Kernel Modules

Modern Operating Systems, by Andrew Tanenbaum

Chap 10.5.5

The Linux Kernel Module Programming Guide, by Salzman, Burian and Pomerantz

OS Internals

- So far, we have USED the operating system
  - We've studied about processes, IPC, Threads, Locks, Semaphores, etc.

- Now let's open up the patient

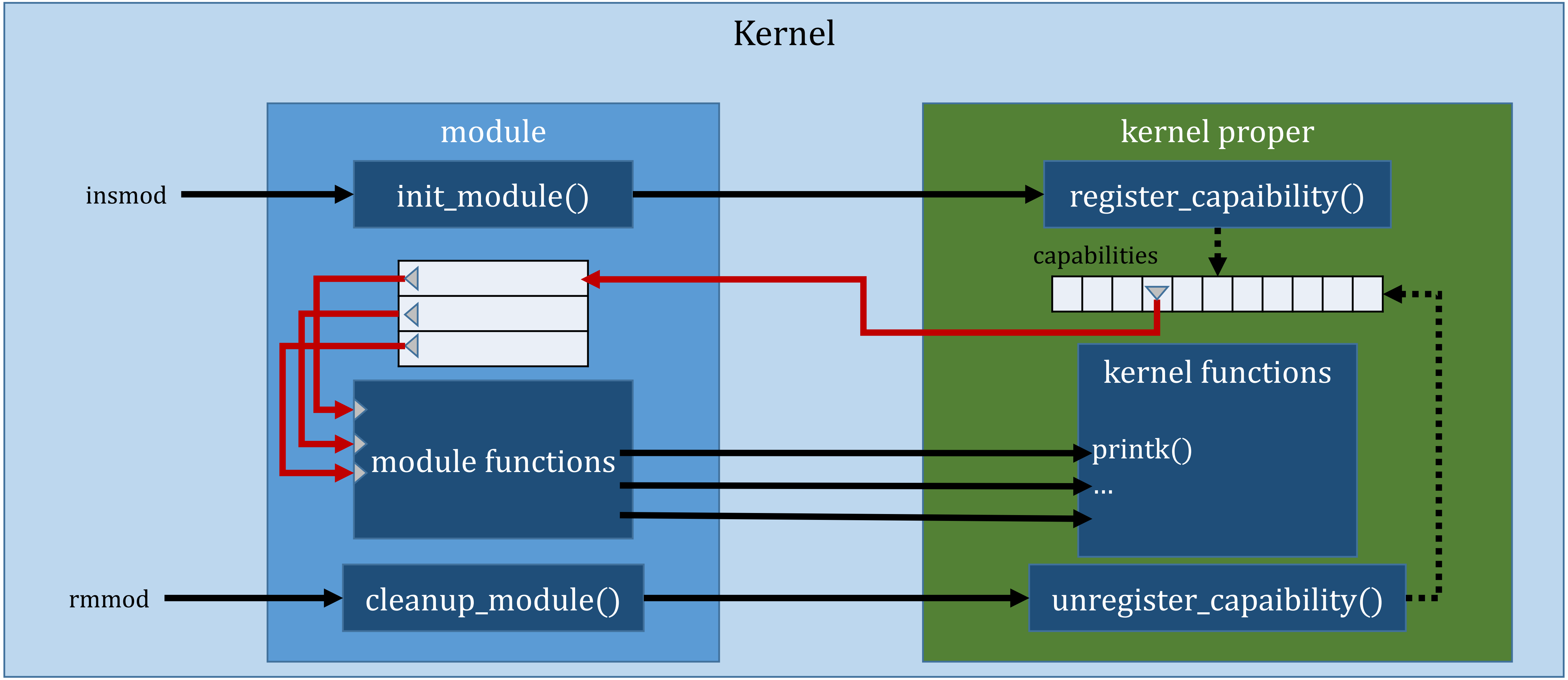
Kernel

- The kernel contains privileged code
  - Code which is allowed to do things ordinary programs are not allowed to do
- Kernel contains "trusted" software
  - With great power comes responsibility
  - We trust the kernel to be fair, honest, and discreet
- Problem: Kernel size
  - We want the kernel to do more and more for us – more devices, etc.
  - We don't want the kernel to take over the world!

