TEMA 6. Gestión dispositivos

(…)

Dispositivos de bloque

Definición: Componente de la computadora en el que los datos se transmiten en conjuntos indivisibles (en bloques de un cierto número de bytes) en la comunicación con la CPU para acceder a la información. Estos dispositivos tienen capacidad de albergar una cache o buffer para los accesos, a diferencia de los de carácter que son 'unbuffered'.

Ejemplos de disp. de bloque = discos duros (HDD, SSD, SSHD, …), medios de almacenamiento secundario como los pendrives, cámaras USB, y la ¡¡RAM!!

Recordar que **en Unix/Linux todo es un fichero**, por lo tanto, estos dispositivos también se representan como ficheros que se pueden leer o escribir en ellos. (b/c)

Importantes:

- **Ram disks = ram0, ram1, (…), ram15.** (No confundir con la RAM, sino que es para la RAM disk, es decir, pedazos de la memoria principal usados para cargar de forma más rápida que si se usase el medio de almacenamiento secundario. Por defecto, 16 RAM disks) → Particiones; ram2p1, ram2p2, …
- **Loop0, loop1.** (Dispositivo de lazo o loop device. Muy práctico, entre otras cosas, para montar una imagen ISO en él sin necesidad de quemarla en un disco óptico real)
- **Sda, sda1, …**

+ Ram disk = es utilizar memoria RAM para almacenar datos como si de un disco duro se tratase, así el acceso es mucho más veloz.

Drivers de tipo bloque:

- **Inicialización:** res = register_blkdev (unsigned int major, const char * name);
  The name of the new block device and it must be unique within the system.
- Lo anterior, solo se usa para obtener el major number, a continuación, hay que usar la **struct block_device_operations** (ver operaciones que permite), que usa la **struct gendisk.**

```
struct gendisk * disk = alloc_disk(max_partitions – 1);
disk->major           = DISK_MAJOR;
disk->fops            = disk_fops;
…
blk_register_region(MKDEV(DISK_MAJOR, 0), 1 << MINORBITS,
                    THIS_MODULE, brd_probe, NULL, NULL);
```

- **blk_register_region =** Reserva el rang de números minor [0:range-1] pel dispositiu.
- **Cancelación:** int unregister_blkdev(unsigned int major, const char *name);

Soporte para la gestión de memoria:

Usando **Simple List Of Blocks (SLOB)** que es uno de los tres asignadores de memoria disponibles en el kernel de Linux. (por tanto, interno al kernel, <linux/slob_def.h>)

- static inline void *kmalloc(size_t size, gfp_t flags);
- void kfree(const void *block);

Otra, interna al kernel <linux/slab.h>:

- static inline void *kzalloc(size_t size, gfp_t flags)
- static inline void *kcalloc(size_t n, size_t size, gfp_t flags)

+ Slab = se usa para denominar la memoria disponible para incrementar una caché manteniendo su rendimiento. Es la cantidad por la que una región de memoria puede crecer o disminuir.

Dispositivos USB (en realidad, son de bloque)

Están conectados en un bus, cada uno atiende a las peticiones que se le dirigen a él. Se identifican con (vendor:product).

Se puede usar la **librería LIBUSB →** Es una biblioteca que proporciona aplicaciones con acceso para controlar transferencia de datos hacia y desde dispositivos USB en sistemas Unix y no-Unix, sin la necesidad de controladores en modo kernel. Permite:

- Listar
- Acceder a la información
- Ver las características del dispositivo (identificador, MaxPacketSize, Endpoints)

USB endpoints:

Los usbs utilizan **endpoints** para la comunicación USB. Un endpoint USB puede transportar datos en una sola dirección, ya sea desde el ordenador host hacia el dispositivo (llamado OUT endpoint) o desde el dispositivo al ordenador (llamado IN endpoint).

Hay diferentes endpoints, según el tipo de transferencia, **puede haber hasta 30**.

El **endpoint 0** siempre existe y es el que se encarga del control. Se utiliza para configurar el dispositivo, obtener información sobre el dispositivo, el envío de comandos al dispositivo, o recuperar los informes sobre la situación sobre el dispositivo.

Soporte para USB:

Estan registrados con una tabla de operaciones particular.

