Khóa học : “Basic Embedded Linux”

❖ **Email** : *training.laptrinhnhung@gmail.com*

❖ **Website** : *http://hethongnhung.com* – *http://laptrinhnhung.com*

Linux Device Driver

Device Drivers

- **Introduction to Device Drivers**

  - Device Drivers are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works.

  - User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver.

  - This programming interface is such that drivers can be built separately from the rest of the kernel, and "plugged in" at runtime when needed.

  - They are often called as "Modules"

Device Drivers

• Concept of Modules

The Device Drivers are again classified into two

- ***Static Drivers***

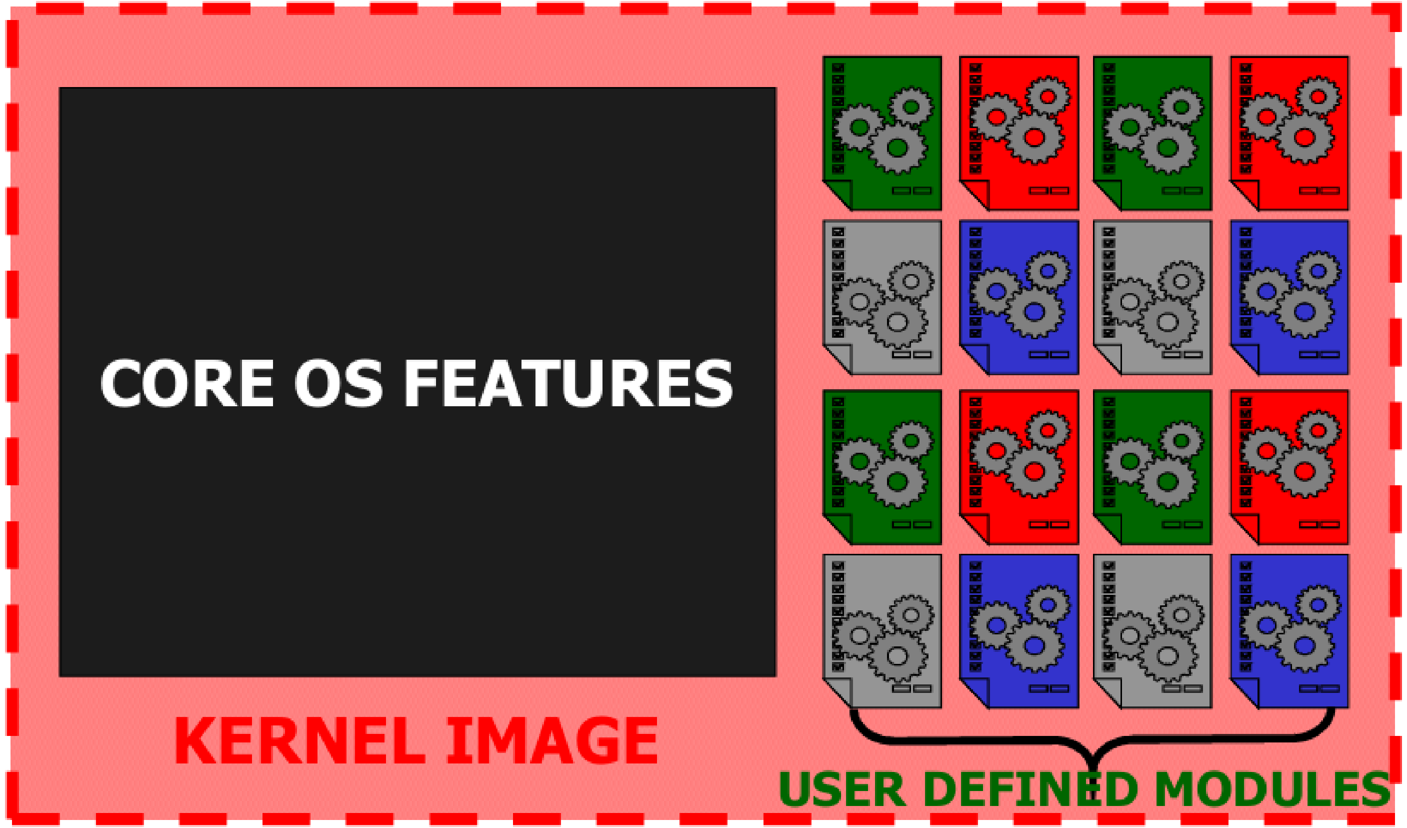
  Static drivers are compiled as part of the kernel and is available at anytime.

