas estructuras de datos en el código, es usual tener estructuras conteniendo otras.

- Existen ocasiones en que se tienen punteros a miembros de las estructuras, pero no a la estructura que contiene el puntero.

- Es usual tener que recuperar esos datos en cualquier momento sin que se consulten offsets en memoria.

Macro container_of

- EJEMPLO: Se tiene una estructura *persona* y su instancia **p**.

```
struct persona  {
        int edad;
        int salario;
        char *nombre;
}p;
```

Macro container_of

- Si se tiene un puntero a los miembros **edad** o **salario** se puede recuperar la estructura que los contiene.
- La macro **container_of** se utiliza para esto: encontrar la estructura contenedora de un campo miembro determinado.
- Esta macro está definida en *include/linux/kernel.h*

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

Macro container_of

- **NOTAS**:
  - El código para la versión 5.10 es un poco diferente. Se agrega una macro de assert para tiempo de compilación y *__mptr* es de tipo void.
  - **typeof** es una keyword del compilador GCC (ver uso).
  - **offsetof** es una macro definida en */linux/stddef.h*
  - En la expresión *((type *)0)->member* se utiliza un puntero a NULL para hacer válida la sintaxis de **typeof**.
  - Esto último se resuelve a tiempo de compilación, por lo que no hace falta que sea una dirección válida. Explicación en StackOverflow (link)

Macro container_of

- A fin de cuentas la macro se lee de la siguiente manera:

```
container_of(puntero, tipo_contenedor,
miembro_contenedor)
```

- **puntero** es el puntero al miembro de la estructura
- **tipo_contenedor** es el tipo de estructura contenedora
- **miembro_contenedor** es el nombre del campo al que está apuntando puntero dentro de la estructura contenedora.

Macro container_of

- EJEMPLO: Uso de macro container_of.

```
struct persona  {
        int edad;
        int
salario;
        char
*nombre;
};
```

```
struct persona somebody;
[.........]
int *edad_ptr =
&somebody.edad
```

Macro container_of

- Teniendo el puntero al miembro **edad** de la instancia **somebody** se puede obtener un puntero a **somebody** usando **container_of**.

```
struct persona *una_persona;
una_persona = container_of(edad_ptr, struct persona,
edad);
```

- **container_of** toma el offset correspondiente al campo **edad** desde el principio de la estructura **persona**.

Macro container_of

- **ATENCIÓN**: la macro **container_of** no funciona para miembros array.

- Esto significa que el primer argumento de **container_of** no puede ser un puntero a otro puntero (ver implementación).

- Esto implica que en el ejemplo anterior, no se podría utilizar el miembro **nombre** dado que es un array de char.

Macro container_of

- La macro **container_of** se utiliza en general para contenedores genéricos en el kernel.

- Su utilización se ve en las implementaciones de Platform Device Drivers.

Macro container_of

- EJEMPLO: Uso de macro container_of.

```
struct mcp23016 {
    struct i2c_client *client;
    struct gpio_chip chip;
}

/* retrieve the mcp23016 struct given a pointer 'chip' field
*/
static inline struct mcp23016 *to_mcp23016(struct gpio_chip
*gc)  {
    return container_of(gc, struct mcp23016, chip);
}
```

Macro container_of

- EJEMPLO: Uso de macro container_of (continuación).

```
static int mcp23016_probe(struct i2c_client *client, const struct i2c_device_id
*id) {
    struct mcp23016 *mcp;
    [...]
    mcp = devm_kzalloc(&client->dev, sizeof(*mcp), GFP_KERNEL);
    if (!mcp)
        return -ENOMEM;
    [...]
}
```

Listas enlazadas

Listas enlazadas

- Si se tiene un driver que administra más de un dispositivo, es muy útil hacer un seguimiento de estos en el driver.
- Para estos casos se utilizan listas enlazadas en espacio kernel.
- Existen dos tipos de listas enlazadas:
  - De enlaces simples.
  - Doblemente enlazadas.
- Los desarrolladores de kernel solo implementan doblemente enlazadas para dar soporte a FIFO y LIFO a la vez.

