S. empotrados y ubicuos

Programación de dispositivos
(3ª sesión)

Fernando Pérez Costoya
fperez@fi.upm.es

Contenido

- Introducción
- Repaso de aspectos básicos del sistema de E/S
- El hardware de E/S visto desde el software
- Aspectos generales de la programación de dispositivos
- **Programación de manejadores dispositivos**
  - Programación de manejadores dispositivos en sistemas sin SO
  - Programación en SS.OO. monolíticos
  - Programación en SS.OO. basados en micronúcleos
  - Caso práctico: programación de manejadores en Linux

Programación de dispositivos en máquinas sin SO

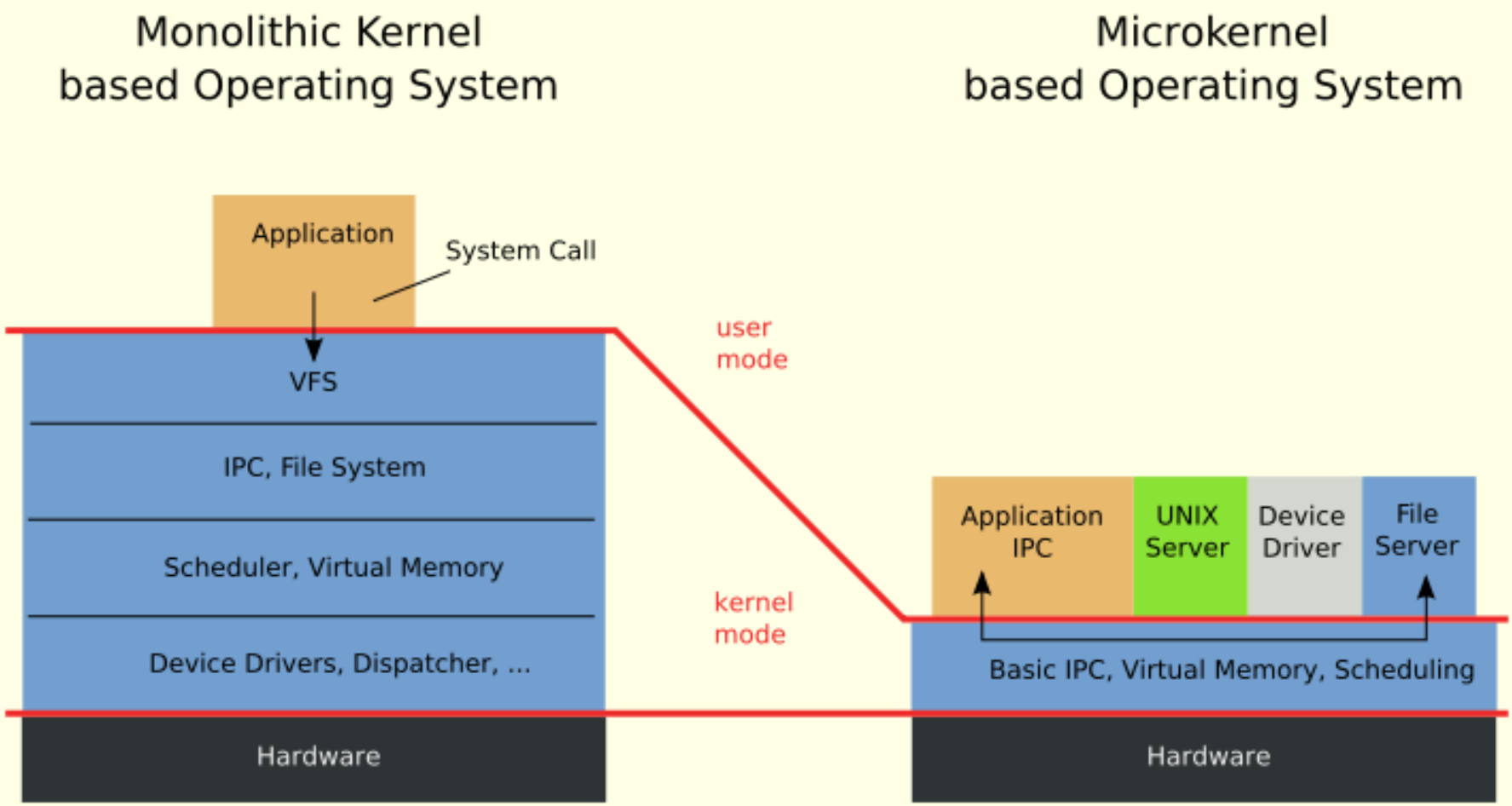
- Programador debe enfrentarse con toda la problemática identificada
  - Sólo con la ayuda del *runtime* del lenguaje

- Alternativas en el diseño de app que gestiona múltiples dispositivos
  - Ejecutivo cíclico con espera activa y *polling* de dispositivos
  - Ejecutivo cíclico con interrupciones
    - Minimizar duración rutinas de interrupción
  - Aplicación concurrente si *runtime* del lenguaje lo permite
    - Ada, Java,…

Programación de dispositivos en sistemas con SO

- SO ofrece enorme soporte para desarrollo de nuevos manejadores
  - API para su desarrollo
  - Abstracciones proceso y *thread* como modelo de concurrencia
- Tipo de arquitectura del SO:
  - Basado en un micronúcleo (p.e. Mach, QNX, Google Fuchsia):
    - Manejadores de E/S fuera del SO (ejecutando modo usuario)
  - Monolítico (p.e. familia UNIX, Linux):
    - Manejadores de E/S dentro del SO (ejecutando modo privilegiado)
    - SO monolíticos actuales con módulos cargables
      - Permite adaptar núcleo a plataforma (p.ej. sistema empotrado)
      - Posibilita técnicas como *hot-plugging*

Monolítico versus micronúcleo (wikipedia)



