Kernel and Driver Fundamentals

Reference:

"Linux Device Drivers", Alessandro Rubini, O'Reilly, 2nd Edition

Free on-line version at:

http://www.xml.com/ldd/chapter/book/

Pointers:

http://www.redhat.com/
http://www.kernel.org/
ftp://ftp.kernel.org/
ftp://sunsite.unc.edu/pub/Linux/docs/
ftp://tsx-11.mit.edu/pub/linux/docs/
http://www.ssc.com/
http://www.conecta.it/linux/

Driver examples:

ftp://ftp.ora.com/pub/examples/linux/drivers/

Kernel and Driver Fundamentals

Device drivers provide **mechanisms**, not **policy**.

Mechanism: "Defines what capabilities are provided?"

Policy: "Defines how those capabilities can be used?"

This strategy allows *flexibility*.

The driver controls the hardware and provides an abstract interface to its capabilities.

The driver ideally imposes **no** restrictions (or *policy*) on how the hardware should be used by applications.

For example, X manages the graphics hardware and provides an interface to user programs.

Window managers implement a particular policy and know nothing about the hardware.

Kernel apps build policies on top of the driver, e.g. floppy disk, such as who has access, the type of access (direct or as a filesystem), etc.

Floppy is policy free -- it makes the disk look like an array of blocks.

Kernel and Driver Fundamentals

Kernel Parts:

Process management:

Kernel is responsible for creating, destroying and scheduling processes.

Memory management:

Kernel implements a virtual memory space on top of the limited physical resources.

File systems:

Almost everything in Unix can be treated as a file (file abstraction).

The kernel builds a structured filesystem on top of unstructured hardware.

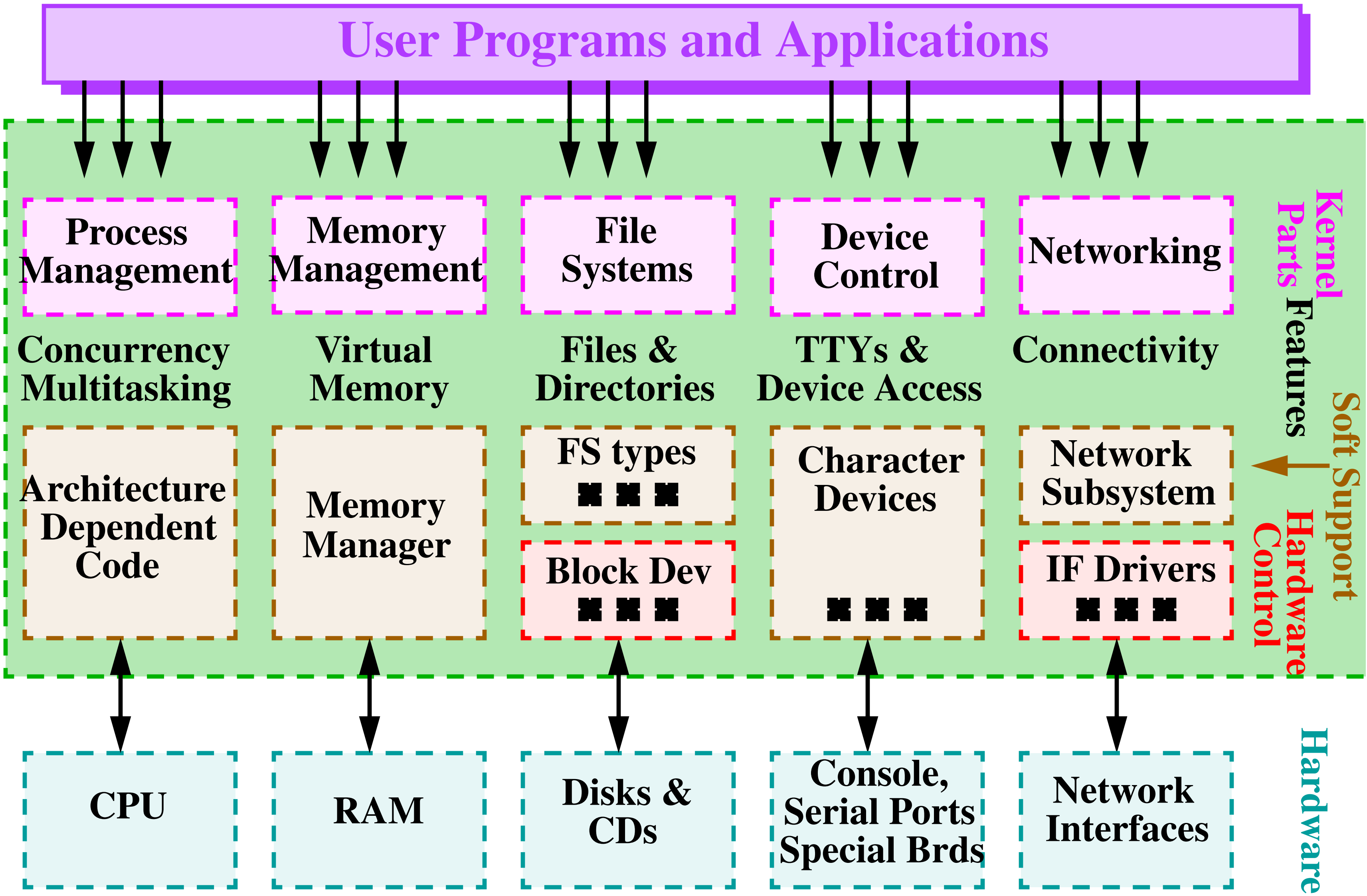
Device control:

The kernel must have a device driver for every peripheral present on the system.

Networking:

Kernel collects, identifies, dispatches and receives data packets to/from network interfaces and user programs.

Kernel and Driver Fundamentals



Kernel and Driver Fundamentals

Modules:

A method by which you can *expand* the kernel code at run time.

A module is made up of *object code* (not stand-alone code) that can be linked (*insmod*) and unlinked (*rmmod*) to a running kernel.

This provides a nice way for you to install and test device drivers.

Device classification:

Most device drivers can be classified into one of **three** categories.

- *Character devices*.

  Console and parallel ports are examples.

  Implement a stream abstraction with operations such as *open*, *close*, *read* and *write* system calls.

  File system nodes such as */dev/tty1* and */dev/lp1* are used to access character devices.

  Differ from regular files in that you usually cannot step backward in a stream.

Kernel and Driver Fundamentals

Device classification:

- *Block devices*.

  A block device is something that can host a filesystem, e.g. disk, and can be accessed only as multiples of a block.

  Linux allows users to treat block devices as character devices (*/dev/hda1*) with transfers of **any** number of bytes.

  Block and character devices differ primarily in the way data is managed **internally** by the kernel at the *kernel/driver* interface.

  The difference between block and char is transparent to the user.

- *Network interfaces*.

  In charge of sending and receiving data packets.

  Network interfaces are not stream-oriented and therefore, are not easily mapped to a node in the filesystem, such as */dev/tty1*.

  Communication between the kernel and network driver is not through read/write, but rather through packet transfer functions.

Building and Running Modules

Hello World:

```
#define MODULE
#include <linux/module.h>

int init_module(void)
   {
   printk("<1>Hello, world\n"); /* <1> is priority. */
   return 0;
   }

int cleanup_module(void)
   {
   printk("<1>Goodbye cruel world\n");
   }
```

Building and Running Modules

Hello World:

You must be superuser to install and remove modules.

To compile and run this code saved in a file named *hw.c*, use:

- root# *gcc -c hw.c*
- root# *insmod hw.o*
- root# *rmmod hw*

On my version of Linux (Redhat distribution 7.2, kernel version 2.4.2), the printk messages are saved at the bottom of the log file:

/var/log/messages

Module versus applications:

Unlike an application, a module registers itself (so it can be invoked by the kernel when needed).

init-module() and *cleanup_module()* are module entry points.

They take care of initialization and cleanup.

Building and Running Modules

Modules versus applications:

insmod links the program to the kernel.