Listas enlazadas

- Además, permite que el set de código se mantenga mínimo con la mayor generalidad.
- Para la utilización de listas enlazadas, se debe incluir el archivo **include/linux/list.h**.
- La estructura de datos en el núcleo de la implementación en el kernel es **struct list_head**.

```
struct list_head {
    struct list_head *next, *prev;
};
```

Listas enlazadas

- Aunque su nombre se preste a confusión **list_head** se utiliza en todos los nodos de la lista enlazada, no solo en el primero.
- En el espacio kernel, para que una estructura de datos se re**ead**.

```
struct auto {
    int cant_puertas;
    char *color;
    char *modelo;
    struct list_head lista; /* kernel's list structure
*/
```

Listas enlazadas

- Para iniciar su utilización, se crea una variable struct list_head que siempre apunte al primer elemento de la lista (head).
- Recordar que la estructura **head_list** solo tiene dos punteros, pero ningún contenido.
- La primer instancia de **list_head** es especial y no está asociada a ningún elemento.

```
static LIST_HEAD(lista_autos) ;
```

Listas enlazadas

- EJEMPLO: Uso de lista enlazada.

```
#include <linux/list.h>
struct car *redcar = kmalloc(sizeof(*car), GFP_KERNEL);
struct car *bluecar = kmalloc(sizeof(*car), GFP_KERNEL);
/* Initialize each node's list entry */
INIT_LIST_HEAD(&bluecar->list);
INIT_LIST_HEAD(&redcar->list);
/* allocate memory for color and model field and fill every field */
[...]
list_add(&redcar->list, &carlist) ;
list_add(&bluecar->list, &carlist) ;
```

Listas enlazadas

- La creación e inicialización de una lista se puede hacer de dos maneras distintas:
  - Método dinámico (asignación de memoria en tiempo de ejecución).
  - Método estático (asignación de memoria en tiempo de compilación).
- El uso de un método u otro dependerá del criterio del desarrollador.
- Ambos métodos hacen uso de macros que ayudan a la legibilidad del código

Listas enlazadas

- **Método estático**
  - La asignación estática se hace mediante la macro **LIST_HEAD**

```
LIST_HEAD(mylist);
```

```
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
```

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

Listas enlazadas

- **Método dinámico**
  - Se debe crear una estructura **list_head** y se utiliza **INIT_LIST_HEAD**

```
struct list_head mylist;
INIT_LIST_HEAD(&mylist);
```

```
static inline void INIT_LIST_HEAD(struct list_head
*list) {
    list->next = list;
    list->prev = list;
}
```

Listas enlazadas

- La creación de nodos es sencilla. Solo se crea la estructura de datos y luego se inicializa el campo **list_head** contenido.

```
struct auto *auto_negro = kzalloc(sizeof(struct auto),
GFP_KERNEL);
/* inicializacion dinamica, porque es el campo lista en la
estructura*/
INIT_LIST_HEAD(&auto_negro->list);
```

