Módulos

LIN - Curso 2015-2016

J.C. Sáez, M. Prieto
High Performance Computing Division
ArTeCs Group
Universidad Complutense de Madrid

Contenido

Contenido

Módulos cargables del kernel Linux (I)

¿Qué es un módulo cargable?

- Un "fragmento de código" que puede cargarse/descargarse en el mapa de memoria del SO (kernel) bajo demanda
- Sus funciones se ejecutan en modo kernel (privilegiado)
  - Cualquier error fatal en el código "cuelga" el SO
  - Herramientas de depuración menos elaboradas
    - `printk()`: Imprimir mensajes en fichero de log del kernel
    - `dmesg` : Muestra contenido del fichero de log del kernel
    - FTrace: `trace_printk()`, `/sys/kernel/debug/tracing`

También existe soporte para módulos cargables en otros sistemas tipo UNIX (BSD, Solaris) y en MS Windows

Módulos cargables del kernel Linux (II)

Ventajas de los módulos del kernel

1. Reducen el *footprint* del kernel del SO
   - Cargamos únicamente los componentes SW (módulos) necesarios
2. Permiten extender la funcionalidad del kernel en caliente (sin tener que reiniciar el sistema)
   - Mecanismo para implementar/desplegar *drivers*
3. Permiten un diseño más *modular* del sistema

Módulos Cargables (II)

- Los módulos disponibles para nuestro kernel se encuentran en el directorio `/lib/modules/${KERNEL_VERSION}`
  - `KERNEL_VERSION=$(uname -r)`
- Podemos saber qué módulos están cargados con `lsmod`
  - `/proc/modules`

Terminal

```
kernel@debian:~$ lsmod
Module                  Size  Used by
binfmt_misc             6293  1
uinput                  7034  1
nfsd                  198027  2
auth_rpcgss            38062  1 nfsd
oid_registry            2131  1 auth_rpcgss
exportfs                3444  1 nfsd
nfs_acl                 2159  1 nfsd
nfs                   154443  0
lockd                  55678  2 nfs,nfsd
fscache                37683  1 nfs
sunrpc                173693  6 nfs,nfsd,auth_rpcgss,lockd,nfs_acl
fuse                   67222  2
vmhgfs                 43005  1
```

Anatomía de un módulo cargable

- Código fuente puede constar de uno o varios ficheros .c y .h
  - Podría incluir también ficheros con código ensamblador (.s)
- En lugar de una función `main()`, un módulo tiene funciones `init` y `cleanup`
  - `init`: Se invoca cuando se carga el módulo en el kernel
  - `cleanup`: Se ejecuta al eliminar/descargar el módulo
- Por defecto, las funciones `init` y `cleanup` deben definirse como sigue:

```
int init_module(void);      /* init */
void cleanup_module(void);  /* cleanup */
```

- Esta asociación puede alterarse mediante las macros `module_init()` y `module_exit()`

Ejemplo simple

```c
#include <linux/module.h> /* Requerido por todos los módulos */
#include <linux/kernel.h> /* Definición de KERN_INFO */
MODULE_LICENSE("GPL"); /* Licencia del módulo */

/* Función que se invoca cuando se carga el módulo en el kernel */
int modulo_lin_init(void)
{
    printk(KERN_INFO "Modulo LIN cargado. Hola kernel.\n");

    /* Devolver 0 para indicar una carga correcta del módulo */
    return 0;
}

/* Función que se invoca cuando se descarga el módulo del kernel */
void modulo_lin_clean(void)
{
    printk(KERN_INFO "Modulo LIN descargado. Adios kernel.\n");
}

/* Declaración de funciones init y cleanup */
module_init(modulo_lin_init);
module_exit(modulo_lin_clean);
```

Gestión de Módulos (I)

- Al compilar el módulo se genera un fichero `.ko` que es un fichero objeto ELF (*Executable and Linkable Format*) especial

Carga y descarga de módulos

- Para cargar el módulo se usa el comando `insmod`

```
$ insmod mi_modulo.ko
```

- Un módulo puede descargarse con `rmmod`

```
$ rmmod mi_modulo
```

- Solo el administrador (*root*) puede ejecutar ambos comandos
  - Aconsejable utilizar `sudo <comando>`
  - Introducir la password del usuario kernel: kernel

Compilación de Módulos (I)

- Compilación gestionada mediante un fichero *Makefile*
  - Es necesario tener instalados los ficheros de cabecera (*headers*) del kernel en ejecución
    - Ya instalados en la máquina virtual
  - Compilar con "`make`" y borrar los restos de compilación con "`make clean`"