The link step for an application gives the program access to library functions, such as those defined in *libc*.

Note that modules do NOT have access to *libc* functions, only those exported by the kernel.

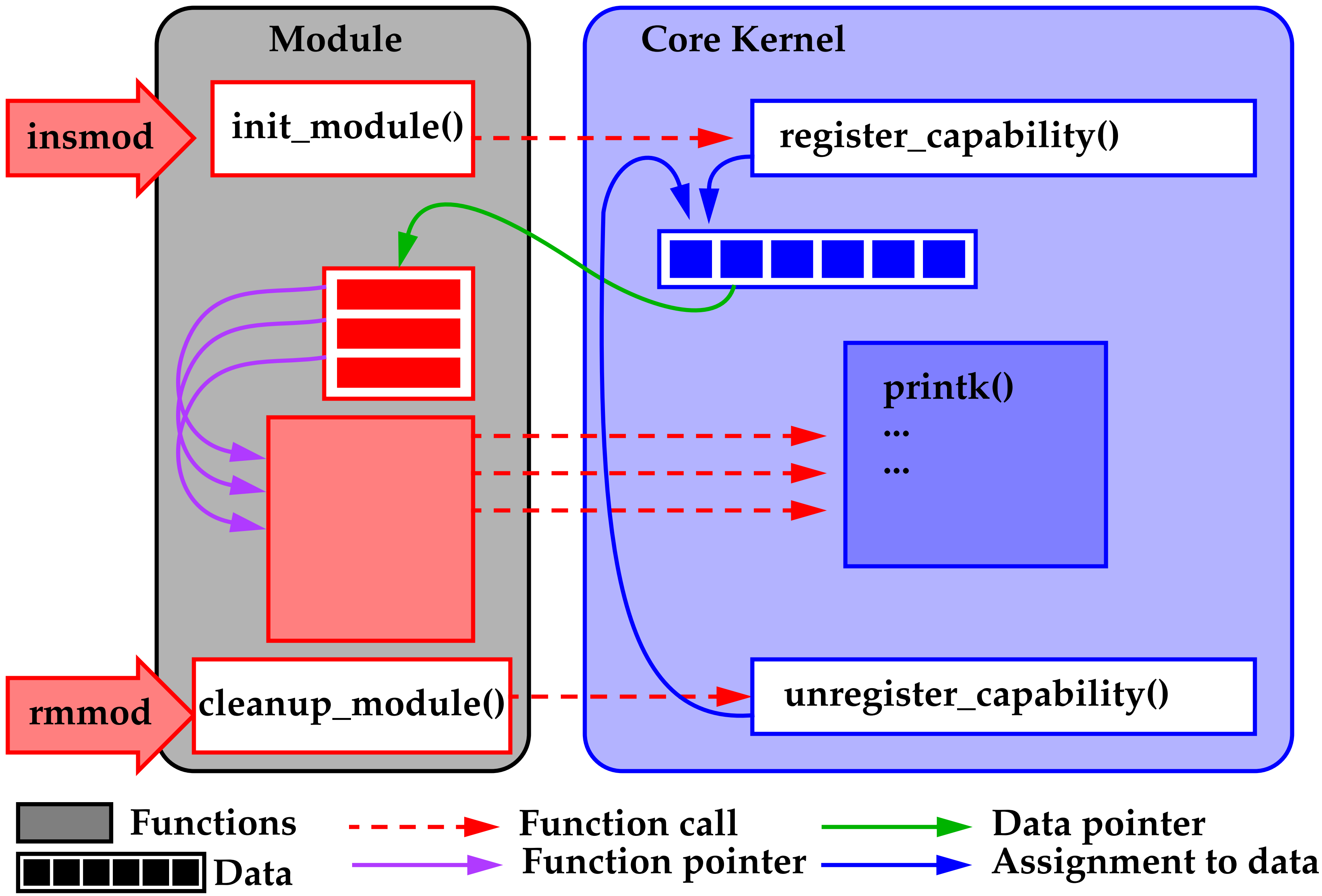
For example, *printk* is the kernel version of *printf* (except there's no floating point support).