- ***Dynamic Drivers***

  Dynamic modules are compiled as modules and are loaded as and when necessary. This method has the advantage that it uses the Memory more efficiently than the Statically linked drivers.

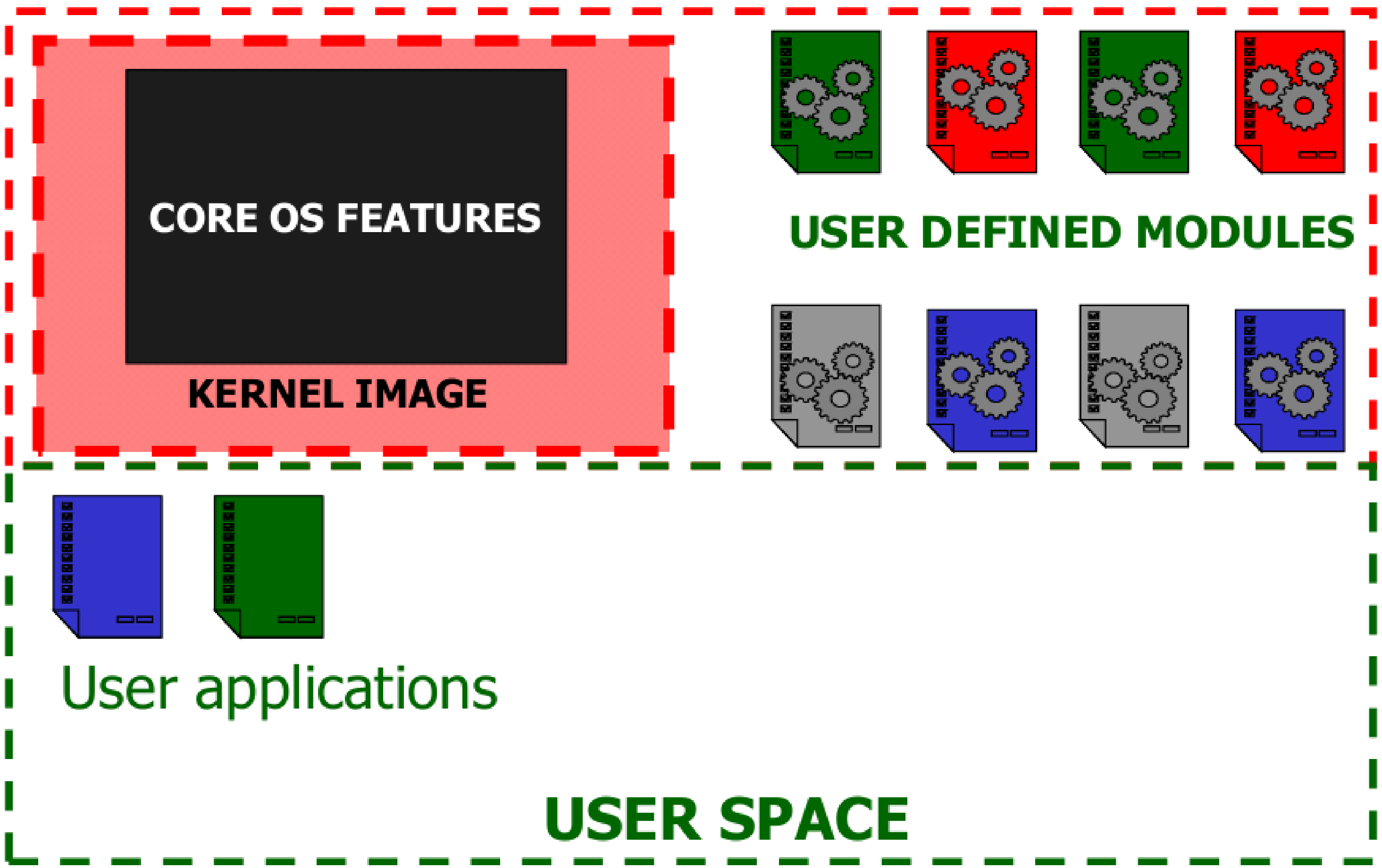
Static Modules

- **Concept of Static modules**



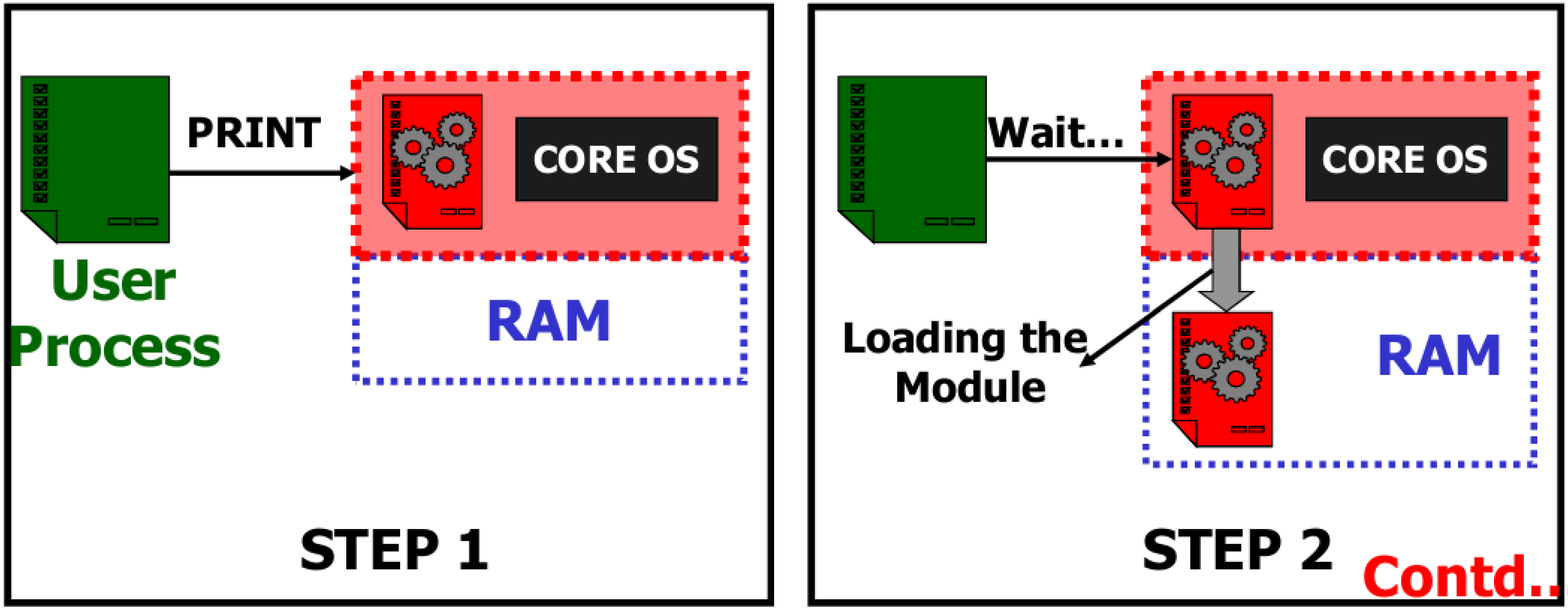
Dynamic modules

- **Concept of dynamic modules**



Dynamic modules

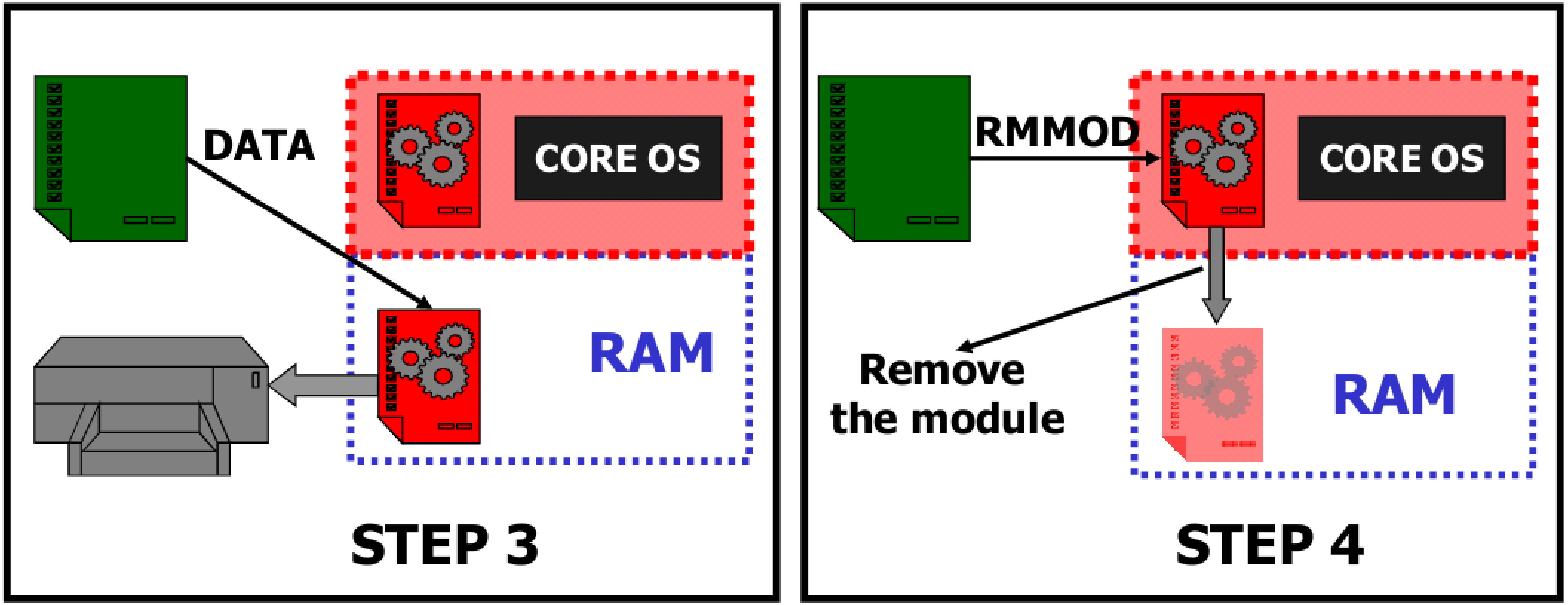
- Loading and unloading dynamic modules
  - Eg: A Printer module
    - A printer module is only loaded when a “Print” command is issued by the User



Contd...

Dynamic modules

- Loading and unloading dynamic modules Contd…



Modules