Makefile (módulo de un solo fichero .c)

```
obj-m =  mimodulo.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Advertencia

La ruta donde se almacena el Makefile/fuentes NO puede contener espacios

Compilación de Módulos (II)

Makefile (varios módulos)

```
obj-m =  modulo1.o modulo2.o modulo3.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Compilación de Módulos (III)

- Si el módulo consta de varios ficheros (ej: fichero1.c, fichero2.c y fichero3.c) es necesario construir un fichero objeto intermedio

Makefile

```
obj-m =  multimod.o  #multimod.c no ha de existir
multimod-objs =  fichero1.o fichero2.o fichero3.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Ejemplo: compilación, carga y descarga (I)

Terminal

```
kernel@debian:~/FicherosP1/MiModulo$ ls
Makefile  mimodulo.c
kernel@debian:~/FicherosP1/MiModulo$ make
make -C /lib/modules/3.14.1.lin/build M=/home/kernel/FicherosP1/MiModulo modules
make[1]: se ingresa al directorio `/usr/src/linux-headers-3.14.1.lin'
  CC [M]  /home/kernel/FicherosP1/MiModulo/mimodulo.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/kernel/FicherosP1/MiModulo/mimodulo.mod.o
  LD [M]  /home/kernel/FicherosP1/MiModulo/mimodulo.ko
make[1]: se sale del directorio `/usr/src/linux-headers-3.14.1.lin'
kernel@debian:~/FicherosP1/MiModulo$ sudo insmod mimodulo.ko
kernel@debian:~/FicherosP1/MiModulo$ lsmod | head
Module                  Size  Used by
mimodulo                 965  0
binfmt_misc             6293  1
uinput                  7034  1
nfsd                  198027  2
auth_rpcgss            38062  1 nfsd
oid_registry            2131  1 auth_rpcgss
exportfs                3444  1 nfsd
nfs_acl                 2159  1 nfsd
nfs                   154443  0
```

Ejemplo: compilación, carga y descarga (II)

Terminal

```
kernel@debian:~/FicherosP1/MiModulo$ dmesg | tail
[   11.776723] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
[   13.807911] input: ACPI Virtual Keyboard Device as /devices/virtual/input/input6
[   16.116601] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[   16.118127] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[   16.119224] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[  816.167004] Modulo LIN cargado. Hola kernel.
kernel@debian:~/FicherosP1/MiModulo$ sudo rmmod mimodulo
kernel@debian:~/FicherosP1/MiModulo$ dmesg | tail
[   11.776723] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
[   13.807911] input: ACPI Virtual Keyboard Device as /devices/virtual/input/input6
[   16.116601] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[   16.118127] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[   16.119224] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[  816.167004] Modulo LIN cargado. Hola kernel.
[  995.874297] Modulo LIN descargado. Adios kernel.
kernel@debian:~/FicherosP1/MiModulo$ lsmod | head
Module                  Size  Used by
binfmt_misc             6293  1
uinput                  7034  1
nfsd                  198027  2
auth_rpcgss            38062  1 nfsd
oid_registry            2131  1 auth_rpcgss
...
```

Gestión de Módulos (II)

Enlace y símbolos del kernel

- ¿Cómo se produce el enlace de un módulo?
  - El kernel mantiene tablas de símbolos (variables, funciones)
    - Secciones `__ksymtab, __ksymtab_gpl` ...
    - `cat /proc/kallsyms`
- ¿Qué símbolos pueden utilizarse dentro de un módulo?
  - Aquellos explícitamente *exportados* por el kernel o por otros módulos

Gestión de Módulos (III)

Símbolos del kernel

- Los símbolos del kernel tienen tres niveles de visibilidad:
  1. `static`: visibles sólo dentro del mismo fichero fuente
  2. `extern`: visibles desde cualquier fichero fuente utilizando en la construcción del kernel
  3. `exported`: visible/disponible para cualquier modulo cargable – *exported kernel interface* / API del kernel – . Macros:
     - `EXPORT_SYMBOL` (cualquier módulo)
     - `EXPORT_SYMBOL_GPL` (sólo con licencia compatible GPL)
     - La sentencia de exportación del símbolo debe estar en el ".c" donde se define

Ejemplo exportación (.c)

```c
#include <linux/export.h> /* IMPORTANTE */

...