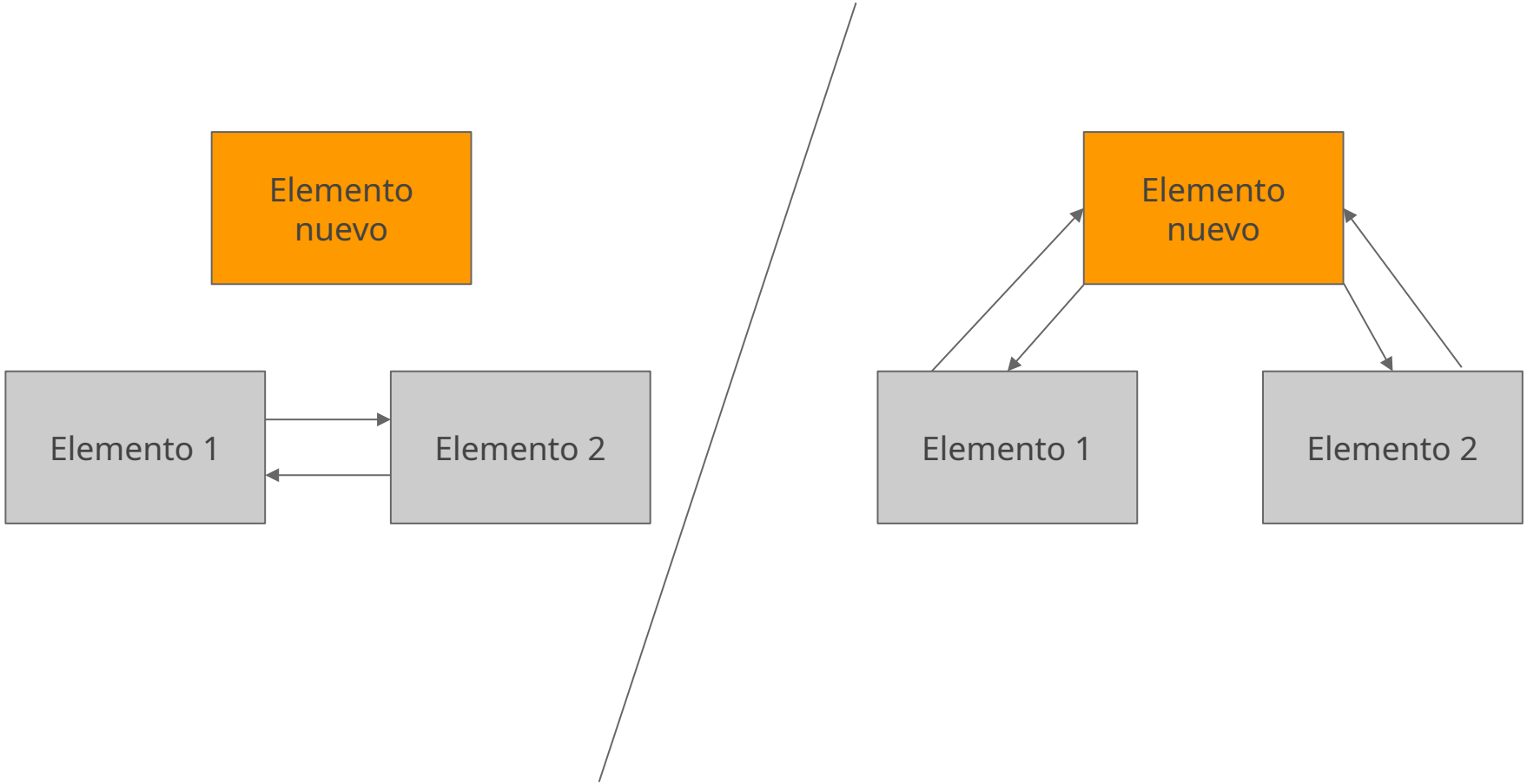
- Se usa **INIT_LIST_HEAD** porque es una lista asignada dinámicamente, y comúnmente parte de otra estructura.

Listas enlazadas

- Para agregar un nodo, el kernel provee el método **list_add**.
- Este método es un wrapper de la función **__list_add()**.
- En resumidas cuentas, agrega un nuevo nodo en la posición siguiente al nodo indicado.
- En el caso de haber elementos anteriores y posteriores existentes, se abre la lista e inserta el nuevo en esa posición.

```
list_add(&auto_rojo->list, &lista_autos);
list_add(&auto_azul->list, &lista_autos);
```

Listas enlazadas



Listas enlazadas

- **NOTA:** al pasar como argumento nodo actual la estructura especial inicio de lista, los nodos se agregan al inicio.
- Se puede asociar el funcionamiento a un stack (LIFO).
- Existe también el método **list_add_tail()** que tendrá el comportamiento contrario, agregando los nodos al final.
- Para agregar un nodo en una posición arbitraria, se debe hacer uso de la función **__list_add()** directamente.

Listas enlazadas

- Para eliminar nodos, se utiliza el método **list_del()**.
- El método **list_del()** desconecta el nodo indicado de la lista enlazada, pero no libera la memoria asociada al mismo.
- Para liberar la memoria asignada dinámicamente debe hacerse manualmente llamando a **kfree()**.

```
list_del(&auto_rojo->list);
```

Listas enlazadas

- Para recorrer la lista el kernel provee la macro **list_for_each_entry(pos, head, member)**.
  - **head** es el nodo list_head.
  - **member** es el nombre de la struct list_head dentro de la estructura de datos.
  - **pos** se utiliza para la iteración. Es un índice de loop que en cada iteración apunta al elemento actual.
  - pos es un puntero a la estructura que contiene el tipo member. Se utiliza la macro container_of para determinar esto.