Manejador de dispositivo (*Device Driver*)

- Módulo que gestiona una clase de dispo. (varias unidades)
  - Esconde particularidades específicas de cada clase
  - Provee interfaz común independiente de disp. a resto de sistema
- Manejador en sistemas monolíticos:
  - Módulo que se enlaza con el resto del SO
    - De forma estática o de forma dinámica (módulo cargable)
  - Proporciona acceso a dispositivos mediante llamadas al sistema
  - Puede hacer uso de todas funciones internas del SO
  - Resto del SO puede usar sus funciones
- Manejadores en sistemas basados en micronúcleos
  - Procesos de usuario que reciben mensajes
- Presentación se centra en sistemas monolíticos (Linux)
  - Aunque al final se recogen aspectos específicos de micronúcleos

Desarrollo manejadores en sistemas monolíticos

- Se trata de una labor de programación pero con peculiaridades...
- Se usa una versión “recortada” de la biblioteca del lenguaje
  - P.e. en caso de C no debe incluir `fread` pero sí `strcmp`
- Pila de tamaño limitado (p.e. 8K) y sin control de desbordamiento
  - Evitar vars. de pila grandes y mucho anidamiento de llamadas
- Economía en el uso de memoria: es memoria no paginable
- Error en código → cuelgue del sistema
  - Opción: desarrollo sobre máquina virtual
- Difícil depuración (errores no reproducibles al tratar con HW):
  - Falta herramientas equivalentes a depuradores de aplicaciones
  - Habitual: Imprimir por consola y análisis de trazas
- Tipo de desarrollo con baja productividad y propenso a errores

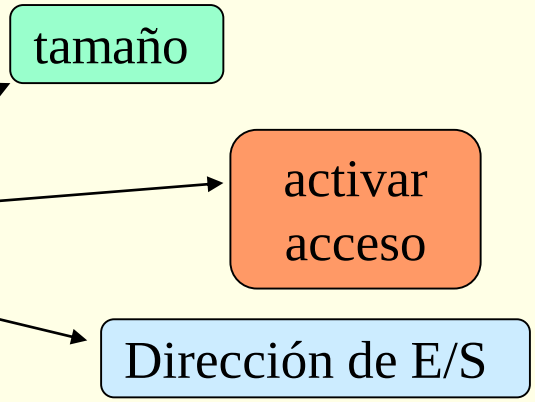
¿Manejadores en modo usuario en monolíticos?

- Mejor si se puede manejar dispo. desde software en modo usuario
  - Todas las ventajas del micronúcleo
- ¿Cómo acceder a E/S desde modo usuario?
- En Linux:
  - PIO: iopl, ioperm, inb, inw, outb, outw,…
  - MMIO: mmap de /dev/mem
- Pero...
  - No es posible manejo de interrupciones
  - Y no suficiente eficiente para dispositivos de altas prestaciones
    - Sobrecarga por cambios de modo y de contexto, código y datos del manejador expulsables de memoria (posible uso de mlock), …

Ejemplo PIO en modo usuario en Linux

Acceso disp. RTC que usa puertos 0x70 y 0x71 de 1 byte

```
#define RTC_REG 0x70
#define RTC_DAT 0x71
int main() {
    uint8_t valor, dato;
    ioperm(RTC_REG, 2, 1);
    ……
    outb(valor, RTC_REG);
    ……
    dato = inb(RTC_DAT));
    ……
```



Ejemplo MMIO en modo usuario en Linux

Local-APIC, por defecto, ofrece sus registros de 4 bytes a partir de d. física 0xfee00000 ocupando hasta 4K. Acceso a un registro con offset *X* con respecto a esa dirección.

```
int main() {
    int fd, tam=4096;
    uint32_t dato, offset = X;
    uint32_t volatile *p;
    fd = open("/dev/mem", O_RDONLY);
    p = mmap(NULL, tam, PROT_READ, MAP_SHARED, fd, 0xfee00000);
    .......
    dato = *(p + offset);
    ..........
```

Debe ser múltiplo del tamaño de la página

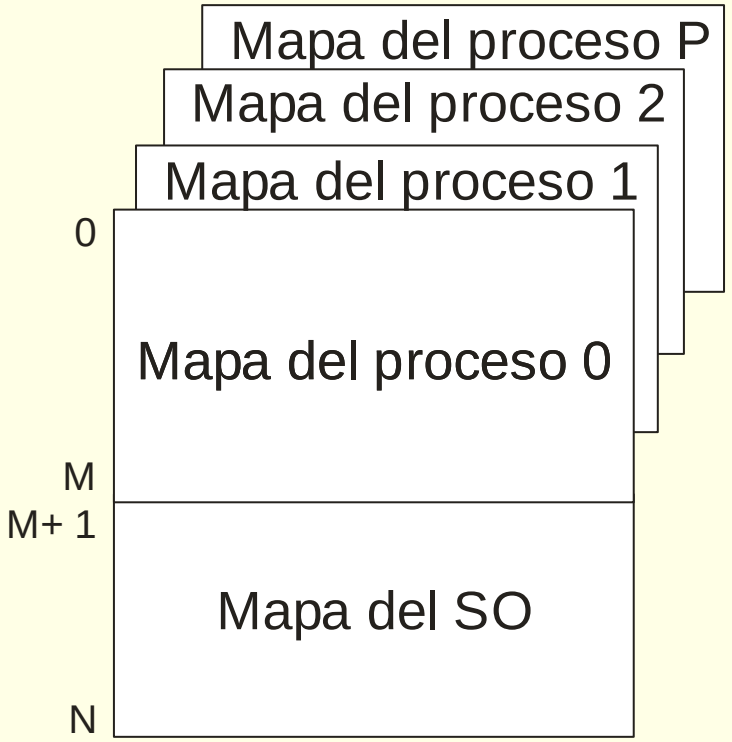
Contexto de ejecución de funciones del manejador