/* Función que deseamos exportar */
int foo(void) {

    /** Cuerpo de la función **/

    ...

    return 0;
}
EXPORT_SYMBOL(foo);

...
```

Gestión de Módulos (IV)

Conocer información (macros)

- modinfo

Instalación o Eliminación "inteligente"

- modprobe (wrapper de insmod y rmmod)
  - Busca por defecto en /lib/modules/${KERNEL_VERSION}
  - Tiene en cuenta dependencias

Inventario de dependencias

- Los módulos también pueden exportar símbolos (en sus secciones ___ksymtab ...) para ser utilizados por otros.
- Para gestionar dichos símbolos es conveniente generar un inventario de dependencias
  - depmod (/lib/modules/${KERNEL_VERSION}/modules.dep)

Contenido

Sistema de ficheros /proc

- `/proc` es un sistema de ficheros virtual
  - No ocupa espacio en disco
- Al leer/escribir de/en un "fichero" particular de este sistema (entrada /proc) se ejecuta una función del kernel que devuelve/recibe los datos
  - Lectura: *read callback*
  - Escritura: *write callback*
- En Linux, /proc muestra información de los procesos, uso de memoria, módulos, hardware, ...
- También puede emplearse como mecanismo de interacción de propósito general entre el usuario y el kernel
  - Los módulos pueden crear entradas /proc para interactuar con el usuario

Interfaz de operaciones de entrada /proc

Interfaz de Operaciones

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t,
         loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned
          long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
         unsigned long, loff_t);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long)
         ;
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
....
};
```

Creación de nueva entrada /proc (I)

- Crear un módulo del kernel con funciones `init_module()` y `cleanup_module()`
- Definir variable global de tipo `struct file_operations`
  - Especifica qué operaciones /proc se implementan y su asociación con las funciones del módulo

```
struct file_operations fops = {
    .read = myproc_read,    //read()
    .write = myproc_write,  //write()
};
```

- Implementar las operaciones de la interfaz para conseguir la funcionalidad deseada
  - Semántica similar a la llamada al sistema pertinente

Creación de nueva entrada /proc (II)

- En la función de inicialización, crear entrada /proc con la función `proc_create()`:

```
struct proc_dir_entry *proc_create(const char *name,
                                   umode_t mode,
                                   struct proc_dir_entry *parent,
                                   const struct file_operations *ops);
```

Parámetros

- `name`: Nombre de la entrada
- `mode`: Máscara octal de permisos (p.ej., 0666)
- `parent`: Puntero al directorio padre (`NULL` → directorio raíz)
- `ops`: Puntero a la estructura que define las operaciones

Valor de retorno

- Devuelve un puntero al descriptor de la entrada creada

Creación de nueva entrada /proc (III)

- En la función *cleanup* del módulo, eliminar la entrada /proc creada

```
void remove_proc_entry(const char *name,
                       struct proc_dir_entry *parent);
```

Parámetros

- `name`: Nombre de la entrada
- `parent`: Puntero al directorio padre

Interfaz /proc (I)

- Read Callback: Datos de salida – se lee la entrada (ej. `cat`)

```
static ssize_t read_proc_proto(struct file *filp,
          char __user *buf, size_t len, loff_t *off);
```

Parámetros

- `filp`: Estructura que describe al *fichero abierto* en Linux
- `buf`: puntero al array de bytes donde escribimos
  - Puntero al espacio usuario
  - Típicamente se usa como cadena de caracteres pero podría ser cualquier tipo de datos
- `len`: número de bytes que podemos escribir como máximo en `buf`
- `off`: puntero de posición (parámetro de entrada y salida)
  - Debemos actualizarlo correctamente (p.ej., `(*off)+=len;`)

Interfaz /proc (I)

- Read Callback: Datos de salida – se lee la entrada (ej. `cat`)

```
static ssize_t read_proc_proto(struct file *filp,
          char __user *buf, size_t len, loff_t *off);
```

Valor de retorno

- Número de bytes realmente leídos (devueltos por el kernel) o negativo (error)
  - 0: *end of file* (no hay más información que devolver)
  - < 0: error
  - > 0: se podría volver a llamar de nuevo a la función

Interfaz /proc (III)

- Write Callback: Datos de entrada – se escribe la entrada (ej. `echo`)

```
static ssize_t write_proc_proto(struct file *filp,
       const char __user *buf, size_t len, loff_t *off)
```