Mecanismo Sleep

Mecanismo Sleep

- Sleep es el mecanismo por el cual un proceso libera al procesador para dar posibilidad de atender otro proceso.
- Distintas causas pueden requerir entrar en sleep, como ser disponibilidad de datos o espera de algún recurso ocupado.
- El kernel administra una lista de tareas para correr, conocida como run queue.
- Todo proceso que pase a estado sleep no se incluye en el scheduling, dado que se quita de la run queue.

Mecanismo Sleep

- A menos que el proceso cambie de estado, no se ejecutará nunca.
- Por este motivo, un agente externo debe despertar el proceso (usualmente un evento).
- Para hacer uso del mecanismo sleep, el kernel provee una estructura de datos llamada **wait queue**.
- Está definida en el archivo **include/linux/wait.h** bajo el nombre de **__wait_queue**

Mecanismo Sleep

- Las wait queues son utilizadas para procesos bloqueados que esperan que alguna condición particular se torne *true*.
- Dentro de **__wait_queue** existe un miembro **task_list**, que no es otra cosa que una lista enlazada.
- Todo proceso en sleep será encolado en esa lista hasta tanto una condición indicada se torne *true*.

Mecanismo Sleep

- Declaración estática de wait queue:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

- Declaración dinámica de wait queue:

```
wait_queue_head_t my_wait_queue;
init_waitqueue_head(&my_wait_queue);
```

Mecanismo Sleep

- Bloqueo (solo si <CONDICION> es *false*):

```
int wait_event_interruptible(wait_queue_head_t q,
<CONDICION>);
```

- Desbloqueo (solo si <CONDICION> se tornó *true*):

```
void wake_up_interruptible(wait_queue_head_t *q);
```

Mecanismo Sleep

- **wait_event_interruptible** no ejecuta un poll continuo, sino que simplemente evalua <CONDICION> cuando es llamada.
- Si <CONDICION> es *false* el proceso pasa a estado **TASK_INTERRUPTIBLE** y se remueve de la *run queue*.
- A partir de ese punto, <CONDICION> se chequea solo cuando se llama a **wait_event_interruptible**.
- Cuando <CONDICION> pasa a ser *true* el proceso se despierta y pasa a estado **TASK_RUNNING**.

Mecanismo Sleep

HANDS ON

1. Compilar el módulo ejemplo *my_sleep*.
2. Observar comportamiento y salidas en kernel log

Administración de delay y timers

Administración de delay y timers

- Como es sabido, el tiempo es el recurso más valioso del sistema, que se traduce en ciclos de CPU.
- El kernel hace uso de distintos timers para el seguimiento del tiempo relativo que es muy utilizado en varias tareas:
  - Scheduling.
  - Work deferring.
  - Timeouts.
  - Sleep.

Administración de delay y timers

- Los timers del kernel se clasifican en dos:
  - Timers estándar, llamados también system timers
  - Timers de alta resolución (high-resolution).
- Los timers estándar basan su operación en una granularidad medida en **jiffies**.
- Un **jiffy** es una unidad de tiempo que utiliza el kernel, declarada en **linux/jiffies.h**

Administración de delay y timers

- En la definición, la constante **HZ** representa la cantidad de veces que se incrementa *jiffies* en un segundo.
- Cada uno de estos incrementos es llamado **tick**.
- Puede decirse entonces que **HZ** es el tamaño de un jiffy.
- Claramente, HZ depende del HW y la versión del kernel.
- Determina la frecuencia de las interrupciones de clock, en algunas arquitecturas es configurable, en otras es fijo.

Administración de delay y timers

- El incremento de jiffies se da mediante la definición de la constante **HZ** y la programación de la **PIT**.
- **PIT**: **programmable interrupt timer**. Es un componente de HW que se da una vez por tick y por lo tanto incrementa *jiffies*.
- Como podrán pensar, esto puede dar lugar a overflow según como esté definida jiffies en sistemas de 32 bits.
- Se soluciona con una definición del tipo u64 para estos sistemas, siendo por defecto para los de 64 bits.