- Importante distinción entre funciones del manejador:
  - Su contexto de ejecución
- Funciones de acceso al dispo. (abrir, leer, escribir, cerrar, ...)
  - Ejecución en el contexto del proceso “solicitante”
  - Se puede acceder a mapa de memoria de proceso actual
  - Se puede realizar una operación de bloqueo
- Funciones de interrupción
  - Ejecución en contexto de proceso no relacionado con la int.
  - No se puede acceder a mapa de memoria de proceso actual
  - No se puede realizar una operación de bloqueo

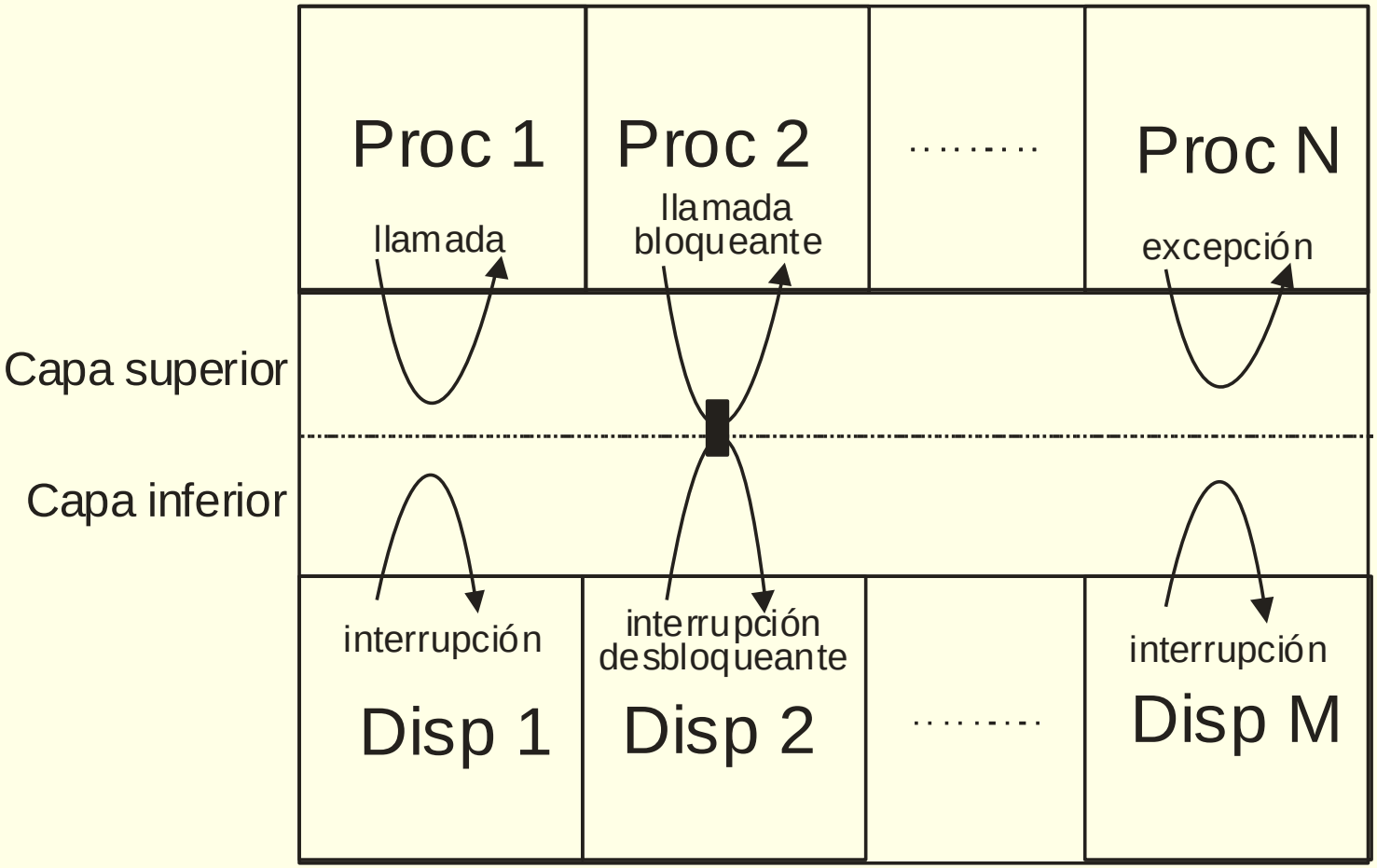
Contextos de ejecución

- SO programa dirigido por eventos:
  - Una vez iniciado el sistema, sólo se ejecuta código del SO si:
    - Interrupción, llamada al sistema, excepción (p.e. fallo de página)
    - Int. es asíncrona, llamada (y excepción) síncrona

- Modelo de memoria del proceso



Organización del SO



Diseño de manejador de dispositivo de caracteres

- Pautas de diseño usando un dispositivo de E/S hipotético X:
  - Opera en modo carácter, dirigido por interrupciones y sin DMA
  - Operación de escritura:
    - dato → r. datos;
    - orden → r.control;
    - Interrupción → fin operación
  - Operación de lectura:
    - orden → r.control;
    - Interrupción → fin operación: dato en r. datos
- **No se tienen en cuenta aspectos de sincronización**
  - **ni entre llamadas concurrentes ni entre llamada e interrup.**

Objetivos del manejador hipotético

- Minimizar cambios de modo (Usuario→Sistema | Sistema→Usuario)
- Minimizar cambios de contexto (cambios de proceso)
- Maximizar paralelismo entre SW y HW:
  - Operación concurrente de aplicación y dispositivo
  - Esquema productor-consumidor concurrente:
    - App. escritora produce datos; dispositivo los consume
    - App. lectora consume datos; dispositivo los produce
    - App. puede acceder a ficheros para generar/procesar datos y
      - puede haber fallos de página al acceder a *buffer* de usuario
      - ► Accesos a disco también concurrentes con operación del dispositivo
- Priorizar uso de dispositivo sobre ejecución de aplicaciones:
  - Dispo. sirve peticiones aunque en ejecución proc. alta prioridad