Parámetros

- `filp`: Estructura que describe al *fichero abierto* en Linux
- `buf`: puntero al array de bytes donde el usuario pasa los datos
  - Puntero al espacio usuario
- `len`: Número de bytes o caracteres almacenados en `buf`
- `off`: puntero de posición (parámetro de entrada y salida)
  - Debemos actualizarlo correctamente (p.ej., `(*off)+=len;`)

Interfaz /proc (III)

- Write Callback: Datos de entrada – se escribe la entrada (ej. echo)

```
static ssize_t write_proc_proto(struct file *filp,
       const char __user *buf, size_t len, loff_t *off)
```

Valor de retorno

- Número de bytes escritos (procesados por el kernel) o error (< 0)

Copia espacio usuario $\Longleftrightarrow$ espacio kernel (I)

- Las operaciones `read()` y `write()` de una entrada `/proc` aceptan como parámetro un puntero al buffer del proceso de usuario (espacio de usuario)
  - Parámetro marcado con el modificador `__user`
- No debemos confiar en los punteros al espacio de usuario
  - Puntero nulo
  - Región de memoria a la que el proceso no tiene acceso

Copia espacio usuario $\Longleftrightarrow$ espacio kernel (II)

- Siempre se ha de trabajar con una copia privada de los datos en espacio de kernel
  - Por ejemplo, declarar array `char kbuf[MAX_CHARS]` local a las funciones read() y write()
  - En `read()`: trabajar sobre `kbuf` + copiar contenido de `kbuf` a buffer de usuario con `copy_to_user()`
  - En `write()`: copiar datos de buffer de usuario a `kbuf` (con `copy_from_user()`) + realizar procesamiento sobre `kbuf`

Copia espacio usuario $\Longleftrightarrow$ espacio kernel (III)

<asm/uaccess.h>

```
unsigned long copy_from_user(void *to, const void __user *from,
                             unsigned long n);
unsigned long copy_to_user(void __user *to, const void *from,
                           unsigned long n);
```

- Semántica de copia similar a `memcpy()`
- Ambas funciones devuelven el número de bytes que NO pudieron copiarse

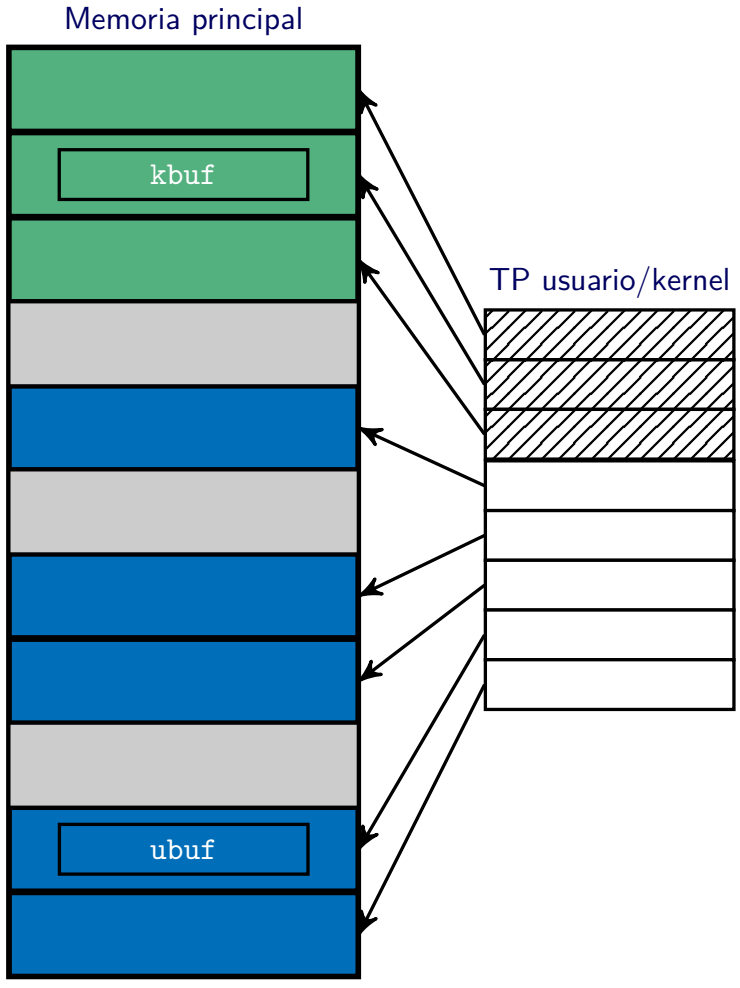
Uso de copy_from_user

```
int main()
{
  char ubuf[128];
  int bytes;
  int fd=open("/proc/my_entry",O_WRONLY);

  ... Comprobación errores ...
  ... Inicialización de ubuf ...

  bytes=write(fd,ubuf,strlen(buf));

  ...
  return 0;
}
```