Administración de delay y timers

- Un timer estándar está representado en el kernel como una instancia del **timer_list** definida en **linux/timer.h**.
- Dentro de sus campos:
  - **expires** es un valor absoluto en jiffies.
  - **entry** es una lista enlazada.
  - **data** es opcional, y se pasa a la función callback.

Administración de delay y timers

- Set up del timer, donde se indica función callback y datos.

```
void setup_timer(struct timer_list *timer, \
                 void (*function)(unsigned long), \
                 unsigned long data);
```

- Tiempo de expiración para llamar al callback.

```
int mod_timer( struct timer_list *timer, unsigned long
expires);
```

Administración de delay y timers

- Liberación del timer, una vez que se deja de utilizar:

```
void del_timer(struct timer_list *timer);
int del_timer_sync(struct timer_list *timer);
```

- **del_timer()** siempre retorna, haya desactivado un timer pendiente o no.
- **del_timer_sync()** espera hasta que termine la ejecución del handler, aun si esta corriendo en otra CPU.

Administración de delay y timers

- **NOTA:** no se debería mantener un lock que prevenga el handler (callback) que complete su ejecución.
- Hacerlo implicaría un deadlock.
- Es buena práctica liberar los timers en la función clean-up (exit) del módulo.
- Se puede utilizar la función **timer_pending()** para saber si hay algún callback pendiente de ejecución.

Administración de delay y timers

HANDS ON

1. Compilar el módulo ejemplo *my_timer*.
2. Observar comportamiento y salidas en kernel log

Administración de delay y timers

- Los timers estándar no son del todo precisos.
- No aplicables a aplicaciones con *time constraints* más exigentes.
- Para esto los **HRT** se introdujeron a partir del kernel 2.6.16.
- Su utilización depende de la habilitación **CONFIG_HIGH_RES_TIMERS** en la configuración del kernel.
- Los *High Resolution Timers* tienen una granularidad de μs (algunas plataformas llegan a ns).

Administración de delay y timers

- Los HRT no dependen de **HZ**, sino que la implementación está basada en **ktime**.
- **NOTA:** No todas las plataformas pueden hacer uso de HRT dado el soporte de HW necesario.
- Un HRT se representa en el kernel como una estructura del tipo **hrtimer** definida en el archivo **linux/hrtimer.h**.
- **NOTA:** teniendo HRT habilitados en el sistema, sleep y timer ya no dependen de jiffies.

Administración de delay y timers

- Con respecto a los delays en el kernel, existen dos tipos dependiendo el contexto en el que corre el código:
  - Atómicos.
  - No atómicos.
- El uso de delays requiere del header **linux/delay.h**
- Delays en contexto atómico:
  - ndelay(unsigned long nsecs)
  - udelay(unsigned long usecs)
  - mdelay(unsigned long msecs)

Administración de delay y timers

- Los delays en contextos atómicos (como ser ISRs) no pueden hacer uso de sleep.
- Es por esto que se implementan como busy-wait loops, basados en jiffies.
- Dada la implementación de los mismos, se recomienda utilizar siempre **udelay()**.
- **ndelay()** tendrá tanta precisión como tenga el timer de HW.
- Dado que es un busy wait, también el uso de **mdelay()** se desalienta.

Administración de delay y timers

- Para contextos no atómicos, el kernel provee la familia de funciones **sleep**.
- La más utilizada es **msleep(***unsigned long msecs***)** que se basa en jiffies. Recomendada para delays de 10+ ms.
- **usleep_range**(*unsigned long min, unsigned long max*) se utiliza para rangos de 10 us a 20 ms.
- **NOTA: usleep_range()** se basa en *hrtimers*, puede no estar disponible en el sistema.

Kernel Locking

Kernel Locking

- Como ya es sabido, el mecanismo de locking se utiliza para compartir recursos a través de diferentes procesos o hilos.
- Estos mecanismos previenen el uso abusivo del recurso o la concurrencia por parte de distintos procesos.
- El kernel provee varios mecanismos de locking, de los cuales solo se abordará el **Mutex**.