Versión lectura sin *buffering*: errónea

```
char *dirb;
tipo_cola_procesos cola_espera_entrada;
int lectura(char *dir, int tam) {
    dirb = dir;
    while (tam--) {
        out(R_CONTROL_X, LECTURA);
        Bloquear(cola_espera_entrada_X);
    }
}
void interrupcion_entrada_X() {
    *(dirb++) = in(R_DATOS_LEC_X);      /*  ERROR: acceso a mapa usuario
                                            desde rutina de interrupción */
    Desbloquear(cola_espera_entrada);
}
```

Versión escritura sin *buffering*: ineficiente

```
tipo_cola_procesos cola_espera_salida;
int escritura(char *dir, int tam) {
    while (tam-- > 0) {
        out(R_DATOS, *dir++);  // puede causar un fallo de página
        out(R_CONTROL, ESCR);
        Bloquear(&cola_espera_salida); // demasiados cambios de contexto: 1/byte
    }
}
void interrupcion_salida_X() {
    Desbloquear(&cola_espera_salida);
}

// Déficit: entre bytes, dispo. "parado" hasta que vuelva a ejecutar el proceso
```

Versión lectura con *buffer* de 1 byte: ineficiente

```
char buf_ent;
tipo_cola_procesos cola_espera_entrada;
int lectura_X(char *dir, int tam) {
    while (tam--) {
        out(R_CONTROL_X, LECTURA);
        Bloquear(cola_espera_entrada); // demasiados cambios de contexto: 1/byte
        *(dir++) = buf_ent;   // puede causar un fallo de página
    }
}
void interrupcion_entrada_X() {
    char caracter;
    caracter = in(R_DATOS_LEC_X);
    buf_ent = caracter;
    Desbloquear(cola_espera_entrada);}
// Déficit: entre bytes, dispo. "parado" hasta que vuelva a ejecutar el proceso
```

Versión lectura con *buffer* de N bytes

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_entrada;
int tam_datos
int a_leer;
int lectura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        a_leer = (min(tam_datos, tam(&buf))); // tam(&buf): tamaño del buffer
        programar();
        Bloquear(&cola_espera_entrada); // nº cambios de contexto: tam/buf.tam
        copiar_de_buf_a_usuario(&buf, dir, a_leer); // puede causar un fallo de página
        tam_datos -= a_leer;
        dir+= a_leer;
    }
}
```

Versión lectura con *buffer* de N bytes

```
void interrupcion_entrada_X() {
      char c = in(R_DATOS);
      insertar(&buf,c); // buf.nelem++
      if (nelem(&buf)==a_leer) // nelem(&buf): nº bytes almacenados en buffer
            Desbloquear(&cola_espera_entrada);
      else
            programar();
}
void programar(){
       out(R_CONTROL, LECTURA);
}
```

// Mejora: entre bytes, dispo. sigue operando aunque no ejecute el proceso

// Déficit: entre llamadas, dispositivo "parado"

Versión escritura con *buffer* de N bytes

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_salida;
int tam_datos;
int a_esc;
int escritura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        a_esc = (min(tam_datos, tam(&buf)));
        copiar_de_usuario_a_buf(dir, &buf, a_esc); // puede causar un fallo de página
        tam_datos -= a_esc;
        dir+= a_esc;
        programar();
        Bloquear(&cola_espera_salida); // nº cambios de contexto: tam/buf.tam
    }
}
```

Versión escritura con *buffer* de N bytes

```
void interrupcion_salida_X() {
    if (vacio(&buf))
        Desbloquear(&cola_espera_salida);
    else
        programar();
}

void programar(){
    char c = extraer(&buf); // buf.nelem--
    out(R_DATOS, c);
    out(R_CONTROL, ESCRITURA);
 }
```

// Mejora: entre bytes, dispo. sigue operando aunque no ejecute el proceso

// Déficit: entre llamadas, dispositivo "parado"

Versión lectura anticipada

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_entrada; int tam_datos, a_leer, dispo_activo=0;
int lectura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        if (!vacio(&buf)) {
            a_leer = (min(tam_datos, nelem(&buf)));
            copiar_de_buf_a_usuario(&buf, dir, a_leer); // puede causar fallo de página
            tam_datos -= a_leer; dir+= a_leer;
        }
        if (!dispo_activo) {
            dispo_activo = 1; programar();
        }
        if (tam_datos>0)
            Bloquear(&cola_espera_entrada);
    }
}
```

Versión lectura anticipada

```
void interrupcion_entrada_X() {
    char c = in(R_DATOS);
    insertar(&buf,c); // buf.nelem++
    if (lleno(&buf))
        dispo_activo = 0;
    else
        programar();
    if (!vacia(cola_espera_entrada) && (nelem(&buf)>= min(tam_datos, tam(&buf))))
        Desbloquear(&cola_espera_entrada);
}
void programar(){
    out(R_CONTROL, LECTURA);
}
```

ALTERNATIVA

```
// despertar antes al proceso: en cuanto haya nº elementos superior a un cierto umbral
    if (!vacia(cola_espera_entrada) && (nelem(&buf)>= min(tam_datos, UMBRAL)))
// intenta evitar buffer lleno si hay lector y permite paralelismo entre copia y dispositivo
```

Versión escritura diferida

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_salida; int tam_datos, a_esc, dispo_activo=0;
int escritura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        if (lleno(&buf))
            Bloquear(&cola_espera_salida);
        a_esc = (min(tam_datos, espacio_libre(&buf)));
        copiar_de_usuario_a_buf(dir, &buf, a_esc); // puede causar un fallo de página
        tam_datos -= a_esc;
        dir+= a_esc;
        if (!dispo_activo) {
            dispo_activo = 1;
            programar();
        }
    }
}
```