TRAP →

Página de proceso de usuario

Página del kernel



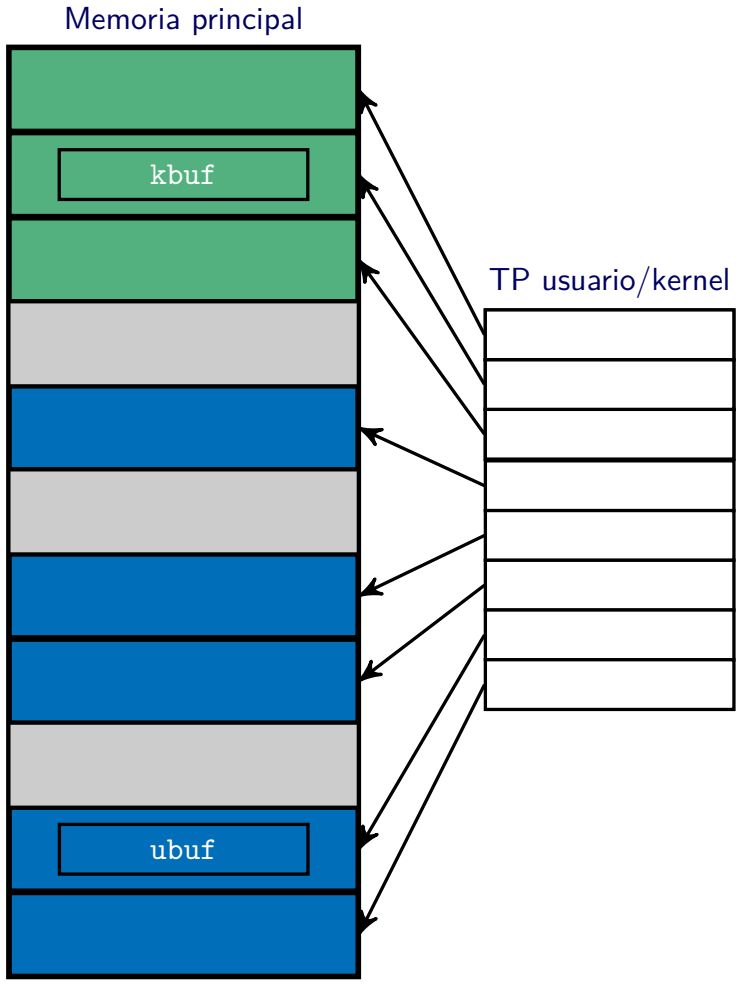
Uso de copy_from_user

```
int main()
{
  char ubuf[128];
  int bytes;
  int fd=open("/proc/my_entry",O_WRONLY);

  ... Comprobación errores ...
  ... Inicialización de ubuf ...

  bytes=write(fd,ubuf,strlen(buf));

  ...
  return 0;
}
```