Kernel Locking

- **Mutual exclusion** es el mecanismo de facto utilizado para locking.
- Definido en el archivo **include/linux/mutex.h**

```
struct mutex {
    /* 1: unlocked, 0: locked, negative: locked, possible waiters
*/
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
    [...]
```

Kernel Locking

- Al igual que en las *wait queues*, tenemos una lista enlazada llamada **wait_list**. El principio del mecanismo sleep es igual.
- Los “contendientes” que quieren hacer uso del recurso son removidos del run queue y agregados al **wait_list**.
- Cuando se libera el lock, uno de los elementos que están esperando se despierta y se quita del **wait_list**.
- Todos los elementos (procesos) del **wait_list** están en estado sleep (sin excepción).

Kernel Locking

- Declaración estática del mutex.

```
DEFINE_MUTEX(my_mutex);
```

- Declaración dinámica del mutex.

```
struct mutex my_mutex;
mutex_init(&my_mutex);
```

Kernel Locking

- Locking.

```
void mutex_lock(struct mutex *lock);
int mutex_lock_interruptible(struct mutex
*lock);
int mutex_lock_killable(struct mutex *lock);
```

- Unlocking.

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

Kernel Locking

- Si se requiere solo chequear que el mutex está bloqueado o no, se utiliza la función **mutex_is_locked()**.
- Esta función verifica si el dueño del mutex es NULL o si es un puntero válido.
- Otra funcion util es **mutex_trylock()**.
- Como su nombre lo indica, trata de tomar un mutex.
- Si lo logra retorna 1, de lo contrario retorna 0.

Kernel Locking

- Se recomienda el uso de **mutex_lock_interruptible()**, resultando en un driver que puede ser interrumpido por cualquier señal.
- **mutex_lock_killable()** sólo permite interrupción del driver por señales que matan el proceso.
- El uso de **mutex_lock()** debe ser solamente en casos que se sepa que el mutex se libera en cualquier situación.
- Ninguna señal interrumpe a **mutex_lock()**, esto incluye CTRL+C

Kernel Locking

- NOTA: Existen reglas muy estrictas para el uso de mutex, listadas en el header correspondiente. Algunos ejemplos:
  - Solo una tarea puede mantener el mutex por vez.
  - Efectuar unlock más de una vez no está permitido.
  - Deben ser inicializados desde la API.
  - Una tarea con mutex tomado debe primero liberarlo y luego retornar, caso contrario si existen contendientes dormirán por siempre.
  - No pueden ser utilizados en contextos atómicos (ej: Timers) porque pueden requerir rescheduling.

Work Queue

Work Queue

- A partir del kernel 2.6 el mecanismo más simple y utilizado para diferir trabajo es work queue.
- Existen dos maneras de utilizar work queues en el kernel:
  - work queue compartida (por defecto).
  - work queue en un kernel thread dedicado.
- Para el alcance de esta materia, solo veremos la *work queue* compartida.
- Al ser una cola global, el trabajo es ejecutado en el momento propicio, dependiendo lo que haya en la cola.

Work Queue

- A menos que se necesite una performance nivel crítico, o no se tenga otra opción, se utiliza la global work queue.
- Al ser una cola global, se debe tener cuidado de no monopolizar su uso por periodos prolongados.
- Al ser una queue serializada, no se debe utilizar sleep por mucho tiempo, porque afecta a las demás tareas.
- No se sabe con quién se comparte el queue (mayor prioridad), a veces toma tiempo que se asigne CPU.

Work Queue

- El trabajo a diferir se debe inicializar con la macro INIT_WORK.
- Al utilizar una cola global, no es necesario crear una estructura work queue.
- Existen pocas funciones para agendar trabajo en la work queue:
  - **schedule_work()**: Asigna al CPU actual.
  - **schedule_delayed_work()**: idem anterior, con delay.
  - **schedule_work_on()**: Asigna al CPU indicado.
  - **scheduled_delayed_work_on()**: Idem anterior, con delay.

Work Queue

- Todas las funciones anteriores agendan el trabajo dado como argumento en la work queue compartida del sistema.
- Esta work queue se llama **system_wq** y está definida en **kernel/workqueue.c**
- Trabajo ya pasado a la *work queue* se puede cancelar con la funcion **cancel_delayed_work()**.
- Se puede hacer un flush de la *work queue* mediante **flush_scheduled_work()**.

Work Queue

HANDS ON

1. Compilar el módulo ejemplo *my_wq*.
2. Observar comportamiento y salidas en kernel log

Gracias.