Versión escritura diferida

```
void interrupcion_salida_X() {
    if (vacio(&buf))
        dispo_activo = 0;
    else
        programar();
    if (!vacia(cola_espera_salida) && (espacio_libre(&buf)>= min(tam_datos, tam(&buf))))
        Desbloquear(&cola_espera_salida);
}
void programar(){
    char c = extraer(&buf); // buf.nelem--
    out(R_DATOS, c);
    out(R_CONTROL, ESCRITURA);
 }
```

ALTERNATIVA

```
// despertar antes al proceso: en cuanto haya hueco de tamaño superior a un cierto umbral
    if (!vacia(cola_espera_salida) && (espacio_libre(&buf)>=min(tam_datos, UMBRAL))
// intenta evitar buffer vacío si hay escritor y permite paralelismo entre copia y dispositivo
```

Estructura interna del manejador

- Manejador exporta al resto del SO funciones para:
  - Iniciarse y terminar (al cargarse y descargarse)
  - Añadir y eliminar los dispositivos a manejar si PnP
  - Ofrecer a las aplicaciones acceso a los dispositivos
  - Tratar interrupciones de dispos. e interrupciones software
  - Cambiar nivel consumo de energía de dispositivo (si procede)
- Manejador usa API interno ofrecido para:
  - gestión de bloqueos, acceso a dispositivos,
  - uso de memoria, uso de DMA, control de tiempo
  - sincronización, concurrencia, ...
- Presentamos primero manejadores no PnP y después PnP
  - Al final, el API interno

Ciclo de vida de manejador dispositivos no PnP

- Enlazado estáticamente con SO o cargable en tiempo de ejecución
  - En Linux configurable al generar la imagen SO
  - Normalmente, recomendable que sea módulo cargable
    - Incluso aunque sea para dispositivos no PnP
  - Ofrece funciones para su inicio (module_init) y fin (module_exit)
- Carga módulo dinámico que maneja dispositivos no PnP:
  - Carga del manejador como parte del arranque del SO
  - Carga a petición del superusuario (insmod)
- Descarga módulo dinámico que maneja dispositivos no PnP:
  - Descarga a petición del superusuario (rmmod)
  - Descarga del manejador como parte de parada del SO

Dispositivos en UNIX

- Distingue entre dispositivos de bloques, caracteres y red
  - Dispositivos de bloques y caracteres mismo esquema de acceso
    - Nos centramos en dispositivos de caracteres
- Cada manejador tiene un ID único interno (*major*) por cada tipo
- Y recibe como argumento de sus ops. un nº unidad (*minor*)
- SO ofrece a aplicaciones fichero especial (por convención en /dev)
    - /dev/nombre → car. o bl. + *major* + *minor* → (manejador + unidad)
  - $ ls -l /dev/sda1 /dev/sda2 /dev/tty0 /dev/tty1
  - brw-r----- 1 root disk 8, 1 nov 23 09:47 /dev/sda1
  - brw-r----- 1 root disk 8, 2 nov 23 09:47 /dev/sda2
  - crw-rw---- 1 root root 4, 0 nov 23 09:47 /dev/tty0
  - crw------- 1 root root 4, 1 nov 23 08:47 /dev/tty1

Func. inicial manejador no PnP: recursos HW

- Manejador conoce a priori qué dispositivos gestiona
  - Y qué recursos hardware requieren
  - Esa información debería recibirla como parámetro
  - No deberían incluirse datos fijos en su código
- PIO: manejador debe indicar rango puertos usados (request_region)
  - Permite que núcleo detecte conflictos entre manejadores
- MMIO:manejador indica rango dir E/S usadas (request_mem_region)
  - Permite que núcleo detecte conflictos entre manejadores
  - Solicita crear rango dir. lógicas asociadas a las físicas (ioremap)
- Instala manejadores de interrupción (request_irq)

Func. inicial manejador no PnP: recursos SW

- Reserva IDs internos para dispositivos (UNIX *major* y *minor*)
  - *Major* seleccionado por manejador (register_chrdev_region)
    - Debería recibirlo como parámetro
  - O por el núcleo (alloc_chrdev_region)
- Registra dispositivo de caracteres (cdev_init y cdev_add)
  - Especificando ops. acceso al dispositivo (struct file_operations)
- Hace visible cada dispositivo a aplicaciones de usuario. Linux:
  - Añadirlo a *sysfs* : class_create y device_create
  - Demonio de sistema (udev) detecta nueva entrada en *sysfs*
  - Crea fichero especial con *major* y *minor* asignados
  - Versiones antiguas: creación “manual” con mknod

Funciones acceso al dispositivo struct file_operations

- Ops. manejador proporciona a aplicaciones para acceso a dispos.
  - Abrir, leer, escribir, operaciones control (p.e. rebobinar cinta)
  - Pueden bloquear al proceso que las invoca si es preciso
  - Suelen ser las mismas para todos los dispos. de un manejador
    - Contraejemplo de Linux: manejador mem (/dev/zero, /dev/null, …)
- Todos los manejadores ofrecen misma API
  - ¿Cómo incluir operaciones de control?
    - Función única "cajón de sastre" (en UNIX ioctl)
- struct file_operations
  - http://lxr.free-electrons.com/source/include/linux/fs.h#L1664
- Funcionalidad principal en las operaciones de lectura y escritura
  - Variedad de tipos de operaciones de lectura/escritura