Se registra un nuevo USB driver usando el USB core de Linux con:

int usb_register (struct usb_driver * new_driver);

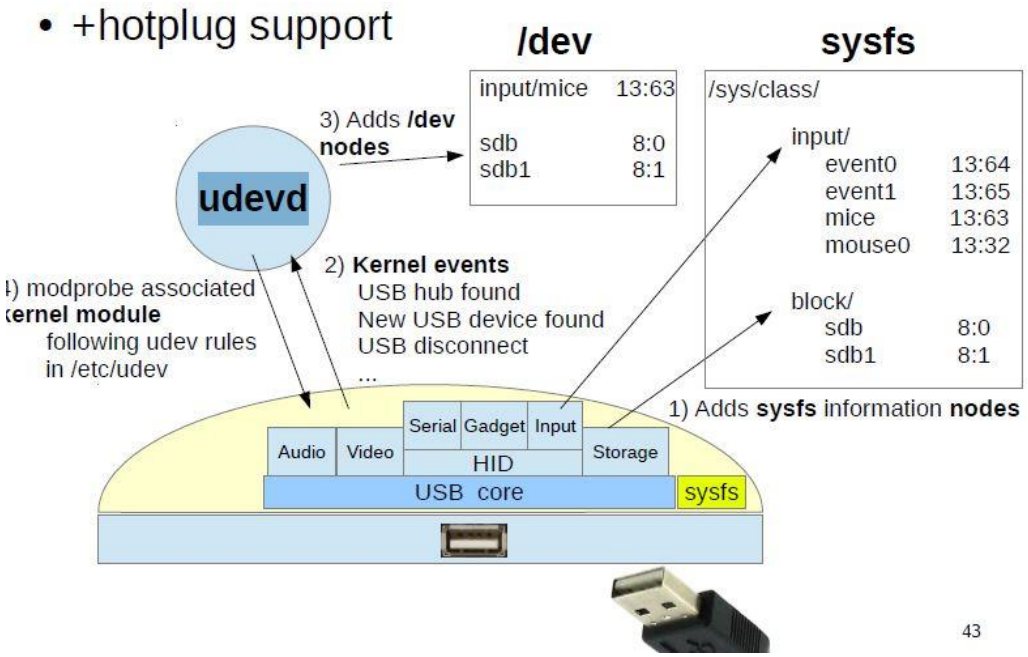
Algunas de las operaciones USB que permite, son las siguientes: (definidas en la struct usb_driver)

```
struct usb_driver {
        const char *name;

        int (*probe) (struct usb_interface *intf,
                      const struct usb_device_id *id);

        void (*disconnect) (struct usb_interface *intf);

        int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
                        void *buf);

        int (*suspend) (struct usb_interface *intf, pm_message_t message);
        int (*resume) (struct usb_interface *intf);
        int (*reset_resume)(struct usb_interface *intf);

        int (*pre_reset)(struct usb_interface *intf);
        int (*post_reset)(struct usb_interface *intf);
 ...
};
```

40

+ La función “probe” registra la interfície! Y las operaciones del dispositivo son conocidas.

Soportan **hotplug** (Conexión en caliente, que permite no apagar la máquina al conectar el USB). Y esto es posible, debido a la siguiente imagen:



43

+ **udev** es el gestor de dispositivos que usa el kernel Linux en su versión 2.6. Su función es controlar los ficheros de dispositivo en /dev. Es el sucesor de devfs y de hotplug, lo que significa que maneja el directorio /dev y todas las acciones del espacio de usuario al agregar o quitar dispositivos en caliente.

+ **Sysfs** exporta información sobre los dispositivos y sus controladores desde el modelo de dispositivos del núcleo hacia el espacio del usuario, también permite configurar parámetros.

Estos dos, permiten un espacio de nombres consistente → **Linux Standard Base (LSB)**. La lista de los diferentes buses utilizados en Linux, se encuentra en /sys/bus.

Dispositivos de red

Drivers Ethernet:

Inicialización: ret = pci_register_driver(&e1000_driver);

Tienen una serie de operaciones y gestión a las interrupciones…

TEMA 7. Soporte por tiempo real

Definiciones y conceptos

Definición: Sistema en tiempo real son aquellos que deben producir respuestas correctas dentro de un intervalo de tiempo definido. Si el tiempo de respuesta excede ese límite, se produce una degradación del funcionamiento y/o un funcionamiento erróneo (aunque el resultado sea correcto).