Kernel Modules

- Divide the kernel up into "kernel proper" and kernel modules
  - "kernel proper" is the base kernel
- Enable dynamic loading and unloading of kernel modules
  - Load a module ONLY when it is needed
  - Unload a module when you no longer need it
- Reduces kernel size, frees up memory and other resources
- Enables independent kernel development
  - For instance, different modules for device drivers for different devices

Kernel Module Mechanics



Hello World Kernel Module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

// called when module is installed
int __init init_module() {
	printk(KERN_ALERT "mymodule: Hello World!\n");
	return 0;
}

// called when module is removed
void __exit cleanup_module() {
	printk(KERN_ALERT "mymodule: Goodbye, cruel world!!\n");
}
```

See [kernel module examples]

Compiling the Module

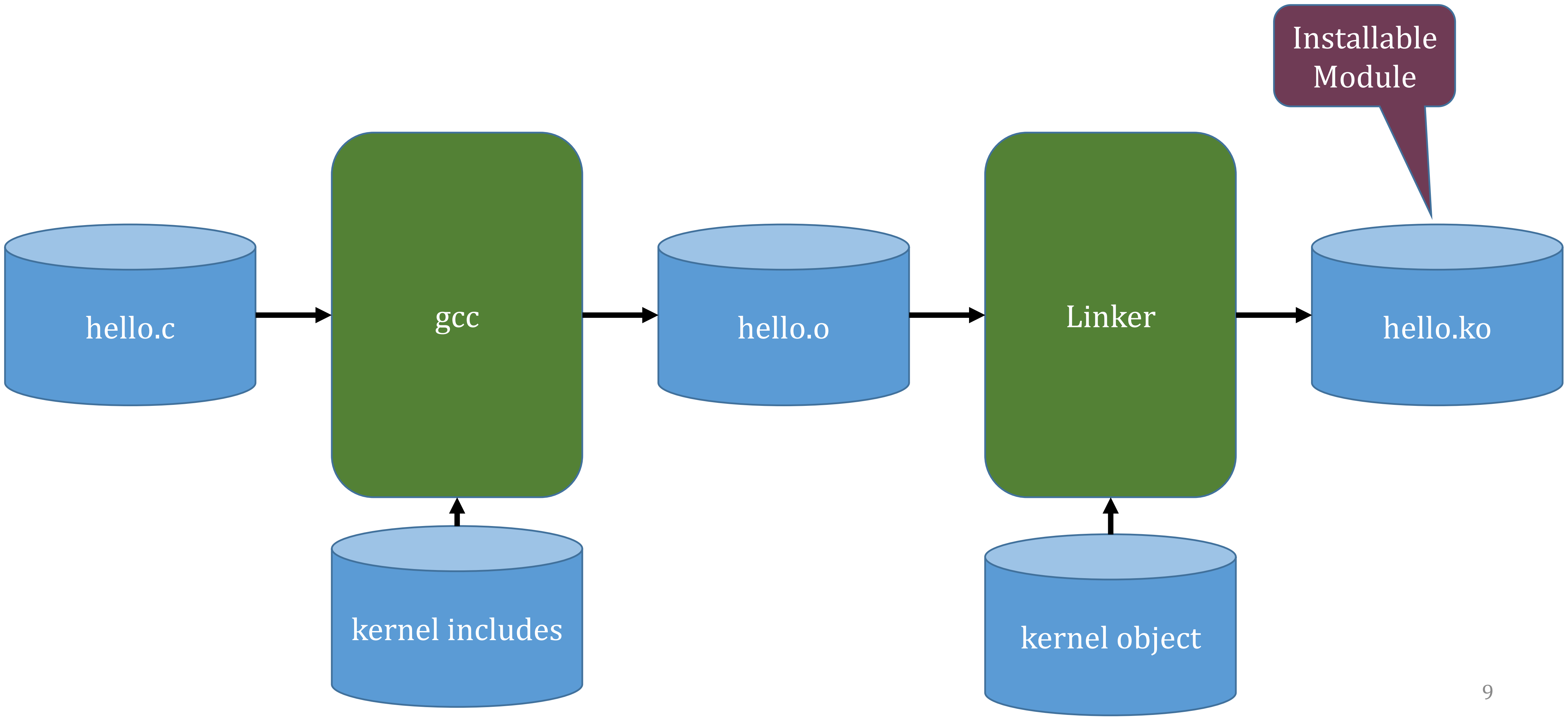
- Makefile
  - obj-m := testmod.o
  - module_objs := file1.o file2.o …. # For multiple files
- Compile:
  - > make –C /lib.modules/($uname –r)/build M=`pwd` modules

- More information on kernel Makefiles
  - https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt
  - https://www.kernel.org/doc/Documentation/kbuild/modules.txt

Two-pass Makefile