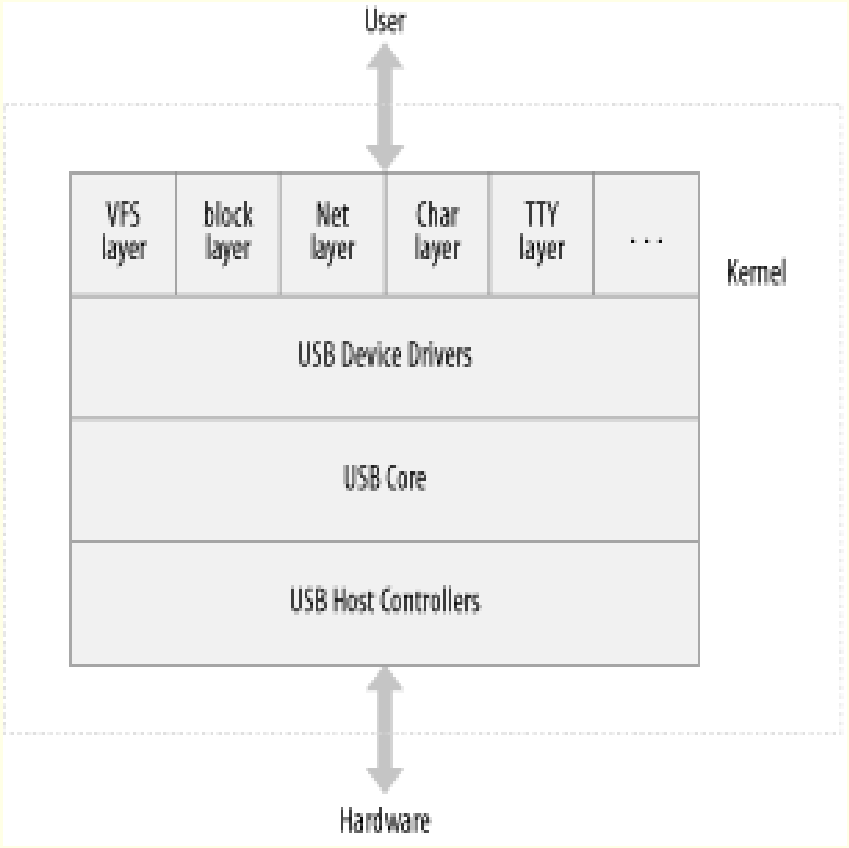
Operaciones de lectura y escritura

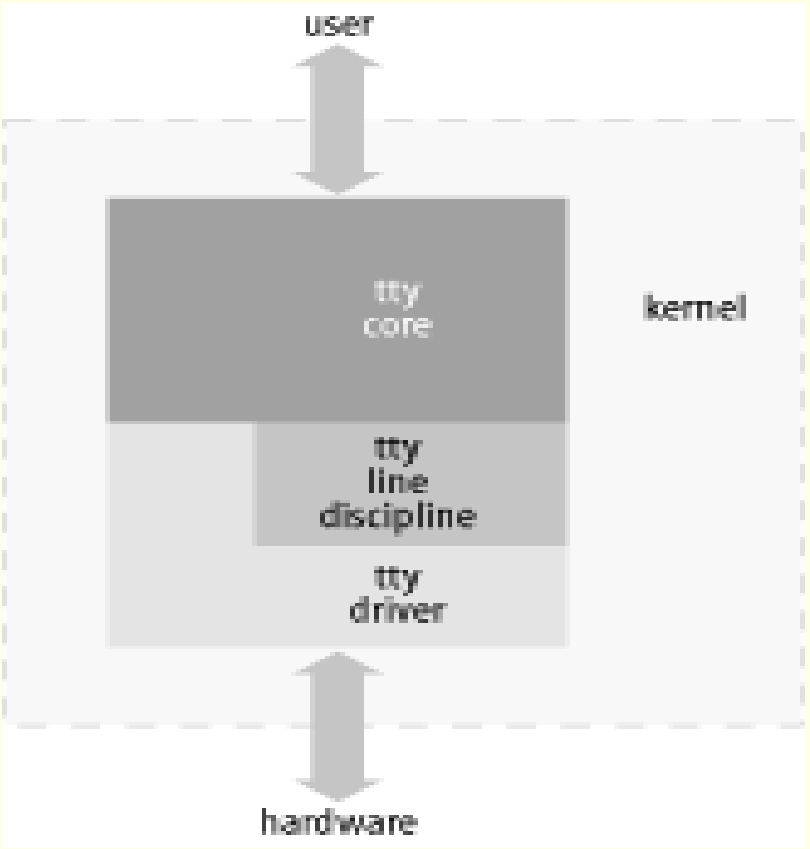
- Convencionales:
  - Programa especifica *buffer* para leer/escribir
  - SO devuelve control cuando datos leídos o escritos
  - Manejador puede usar *buffer* interno como almacén temporal
- No bloqueantes
  - Si operación no puede completarse inmediatamente → error
- Sin *buffers* intermedios
  - disp ↔ espacio de usuario
- Asíncronas
  - Manejador inicia oper. y SO devuelve control inmediatamente
- Con *scatter-gather* a nivel de usuario
  - Programa puede especificar varios *buffers*
- Uso de mmap en vez de read y write

Jerarquía de manejadores de dispositivos

- Conjunto de manejadores no es plano sino jerárquico
  - Código común y necesidad de apilamiento de manejadores
  - Manejadores de buses y de dispositivos
- Ej: webcam USB requiere man. de buses y del propio dispositivo
  - Manejadores de buses:
    - Manejador bus PCI
      - Manejador general de PCI + específico del controlador PCI
    - Manejador del controlador USB conectado a PCI
      - Manejador chip específico + manejador OHCI, EHCI, UHCI
    - Manej. clase HUB de dispo. USB (interacción con *root hub*)
  - Dispo. USB/UVC y cámara: manejador se apoyará en:
    - Manejador de funcionalidad común de todos los disp. USB
    - Manejador específico de clase VIDEO de dispositivo USB
    - Manejador general para todas las cámaras (V4L2)