- **The Structure of a Module**

Initialization
(Registering the Module)

Actual functions
(Open, Read, Write.etc)

Cleanup
(Unregistering the Module)

Modules

Creating and using the Modules

- While creating a module, it must have the structure as described in the previous section
  - ***Initialization***

    init_module()

  - ***Actual functions***

    open(),read(),write(),ioctl(),etc..

  - ***Cleanup***

    cleanup_module()

Modules

Compiling the Module

- After creating the Source file, it must be compiled to get a final OBJECT (*.obj) file.
- Since we are creating a MODULE, we must provide these options during compilation
  - -D__KERNEL__

    Indicates we are creating a Kernel utility
  - -DMODULE

    Indicates we are creating a Module
  - So the compilation option may look like

    $:> gcc -D__KERNEL__ –DMODULE –I/<my linux source path>/ -Wall –c module_source.c –o final_module.o

The path may look like: /usr/src/linux-2.4.20-8/include/

Modules

Installing the Module

- After creating the Object module, we have to install the Module. For this we use the following command
  - $:> insmod final_module.o

- If the module is successfully installed, it will be listed in the special file /proc/modules
  - $:>cat /proc/modules
  - Or $:>lsmod

Removing a Module

- After the functions inside the modules are used, these modules can be set free if those functions are not used again.