```
ifeq ($(KERNELRELEASE),)  # If KERNELRELEASE is not specified
    # This is the local (first part) of the make.
    # It has recipes for making and installing modules as well as cleaning up
    # by default, "modules" target is built
    KERNELDIR ?= /lib/modules/$(shell uname –r)/build
modules:
        $(MAKE) –C $(KERNELDIR) M=($PWD) modules
    # Invokes make in the kernel directory… this is a kernel make
    # Makefile in KERNELDIR invokes this Makefile, which uses the "else" clause
else
    obj-m := hello.o # Make knows how to do the rest! (including KERNRELEASE)
endif
```

Compiled Kernel Modules



Kernel Module Utilities

> sudo insmod hello.ko # "Inserts" (dynamically loads) a module

- Calls sys_init_module()
- Calls vmalloc() to allocated kernel memory
- Copies module binary to memory
- Resolves kernel references (e.g. printk) via kernel symbol table
- Calls module's initialization function

> modprobe hello.ko # Same as insmod, but installs references too

>sudo rmmod hello # Removes a module

- Fails if module is still being used

>sudo lsmod # Tells what modules are current loaded

- Internally, reads /proc/modules

Linux Kernel Licensing

- Linux Kernel licensed with the GNU General Public License (GPL)
  - See https://en.wikipedia.org/wiki/GNU_General_Public_License
  - Allows users to modify software as they see fit
  - Requires source code to be distributed with the binaries
- Question: Does a kernel module fall under the GPL license?
  - Device drivers do not have to be licensed under GPL, but the mainstream drivers are
  - See https://lwn.net/Articles/154602/ for the difference between EXPORT_SYMBOL and EXPORT_SYMBOL_GPL

Module Coding Hints

- Modules can call other kernel functions
  - such as printk, kmalloc, kfree, etc.
  - But only those that are exported by the kernel using EXPORT(name)
  - See /proc/kallsyms for a list of kernel symbols exported
- Kernel Code (including modules) CANNOT call user library functions
  - Such as malloc, free, printf, etc.
- Kernel Code should not include standard header files
  - Such as stdio.h, stdlib.h, etc.
- Segmentation fault in the kernel can crash the entire system!
  - Often harmless in user space
- The version of the kernel is compiled into a module
  - Need to recompile for each version of the kernel it can be linked to

Concurrency Issues

- Different processes can invoke your module concurrently
  - Different parts of your module can be active at the same time
- Device interrupts can trigger Interrupt Service Routines (ISRs)
  - ISR may access data that your module uses as well
- Kernel timers can execute concurrently with your module
  - May access common data
- You may have a symmetric multi-processor (SMP) system
  - Multiple processes may be executing your code *simultaneously*
- Module (and most kernel code) must be re-entrant
  - Capable of multiple simultaneous executions

Error Handling

register/unregister take a pointer and name

```
int __init  my_init_function(void){
    int err=register_A(ptr1,"skull");
    if (err) goto fail_A;
    err=register_B(ptr2,"skull");
    if (err) goto fail_B;
    err=register_C(ptr3,"skull");
    if (err) goto fail_C;
    return 0; // success
    fail_C: unregister_B(ptr2,"skull");
    fail_B: unregister_A(ptr1,"skull");
    fail_A: return err; // propagate
    error
}
```

```
void __exit my_cleanup_function(void) {
    unregister_C(ptr3,"skull");
    unregister_B(ptr2,"skull");
    unregister_A(ptr1,"skull");
    return;
}
```

In case of failure, need to unregister every *successful* registration

Module Parameters

Parameter name

type

permissions

- In hellon.c:

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
MODULE_PARM_DESC(howmany,"Number of times to print msg");
module_param(whom, charp, S_IRUGO);
```

- Command Line> `insmod hellon.ko howmany=10 whom="Class"`

- Description printed out with >modinfo hellon.ko
- See example: [hellon.c]