Modules can NOT include standard header files in */usr/include* except the kernel headers in */usr/include/linux* and */usr/include/asm*.

Kernel code in these headers is protected with #*ifdef __KERNEL__* since applications can include these headers for their other info.

Note these directories are symbolic links to */usr/src/linux/include*.

Building and Running Modules



Building and Running Modules

Modules versus applications:

Name space pollution:

Name space pollution results from the presence of many (badly named) functions and global variables that are difficult to track.

Remember, a module is added to a really big application, the kernel.

Declare most symbols as **static** and use a *well-defined prefix* for global symbols.

You can also declare a **symbol table** to avoid this (discussed later).

Fault handling:

Application segmentation faults are harmless, since you can print out the core dumps or use a debugger :)

A kernel fault is **fatal** to the current process and sometimes the whole system !

Building and Running Modules

Modules versus applications:

Applications run in *user space* while modules run in *kernel space*.

In user mode, the processor inhibits direct access to hardware and unauthorized access to memory.

Applications switch to kernel mode through a limited number of gates, implemented as **system calls** and **hardware interrupts**.

- **System calls** allow the kernel to access a process's data (the process that called it.)
- **Hardware interrupts** are asynchronous and are not associated with a particular process.

Drivers typically provide code for both of these tasks.

Building and Running Modules

Concurrency in the kernel:

Drivers should support *concurrency*, e.g., the ability to support calls by two different processes simultaneously.

This requires the driver (and kernel) to maintain **distinct data structures** for each process (since the same code is executed) or to ensure shared data is not corrupted.

Driver code must be *reentrant*.

Reentrant code is code that does not keep status in global variables, but rather in local (stack allocated) variables.

This allows the process executing to suspend (e.g., wait for keyboard data) and other processes to execute the same code.

It is not a good idea to assume your code won't be interrupted.

Kernel code cannot be preempted (at this point in time anyway), but SMP (symmetric multiprocessors) will allow multiple simultaneous copies to be run.

Building and Running Modules

Concurrency in the kernel:

There are several approaches to keeping data separate.

Kernel **global** variables is one approach:

current is a pointer to **struct task_struct** (*linux/sched.h*), which refers to the currently executing user process.

A module can refer to this global variable as in:

```
printk("The process is \"%s\" (pid %i)\n",
   current->comm, current->pid);
```

This wont link, of course, without #*include <linux/sched.h>*.

Compiling and Loading Modules

Makefile:

- Define statements:

  `#define __KERNEL__`

  This allows header declarations included between #*ifdef* __*KERNEL*__ to be compiled.

  `#define MODULE`

  Define this before the include </*linux/module.h*>.

- Compiler flags:

  -*O* must be specified, since many function are declared as *inline* in the headers, and gcc doesn't expand them unless optimization is turned on.

  Don't use -*O2* !

  -*g* can be used for debugging.

  -*Wall* for warnings is suggested.

Compiling and Loading Modules

Makefile:

- Multiple source files:

  If you choose to split your code across multiple source files, then *ld -r* is required to link them.

A makefile for a simple driver called *skull* which uses two source files.

```
KERNELDIR  = /usr/src/linux

include $(KERNELDIR)/.config

CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)-O -Wall

ifdef CONFIG_SMP
 CFLAGS += -D__SMP__ -DSMP
endif
all: skull.o

skull.o: skull_init.o skull_clean.o
  $(LD) -r $^ -o $@

clean:
 rm -f *.o *~ core
```

Compiling and Loading Modules

Loading:

- *insmod* is similar to *ld*.
  - It links **unresolved** symbols in the module to the symbol table of the running kernel.
- *insmod* also differs from *ld*.
  - It doesn't modify the disk image, only the **in-memory image**.

- *insmod* takes parameters that allows *on-the-fly* configuration of the driver, which is preferred over compile time configuration.

Version dependency:

- You will likely need to recompile the module for each version of the kernel that it is linked to.

- Each module defines a symbol called *__module_kernel_version*.
  - *insmod* matches this against the version number of the current kernel.

- Newer kernels define it for you in *<linux/module.h>*