- This is done by calling the *cleanup_module()* function inside the module which will unregister the specific module from the memory

  $:> rmmod final_module

Kernel Symbol Tables

Details about the Kernel Symbol table

- The Kernel Symbol Table contains the addresses of global kernel items – Functions and Variables – that are needed to implement modularized drivers
- The public symbol table can be read from /proc/ksyms
- When a module is loaded, any symbol exported by the module becomes part of the Kernel Symbol Table and will appear at the /proc/ksyms
- New modules can use symbols exported by the module and can stack the modules on top of the others.

The I/O Ports

- The basic function of a Driver is to provide the access to I/O Ports or I/O Memory.

- The driver should should be guaranteed to perform the I/O operations without any interference from other drivers

- This is accomplished by the request/free mechanism for I/O regions.

- Information about the Ports can be found at */proc/ioports* and about the I/O Memory at /proc/iomem

The I/O Ports

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
01f0-01f7 : ide0
02f8-02ff : serial(auto)
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
0cf8-0cff : PCI conf1
5000-500f : Intel Corp. 82801AA SMBus
c000-cfff : PCI Bus #01
  c000-c0ff : D-Link System Inc RTL8139 Ethernet
    c000-c0ff : 8139too
d000-d01f : Intel Corp. 82801AA USB
  d000-d01f : usb-uhci
d400-d4ff : Intel Corp. 82801AA AC'97 Audio
  d400-d4ff : Intel ICH 82801AA
d800-d83f : Intel Corp. 82801AA AC'97 Audio
  d800-d83f : Intel ICH 82801AA
f000-f00f : Intel Corp. 82801AA IDE
  f000-f007 : ide0
  f008-f00f : ide1
```

The I/O Memory

```
00000000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-07efffff : System RAM
  00100000-0024b8dd : Kernel code
  0024b8de-003457a3 : Kernel data
d0000000-d3ffffff : Intel Corp. 82810 CGC [Chipset Graphics Controller]
d4000000-d40fffff : PCI Bus #01
  d4000000-d40000ff : D-Link System Inc RTL8139 Ethernet
    d4000000-d40000ff : 8139too
d4100000-d417ffff : Intel Corp. 82810 CGC [Chipset Graphics Controller]
ffb00000-ffffffff : reserved
```

Resource allocation

The functions used for I/O registry

- The following functions are used for accessing the I/O
  - `int check_region(unsigned long start, unsigned long length)`
  - `struct resource *request_region(unsigned long start, unsigned long length, char *name)`
  - `void release_region(unsigned long start, unsigned long length)`

Resource allocation

The Resource structure

- The Resource ranges are described via a resource structure declared in <linux/ioport.h>