Se utiliza un scheduler/planificador en la ejecución de los diferentes procesos para cumplir a tiempo (deadline) y no ser interrumpidos por otros. (a no ser que haya otros más prioritarios)

+ Deadline = Asignada a una tarea, es el tiempo máximo en el que la tarea debe haberse ejecutado, para que el sistema pueda continuar funcionando.

Clasificación (según requisitos temporales)

– Tiempo real estricto (**hard real time**): Cuando es absolutamente necesario que la respuesta se produzca dentro del límite especificado. Fallada total del sistema. Ej.: control de vuelo.

– Tiempo real no estricto (**soft real time**): Cuando se permite la pérdida ocasional de especificaciones temporales, aunque debe cumplirse normalmente. La utilidad de los resultados decae cuando más tarde llegan. Ej.: sistema de adquisición de datos, Linux.

– Tiempo real firme (**firm real time**): Cuando se permite la pérdida ocasional de especificaciones temporales, pero dicha perdida no implica beneficios ya que la respuesta retrasada es descartada. Ej.: sistema multimedia, video rendering.

Tipos de tareas:

- **Periódicas** → Son aquellas que se repiten indefinidamente. Habitualmente responden a un evento externo.
- **Aperiódicas** → Las tradicionales, que empiezan y terminan, sin repetirse necesariamente.

OS vs RTOS:

- Multitasking. → OS: Mismo tratamiento a todos los usuarios y procesos. RTOS: Usan las prioridades de los procesos/flujos de forma estricta.
- Sobrecarga sistema. → OS: Se permite. RTOS: NO se permite. Puede ir acompañada de pérdidas de deadlines.

Políticas de planificación

El RTOS Scheduler tiene en cuenta la prioridad de los procesos y sus diferentes estados:

Running, **Ready**, **Suspended**, **Blocked** (esperando evento), **Timed** (esperando evento, pero con un timeout), **Delaying** (esperando que pase un tiempo, timer).

Estos cambios de estado son producidos por eventos externos o por otra tarea.

Reglas que implementa el scheduler:

- Los N procesos (ready) con más prioridad corren.
- Si se debe elegir entre varios procesos que tienen la misma prioridad, se escoge aquel que hace más tiempo que no se ha activado.
- Si varios procesos están esperando (blocked) un evento, activan cuando pasa el evento respectivo, en el orden por su prioridad.
- En el momento en que la primera regla no es cierta, se hace un cambio de contexto a otro proceso (mes prioritario).

Algunas políticas de planificación usadas en RT:

Rate-monotonic scheduling (RMS) → Se usa para tareas periódicas. Si el proceso tiene una duración de trabajo pequeña, entonces tiene la máxima prioridad. Otra forma de decirlo, la tarea con mayor frecuencia será más prioritaria. Prioridad estática.

Earliest Deadline First → Se usa para tareas periódicas. Se busca el proceso más cercano a su deadline para ponerlo como prioritario. Prioridad dinámica, puede ir cambiando con el tiempo.

Soporte en Linux (de RT)

Linux con soporte por tiempo real, sería Tiempo real no estricto (soft). Algunas de las distribuciones Linux con RT son; **Arch Linux**, **Debian, Suse**, **Gentoo**, **RedHat** y **Ubuntu**.

PAM – Pluggable Authentication Modules → Proporcionan soporte de autenticación dinámica para aplicaciones y servicios en un sistema Linux. En RT, se puede utilizar para configurar los límites de los procesos (MEMLOCK, NICE, RTPRIO) y definir clases de prioridades para la I/O.

Inversión de prioridad

¿Qué es? Es un suceso que aparece cuando un flujo más prioritario espera para poder acceder a una región crítica o a un recurso/dispositivo que está ocupado por un flujo menos prioritario.

Puede ser una **inversión ilimitada de prioridades** (tareas intermedias que retrasa aún más la ejecución de una tarea prioritaria)

Soluciones:

- **Herencia de prioridad** → Consiste en transferir la prioridad del flujo más prioritario, al flujo menos prioritario de manera que salga de la región critica lo antes posible y solucionar el problema.
- **Priority ceiling** → Cada recurso tiene una prioridad por default, pero las tareas prioritarias se ejecutan en un nivel superior (ceiling) para evitar la inversión de prioridad.

