

INTRODUCTION TO LINUX WORLD

PART 1
Mukul Bhardwaj

TOPICS

- Introduction about Linux
- Cross-development Environment
- Linux configuration
- Introduction to Device Driver Framework
 - Char driver example
- LSP
 - Linux porting
 - Linux boot-up flow
- Kernel
 - Processes
 - Scheduler
 - Memory manager
- System Calls

TOPICS (cont)

- VFS
- IPC
- Kernel preemption
- kernel synchronization
- Interrupt
- Timer
- Network
- Reference/books

INTRODUCTION TO LINUX

- Monolithic / Microkernel
- Dynamic Modules support
- Kernel threading
- Multi-threaded application support
- Preemptive kernel
- Multi-processor
- File system
- Virtual memory
- User Space/Kernel Space
- Shared Library
- Multistack networking including IPv4/IPv6

LINUX VERSION NUMBERING

- Two flavors .. Stable or Development
- Stable .. Suitable for Deployment
- Development Kernel .. Undergo rapid Change

LINUX VERSION NUMBERING(CONT)

- 2 . 5 . 1
- Major Version is 2
- This is the first release
- Minor Version is five

LINUX VERSION NUMBERING(CONT)

- Current version
 - 2.6.X series
 - 2.6.25
- Current Pattern

LINUX DIRECTORY STRUCTURE

- /
 - This is referred to as the root directory
- /bin
 - Contains essential programs for the operating system in executable form
- /boot
 - This directory, as the name suggests, contains the boot information for Linux, including the kernel.
- /dev
 - In Linux, devices are treated in the same way as files, so they can be read from and written to in the same way. So, when a device is attached, it will show up in this folder. Bear in mind that this includes every device in the system, including internal motherboard devices and more.

LINUX DIRECTORY STRUCTURE(CONT)

- /mnt
 - this is the directory in which storage devices are "mounted."
- /proc
 - This "virtual" directory contains a lot of fluid data about the status of the kernel and its running environment.
- /root
 - Rather than being part of the /home directory, the superuser (or root user)'s directory is placed here
- /sbin
 - This is where system administration software is stored

LINUX DIRECTORY STRUCTURE(cont)

- /etc
 - This is where system administration software is stored
- /home
- /lib
- /tmp
- /usr
- /var

LINUX SOURCE CODE

- Kernel.org
 - Download full version source code
- What is Patch?

SOURCE CODE LAYOUT

- Arch
- Documentation
- Drivers
- Fs
- Include
- Init
- IPC

SOURCE CODE LAYOUT(cont)