Ejemplos de jerarquías de manejadores (LDD)





Ciclo de vida de manejador dispositivos PnP

- Especifica qué dispositivos maneja (MODULE_DEVICE_TABLE)
  - Durante compilación módulos recolecta esta info. (depmod)
- Carga del módulo
  - **Manejador de bus** descubre/configura dispositivo
    - En arranque (PnP) o al conectar el dispositivo (*hot-plugging*)
    - Genera evento de descubrimiento hacia modo usuario
  - Proceso de usuario (udevd) lo recibe
    - Consulta información recolectada → módulo manejador
    - Si todavía no cargado (primer dispositivo), lo carga
    - Llama a la función para añadir un dispositivo
- Descarga del módulo (además de al parar el SO)
  - **Manejador de bus** descubre desconexión de dispositivo
    - Genera evento de desconexión hacia modo usuario
  - Proceso de usuario llama a la función para eliminarlo
    - Descarga módulo si último dispositivo

Aspectos específicos manejadores dispo PnP

- Función inicial: no añade dispositivo
  - Registra funciones para añadir/borrar dispositivo
    - pci_register_driver, usb_register, …
  - Serán invocadas cuando se descubran los dispositivos
  - Ejemplo info. de dispositivos PnP en PCI y USB
  - http://lxr.free-electrons.com/source/drivers/usb/host/ehci-pci.c
  - http://lxr.free-electrons.com/source/drivers/media/usb/uvc/uvc_driver.c
  - cat /lib/modules/`uname –r`/modules.pcimap
  - cat /lib/modules/`uname –r`/modules.usbmap
- Función que añade dispositivo:
  - Determina su configuración leyendo mediante dir. geográfico
    - Direcciones MMIO, puertos PIO, líneas de interrupción,…

API del SO usado por los manejadores

- Sincronización
- Acceso a registros de los dispositivos
- Gestión de bloqueos
- Soporte a las necesidades de memoria del manejador
- Soporte de DMA
- Control de tiempo
- Interrupciones software
- Creación de procesos/hilos de núcleo

Sincronización

- Tipos de problemas de sincronización:
  - Producidos por tratamiento interrupciones
  - Debidos a ejecución concurrente de procesos
    - Ejecución entremezclada de procesos en un procesador
    - Ejecución paralela real de procesos en un multiprocesador
- Es necesario crear secciones críticas (SC) dentro del manejador
- SO ofrece cuatro mecanismos para crear SC
  - Inhibir las interrupciones (local_irq_disable)
  - Inhibir la expulsión de procesos (preempt_disable)
  - *Spinlocks*: espera activa basada en operaciones de tipo *test&set*
    - Convencionales (spin_lock_init) o lectores/escritores (rwlock_init)
    - Linux también ofrece *seqlocks* y *RCU-locks*
  - Semáforos/*mutex*: espera bloqueante
    - Convencionales (sema_init) o lectores/escritores (init_rwsem)

Uso de los mecanismos de sincronización

- Sincronización entre llamada/rutina de int. y otra rutina de int.:
  - Ambas usan *spinlock*
  - Rutina interrumpida además inhibe localmente int (spin_lock_irq)
    - No válido semáforos: rutina de interrupción no puede bloquearse
- Sincronización entre llamadas concurrentes
  - Si SC muy corta y sin bloqueos:
    - *spinlock* + expulsión de procesos inhibida
    - En Linux spin_lock incluye preempt_disable
  - En caso contrario: semáforos/mutex proporcionados por SO
    - semáforo internamente usa *spinlock* + expulsión inhibida
- Cuestión de diseño: granularidad de la sincronización
  - Mayor la zona protegida por un elemento de sincronización
    - Menor paralelismo pero también menor sobrecarga

Acceso a registros del dispositivo

- Funciones de acceso al dispositivo del manejador lo requieren
- Acceso PIO
  - SO debe proveer macros portables
  - Evita uso ensamblador en manejador
    - inb, inw, inl, outb, outw, outl, …
- Acceso MMIO: uso de dirección lógica obtenida a partir de ioremap
  - En principio, podría usarse el puntero directamente
  - Pero en Linux desaconsejado: Problemas en algunas UCP
    - Y sobretodo mejor que se identifiquen esos accesos en el código
    - ioread8, ioread16, ioread32, iowrite8, iowrite16, iowrite32…
- SO proporciona barreras de memoria:
  - Para compilador: barrier; Para HW (y compilador): mb

Gestión de bloqueos en Linux

- Funciones de acceso al dispositivo del manejador pueden requerirlo
  - Recordatorio: Interrupciones sólo pueden desbloquear
- Ofrece diversas funciones para bloquear/desbloquear procesos

- void **wait_event**(wait_queue_head_t q, int condition);
- int **wait_event_interruptible**(wait_queue_head_t q, int condition);
- int **wait_event_timeout**(wait_queue_head_t q, int condition, int time);
- int **wait_event_interruptible_timeout**(wait_queue_head_t q, int condition, int time);
- void **wake_up**(struct wait_queue **q);
- void **wake_up_interruptible**(struct wait_queue **q);
- void **wake_up_nr**(struct wait_queue **q, int nr);
- void **wake_up_interruptible_nr**(struct wait_queue **q, int nr);
- void **wake_up_all**(struct wait_queue **q);
- void **wake_up_interruptible_all**(struct wait_queue **q);
- void **wake_up_interruptible_sync**(struct wait_queue **q);

Soporte necesidades de memoria del manejador

- SO debe ofrecer diversas funciones para reservar memoria:
  - Reserva tipo "malloc" (kmalloc)
  - Reserva de páginas contiguas (alloc_pages)
  - Reserva espacio físicamente no contiguo pero sí lógica (vmalloc)
  - Soporte para crear cachés de objetos (kmem_cache_create)
    - Para manejador que reserva y libera mismo tipo de objeto
- Petición de memoria dentro de rutina de interrupción
  - Se deben usar funciones de reserva que no puedan bloquear
    - kmalloc con *flag* GFP_ATOMIC
- Acceso mapa de usuario (copy_from_user | copy_to_user)
  - No se pueden usar desde contexto asíncrono
  - Puede causar fallo de página pero eso es transparente a manej.