```
struct resource
{
   const char *name;
   unsigned long start,end;
   unsigned long flags;
   struct resource *parent,*sibling,*child:
}
```

Resource allocation

The functions used for I/O registry

- As far as the driver is concerned the functions used are
  - `int check_mem_region(unsigned long start, unsigned long length)`
  - `int req_mem_region(unsigned long start, unsigned long length, char *name)`
  - `int release_mem_region(unsigned long start, unsigned long length)`

Address Space

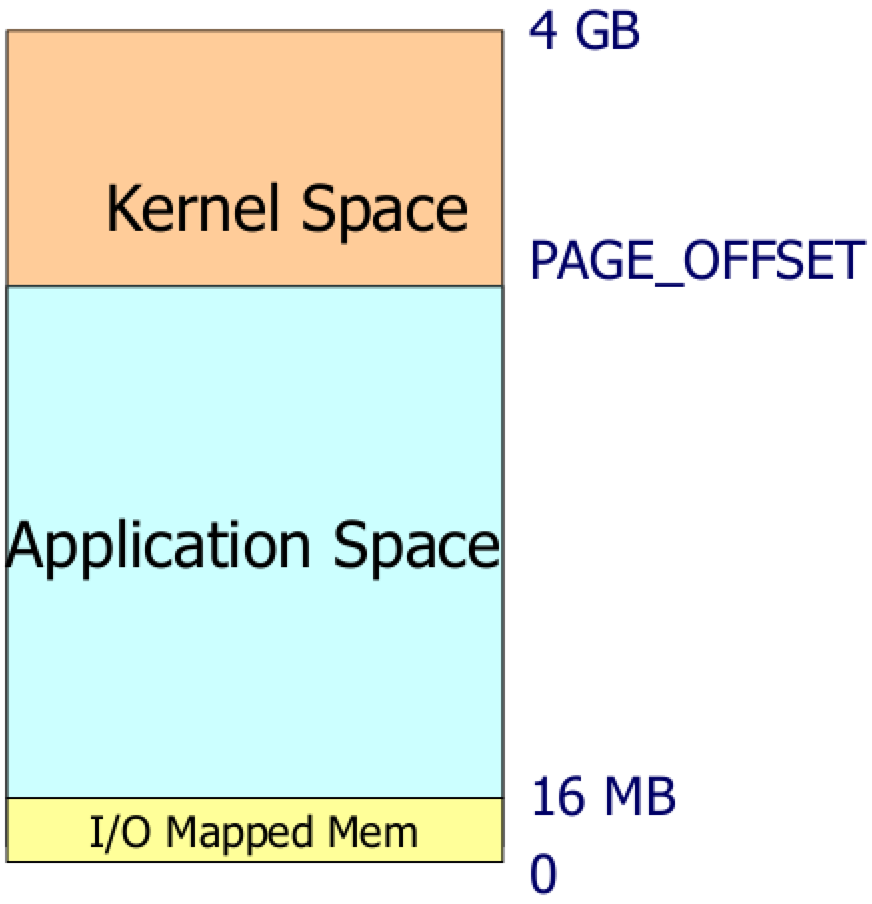
The entire addressable space of memory is split it into two major areas

- Kernel Space
- User Space

- PAGE_OFFSET defines the split and can be configured in **`asm/page.h`**

- The kernel space is located above the offset and the user space below it. The default value of PAGE_OFFSET is 0xc0000000 so kernel has memory for user space.

Address Space



Driver Memory Allocation

- Memory is allocated in chunks of the PAGE_SIZE on the target machine, for Intel platforms it is 4Kb.
- One way of allocate memory for driver is by
  - `unsigned long __get_free_page(int gfp_mask)` which allocates exactly one page
  - The `gfp_mask` describes priority and attributes of the page
    - `GFP_ATOMIC` Memory should be returned without blocking or bringing in pages from swap
    - `GFP_KERNEL` The call may block if pages need to be swapped out.
    - `GFP_DMA` The memory returned should be below 16MB, suitable for DMA

Driver Memory Allocation

`__get_free_pages`

- `unsigned long __get_free_pages(int gfp_mask, unsigned long order)` is used to allocate multiple pages in orders of 2.

- Memory has to be freed by the module itself, since allocated memory wont be reaped by the kernel when the module is unloaded. Memory pages are freed using
  - `void free_page(unsigned long addr)`
  - `void free_pages(unsigned long addr, unsigned long order)`

Driver Memory Allocation

kmalloc vmalloc

- Instead as pages, memory is allocated as bytes using

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

- Memory is freed using

```
void kfree(const void *addr)
```

- get_free_page and kmalloc return memory that is physically contiguous. vmalloc is used to get memory contiguous in the virtual address space

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

- vmalloc allows to allocate larger arrays than kmalloc, but the returned memory can only be used within the kernel

Data Transfer Between Spaces

- Applications are not allowed to alter memory space of kernel.

- Data is transferred between user and kernel space using functions:

  - `get_user(void *x, const void *addr)`
  ```
  //copies sizeof(addr) bytes from user space address addr to x.
  ```
  - `put_user(void *x, const void *addr)`
  ```
  //copies sizeof(addr) bytes to user space to variable x from addr.
  ```

Data Transfer Between Spaces

Data can be transferred in large amounts between spaces using:

```
copy_to_user(void *to, void *from, unsigned long size)
```

- Copies data residing in kernel address space between addresses pointed by *from and *to.

- Similarly data can be transferred from user using

```
copy_from_user(void *to, void *from, unsigned long size)
```

Questions?