Character Devices in Linux

Implementing a device driver using kernel modules

Devices

- All devices connected to your computer are registered with the kernel, and can be seen by inspecting the virtual file system: /dev

- Each device is assigned a Major number and a Minor number

```
csvb@CS550-tbartens:~$ ls -l /dev/tty*
crw-rw-rw- 1 root tty     5,   0 Feb 17 13:25 /dev/tty
crw--w---- 1 root tty     4,   0 Feb 17 11:01 /dev/tty0
...
```

Major Number

Minor Number

'c' for character
'b' for block

- Major number corresponds to device driver software
  - see linux source Documentation: devices.txt
- Minor number controls variations in the hardware

Device Drivers

- Kernel software responsible for bridging between standard operating system device mechanisms, and the actual hardware
- From an OS point of view, there are three kinds of device drivers:
  - Character oriented drivers, such as keyboard and mouse
  - Block oriented devices such as hard disks, CD drives
  - Network (message oriented) devices such as network interface cards
  - (Others, such as USB, SCSI, Firewall, I2O often variations on one of the above)

Device Communication (w/ kernel)

- Character (char) devices
  - Read/Write a single byte at a time
  - Use a byte-stream abstraction
- Block devices
  - Read/Write a fixed block size chunk of data
  - Often use buffers to imitate character (byte-stream) abstraction
- Network devices
  - Read/Write packets of varying size
    - Size of packet included in the packet header
  - Message abstraction

Character device drivers

- OS expects each character device driver to implement a set of pre-defined functions (e.g. open, close, read, write, lseek, ioctl, …) known as ***file operations***

- In linux, there is a structure, struct file_operations, defined in "fs.h" that consists of function pointers to each file operation

- When you register a character device driver, you must supply a file_operations structure so the kernel knows what to call

"Miscellaneous" Devices

- Character devices used for simple device drivers
- Major number = 10
- Each device gets it's own minor number
  - Requested at registration (mkmod) time

Implementing a misc. device driver

- Step 1: Declare a device struct

```
static struct miscdevice my_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = “my device",
    .fops = &my_fops
};
```

Implementing a misc. device driver

- Step 2: Declare a file operations structure

```
static struct file_operations my_fops= {
    .owner = THIS_MODULE,
    .open = my_open,
    .close = my_close,
    …
    .llseek = noop_llseek
};
```

- Uninitialized function pointers get a sensible default value

Implementing a misc. device driver

- Step 3: Register the device in module_init function

```
static int __init my_module_init(){
    misc_register(&my_misc_device);
}
```

- Registration creates an entry in /dev for "mydevice"
  - and connects file operations to my_fops

Implementing a misc. device driver

- Step 4: Implement the fops functions

```
static ssize_t my_read(struct file *file, char __user * out, size_t size, loff_t * off) {
    ....
    sprintf(buf, "Hello World\n");
    copy_to_user(out, buf, strlen(buf)+1);
    ....
}
```

- Don't forget to:
  - allocate memory for buf,
  - check if "out" points to valid user memory using access_OK()
  - check for errors after copy_to_user()

Implementing a misc. device driver

- Step 5: Don't forget to unregister the device when removing module

```
static void __exit my_exit(void) {
    misc_deregister(&my_misc_device);
    ...
}
```

After installing your device driver module

- User then opens the device:

```
fd = open("/dev/mydevice",O_RDWR);
```

- OS then invokes my_open(inode,file) which returns a file descriptor

Moving data in and out of the Kernel

- Each process has it's own address space
- The kernel has a kernel address space
  - All kernel modules and the base kernel reside in this address space

`unsigned long copy_to_user(void __user *dst,const void *src,unsigned long n);`

  - Copies from **kernel** space to **user** space
  - Checks target is writable by `access_ok(dst,VERIFY_WRITE)`
    - If the result is true (non-zero), copy proceeds
  - Returns number of bytes that could not be copied (success=0)

`unsigned long copy_from_user(void *dst,const void __user *src,unsigned long n);`

  - Copies from **user** space to **kernel** space

Managing kernel heap space

- kmalloc() : allocates physically contiguous memory

```
void * kmalloc(size_t size,int flags);
```

- kzalloc() : allocates memory and sets it to zero
- vmalloc() : allocates virtually contiguous memory
  - may not be physically contiguous

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

- kfree() : deallocation