TRAP →

Página de proceso de usuario

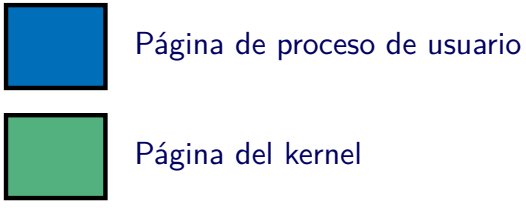
Página del kernel

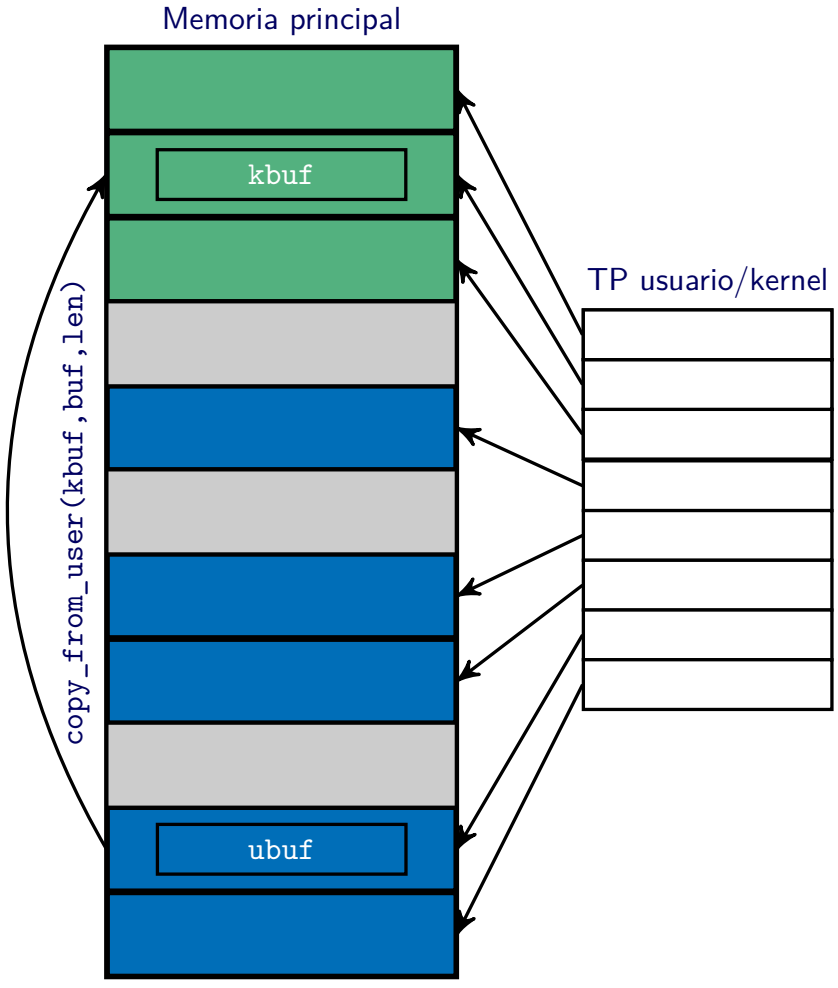


Uso de copy_from_user

```
int main()
{
  char ubuf[128];
  int bytes;
  int fd=open("/proc/my_entry",O_WRONLY);

  ... Comprobación errores ...
  ... Inicialización de ubuf ...

  bytes=write(fd,ubuf,strlen(buf));

  ...
  return 0;
}
```

TRAP →





Otras funciones útiles

Manejo de Cadenas

`sprintf`, `strcmp`, `strncmp`, `sscanf`, `strcat`, `memset`, `memcpy`, `strtok`, ...

Reservar y liberar memoria dinámica <linux/vmalloc.h>

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

Consultar dónde estan definidas usando un buscador del kernel

Contenido

Listas doblemente enlazadas en C (I)

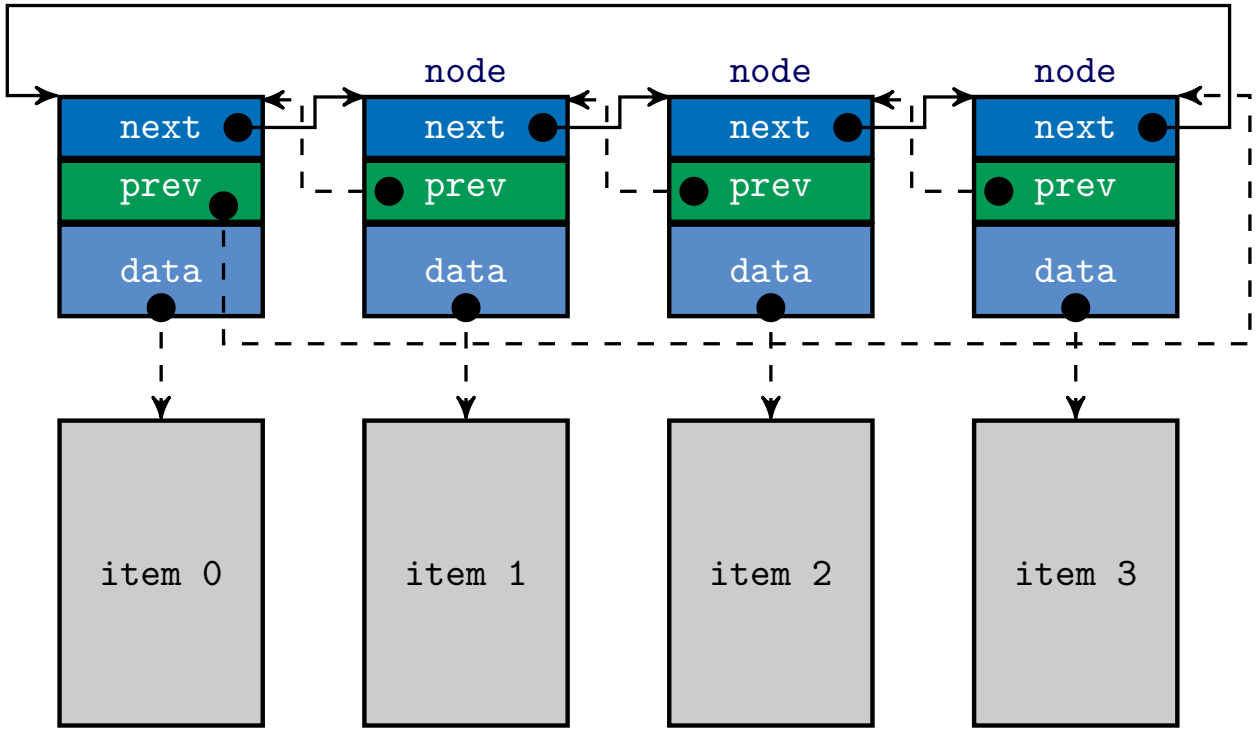
```
struct node {
    struct node *next;
    struct node *prev;
    void* data;
};
```

```
struct list {
    struct node *first;
    int nr_items;
    ...
};
```