RT-Preempt

Proporciona características de tiempo real estricto (**hard real time**) en Linux. Permite:

- Las regiones críticas puedan sufrir **preempciones**. (Apropiación previa de algo)
- Implementa herencia de prioridades para el kernel (spinlocks y semáforos)
- Convierte los gestores de interrupciones en flujos, los cuales se les puede cambiar la prioridad.
- Soporta herencia de prioridades en el pthread_mutex con un nuevo atributo PTHEREAD_PRIO (none, inherit, protect).
- Disponible para Debin, SUSE y Ubuntu.

+ Spinlocks VS. Semáforos: https://stackoverflow.com/questions/195853/spinlock-versus-semaphore

Contiene: (según el OSADL)

- Deterministic scheduler
- Preemption support
- PI mutexes
- High-Resolution timer
- Preemptive Read-Copy update
- IRQ threads
- Raw Spinlock annotation
- Forced IRQ threads
- R/W semaphore cleanup
- Full Realtime preemption support

+ Derministic scheduler → Tiene varias alternativas:

- Earliest Deadline First (visto antes)
- **Least Laxity First** (Le da prioridad a la tarea que tiene menos espacio para que alcance el deadline a tiempo, corre lo más tarde posible)

Otros sistemas de tiempo real

VxWorks: Sistema operativo de tiempo real desarrollado como software propietario por Wind River Systems. Es utilizado para sistemas embebidos que habitualmente necesitan una respuesta rápida del orden de ms o microsegundos ante interrupciones en su funcionamiento, una reconocida estabilidad y una seguridad certificada. Se usa como ejemplo en cajeros automáticos, impresoras, cámaras fotográficas y en sistemas críticos de complejos productos aeroespaciales.

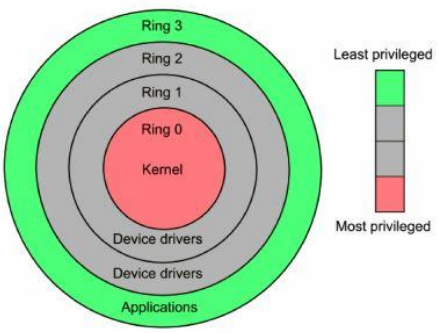
Usa primitivas de sincronización de bajo nivel, CPU affinity (visto en tema 1) y minimiza el desbalanceo en el trabajo de los flujos, evitando así, la sobrecarga de recursos.

Xenomai: Es un marco de desarrollo en tiempo real que coopera con el kernel de Linux, para proporcionar un soporte generalizado, agnóstico a la interfaz y hard RT para aplicaciones de espacio de usuario, perfectamente integrado en el entorno Linux. Permite ejecutar servicios en tiempo real al lado de aplicaciones que no requieren de RT.

Implementa un orden en la distribución de eventos llamado **Adeos**. Permite que varias entidades llamadas "dominios" existan simultáneamente en la misma máquina, estos no se ven necesariamente entre ellos, pero todos ven Adeos. Estos dominios son eficientes y con ráfagas cortas de ejecución.

Todos estos dominios compiten por procesar eventos externos (por ejemplo, interrupciones) o internos (por ejemplo, trampas, excepciones), de acuerdo con la prioridad que se les ha otorgado en todo el sistema. Los eventos se distribuyen primero al dominio más prioritario.

- aconseguir que Linux cedeixi el control a Adeos
- per això es fa que el mòdul d'Adeos mogui Linux al ring 1
  - Les instruccions privilegiades causen una excepció



Hay una serie de instrucciones…Pag 54 power. Las más usadas por los drivers son; in, out, ins, outs. También la interfície de Xenomai tiene una serie de funciones para las tareas:

Rt_task_create(), rt_task_start(), rt_task_spawn() y rt_task_srt_periodic() son las más importantes. Se parece a Mach que debe primero crear la tarea y luego lanzarla con otra función.

Pthreads RT: Basicamente, pthread tiene implementado algunas características de real time. Como ejemplo:

- Barriers → Uso de funciones como pthread_barrier_init(), pthread_barrier_wait() y de algunos atributos; PTHREAD PROCESS PRIVATE//PTHREAD PROCESS.SHARED
- Pthread_spin
- Pthread_getcpulockid()

(ver implementación barriers con pthreads, pag 65 - 68)