Soporte de DMA

- Mantenimiento de la coherencia (dma_alloc_coherent)
- Manejo de direcciones de bus requeridas por IOMMU
  - virt_to_bus, bus_to_virt
- Gestión de *scatter-gather* (struct scatterlist)
- Uso transparente de *bounce buffers*

Control del tiempo

- SO ofrece diversas funciones relacionadas con el tiempo:
  - Temporizadores basados en int. de reloj
    - Manejador requiere realizar una actividad periódica
    - Asocia una función suya con temporizador (add_timer)
      - Función ejecutará en contexto asíncrono (en una interrupción SW)
  - Funciones de espera por un plazo de tiempo
    - Espera bloqueante:
      - Sólo por tiempo (schedule_timeout)
      - Por un evento y por tiempo (wait_event_timeout)
    - Espera activa:
      - Sólo para esperas brevísimas (nanosegundos) (ndelay)
      - SO usa un bucle precalculado o basado en TSC

Interrupciones de dispositivo e int. software

- No todas las operaciones asociadas a interrupción son urgentes
- Importante minimizar duración de rutinas de interrupción
  - Mientras algunas interrupciones están inhibidas
- Ej. interrupción teclado:
  - urgente leer código de tecla; no urgente averiguar car. pulsado
- Rutina interrupción realiza operaciones urgentes
  - No urgentes ejecutan en contexto con interrupciones habilitadas
- Mecanismo de int. software: int. mínima prioridad pedida por SW
  - Rutina int. realiza operaciones urgentes y activa int. software
  - Tratamiento de interrupción SW $\rightarrow$ ops. no urgentes
    - En Linux *softirq (tasklet)* ; En Windows DPC

Uso de procesos/hilos de núcleo

- Manejador sólo se activa cuando se invocan sus funciones
- En ocasiones puede requerir estar activo aunque no sea invocado
  - Puede crear proceso de núcleo que ejecuta en su propio contexto
- Proceso/hilo de núcleo: es un proceso más en el sistema pero
  - Ejecuta sólo código del SO
  - No tiene mapa de memoria de usuario asociado
  - Puede realizar operaciones de bloqueo
  - Pero no acceder a direcciones de memoria de usuario
- Para evitar proliferación de procesos de núcleo
  - Colas predefinidas de trabajos servidas por procesos de núcleo
    - En vez de crear un nuevo proceso de núcleo, se encola trabajo
    - Linux workqueues

Definición de contexto atómico

- Como recapitulación sobre los contextos de ejecución
- Contexto atómico si se cumple **alguna** de estas condiciones:
  - Rutina de interrupción de un dispositivo
  - Rutina de interrupción software
  - Prohibidas las interrupciones de los dispositivos
  - Inhibidas las interrupciones software.
  - Deshabilitada la expulsión de procesos
  - En posesión de un *spinlock*
- Sólo se puede hacer un bloqueo
  - Si en contexto **no atómico**
- Sólo se puede acceder a mapa de usuario
  - Si en contexto **no atómico y no se trata de proceso de núcleo**

Desarrollo de manejadores en micronúcleos

- Se eliminan "peculiaridades" en su desarrollo
  - Biblioteca del lenguaje completa
  - Uso de llamadas al sistema
    - Además de los servicios proporcionados por el micronúcleo
  - Depuración convencional
  - Error en manejador sólo afecta al acceso a ese dispositivo
    - Puede activarse nueva versión del manejador sobre la marcha
  - Uso de memoria convencional
    - Memoria paginable y pila sin restricciones
- Mayor productividad y menor propensión a errores
- Menor eficiencia
  - Más paso de mensajes y cambios de proceso

Estructura del manejador en micronúcleos

- Programa servidor convencional (¡tiene su main!)
  - Bucle que espera mensajes
    - Ops. implementadas por manejador (lect., escr., ...) son mensajes
    - Interrupciones también como mensajes
    - Micronúcleo ofrece servicios para que proceso reserve IRQ
      - Cuando se produce int., micronúcleo envía mensaje a ese proceso
  - Al recibir mensaje, comprueba su tipo y lo procesa
  - El servidor puede ser concurrente
    - Sincronización igual que cualquier programa de usuario
  - Manejador realiza directamente accesos PIO/MMIO
    - Micronúcleo ofrece servicios para habilitar acceso a los mismos

Bibliografía

- *Linux Device Drivers*, Jonathan Corbet, Alessandro Rubini, y Greg Kroah-Hartman. O'Reilly Media, 3ª edición, 2005
- *Building Embedded Linux Systems*, Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, y Philippe Gerum, O'Reilly Media, 2ª edición, 2008
- *Understanding the Linux Kernel,* Daniel P. Bovet y Marco Cesati. O'Reilly Media, 3ª edición, 2005
- *Programming Embedded Systems*, Michael Barr y Anthony Massa. O'Reilly Media, 2006
- *Designing Embedded Hardware*, John Catsoulis, O'Reilly Media, 2005
- *Sistemas Operativos: Una visión aplicada*. J. Carretero, P. de Miguel, F. García y F. Pérez. McGraw-Hill, 2ª edición, 2007
- *Gestión de procesos*. F. Pérez Costoya. http://laurel.datsi.fi.upm.es/~ssoo/DSO4/gestion_de_procesos.pdf