Listas doblemente enlazadas en C (I)

```
struct node {
    struct node *next;
    struct node *prev;
    void* data;
};
```

```
struct list {
    struct node *first;
    int nr_items;
    ...
};
```



Listas doblemente enlazadas en C (II)

```
struct node {
    struct node *next;
    struct node *prev;
    void* data;
};
```

```
struct list {
    struct node *first;
    int nr_items;
    ...
};
```

- Esta implementación requiere solicitar memoria dinámica en cada inserción (struct node)
- No es adecuado en entornos donde la gestión de memoria dinámica es problemática
  - Sistemas de tiempo real
  - kernel del SO

Listas doblemente enlazadas en C (II)

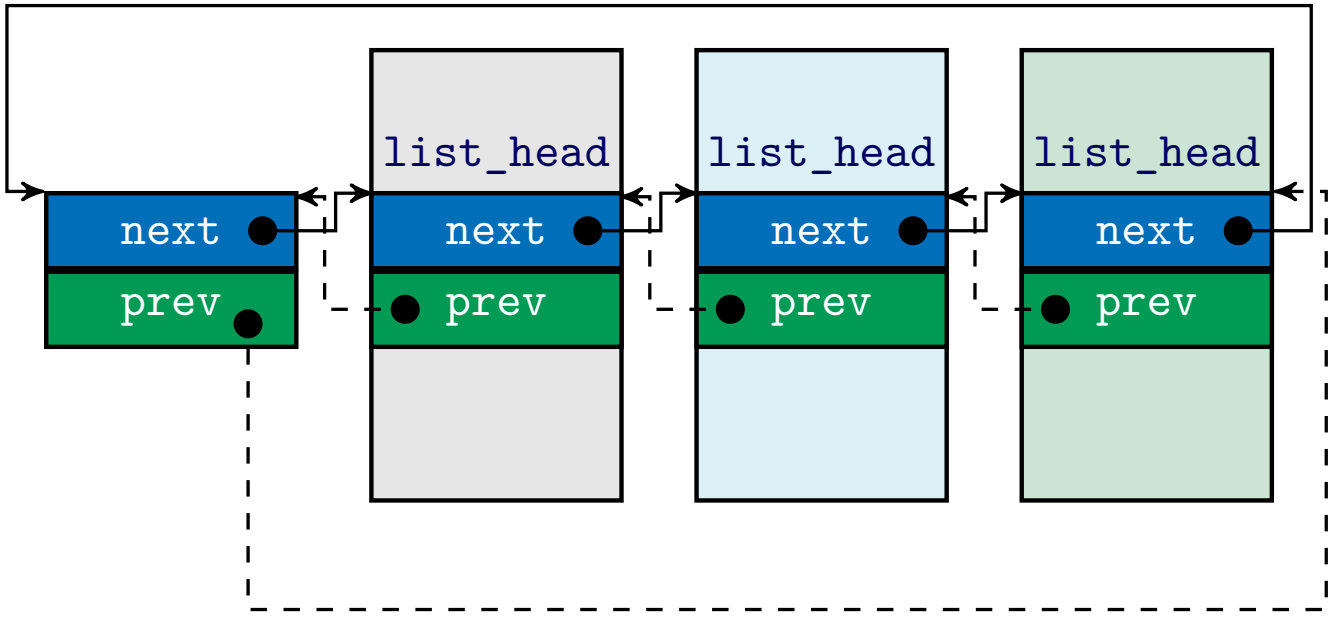
```
struct node {
    struct node *next;
    struct node *prev;
    void* data;
};
```

```
struct list {
    struct node *first;
    int nr_items;
    ...
};
```

- Esta implementación requiere solicitar memoria dinámica en cada inserción (struct node)
- No es adecuado en entornos donde la gestión de memoria dinámica es problemática
  - Sistemas de tiempo real
  - kernel del SO
- **Solución**: Incorporar los enlaces `prev` y `next` directamente en las estructuras que se enlazan

Listas doblemente enlazadas en Linux (I)

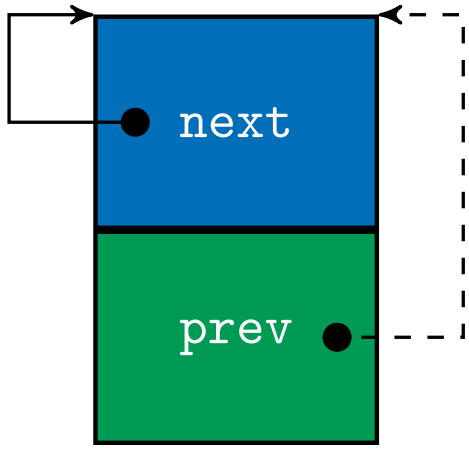
- `struct list_head`: implemementación genérica de lista doblemente enlazada
  - Estructura con sólo dos campos: `next` y `prev`
- Los elementos/estructuras que forman parte de la lista han de contener un campo tipo `struct list_head` (enlaces)



Listas doblemente enlazadas en Linux (II)

<linux/list.h>

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



Función/Macro	Descripción
`LIST_HEAD(name)`	En declaración, Inicializa una variable tipo `struct list_head` con nombre `name`
`INIT_LIST_HEAD(plist`[1]`)`	Inicializa la lista pasada como parámetro
`list_add(node,plist)`	Insertar nodo al principio de la lista
`list_add_tail(node,plist)`	Insertar nodo al final de la lista
`list_del(node)`	Eliminar nodo de la lista
`list_empty(plist)`	Consultar si lista es vacía

[1] `plist` ha de ser un puntero a `struct list_head`.

Listas doblemente enlazadas en Linux (III)

- `list_for_each(pos,head)`

```
#define list_for_each(pos, head) \
    for(pos = (head)->next; pos != (head); pos = pos->next)
```

- `list_entry(pointer,type,name)`
  - Permite recuperar el puntero a la estructura a partir de los enlaces
  - Devuelve la dirección de la estructura de datos de tipo `type` en la que se incluye un campo `list_head` con el nombre `name` y cuya dirección es `pointer`
- `list_for_each_safe(pos, aux, head)`
  - Versión segura de `list_for_each` que permite eliminación de un nodo durante el recorrido de la lista
  - Declarar variable `struct list_head* aux` para almacenar el si-guiente al nodo actual

Ejemplo

```
struct list_item {
    int data;
    struct list_head links;
};

struct list_head my_list;

void do_something(struct list_head* list) { ... }

void print_list(struct list_head* list) {
    struct list_item* item=NULL;
    struct list_head* cur_node=NULL;

    list_for_each(cur_node, list) {
    /* item points to the structure wherein the links are embedded */
        item = list_entry(cur_node, struct list_item, links);
        printk(KERN_INFO "%i\n",item->data);
    }
}

void f(void) {
    INIT_LIST_HEAD(&my_list); /* Initialize the list */
    do_something(&my_list); /* Populate the list */
    print_list(&my_list);
}
```

Referencias

- The Linux Kernel Module Programming Guide (Cap. 1, 2 y 3):
  - `http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html`
  - `http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf`
- Linux Kernel Development
  - Cap. 6 *"Kernel Data Structures"*
    - Información sobre listas enlazadas
  - Cap. 17 *"Devices and Modules"*
    - Más información sobre módulos

Licencia

LIN - Módulos
Versión 0.2

Este documento (o uno muy similar) está disponible en
`https://cv4.ucm.es/moodle/course/view.php?id=62472`