- Kernel
- Lib
- Mm
- Net
- Scripts
- Security
- Sound
- Usr

Domains

- Bootloader
- LSP
- Device drivers
- Tool-Chain
- File-system

User Applications

O/S Services

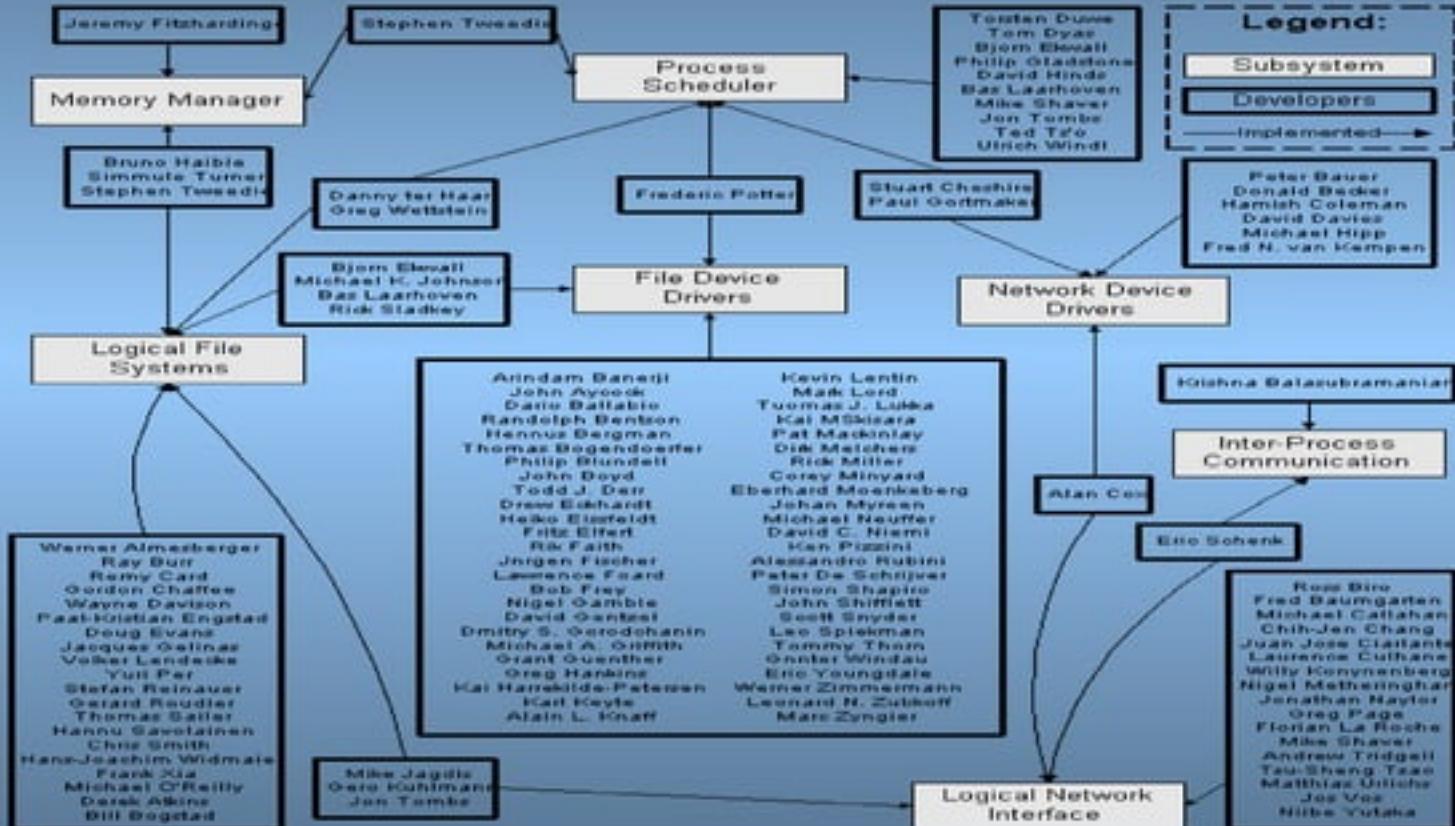
Linux Kernel

Hardware Controllers

Kernel Components

- Process Scheduler
- Memory Manager
- Virtual File-System
- Network Interface
- IPC

Open source community



COMMUNITIES

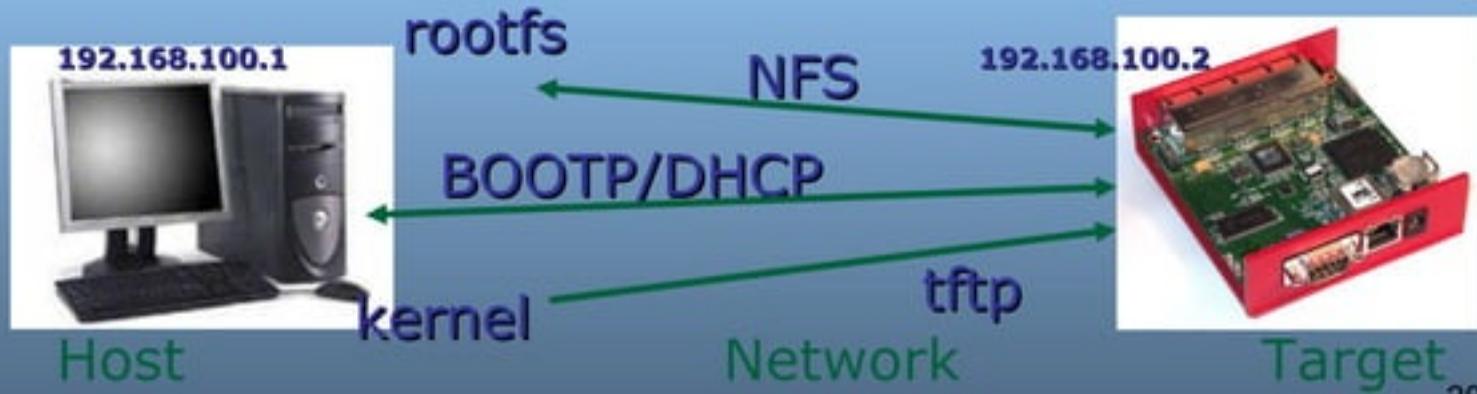
- Arm-linux
- U-boot
- Google linux-kernel
- Real-time
- Bugzilla – linux , gcc etc..

Cross-Development

- Cross development tools – gcc, gdb
- Kernel Source files – LSP
- File system – NFS mount
- Target executables – to be run on target

Cross-Development(cont)

- This course will assume that you are using cross development
- You will need to be able to boot your target
 - Typically, this will entail downloading a kernel from the boot host
 - Obtaining network parameters via BOOTP/DHCP
 - Mounting a root file system via NFS



Booting up Target

- » Configure the host
- » Power on the target board
- » Run minicom, and set the parameters
- » Download kernel image using tftp
- » Boot up the target

Booting up Target (Services)

- Configure the Host
 - dhcpcd – to assign the ip-address for the target
 - tftp – to download kernel image to the target
 - nfs – to nfs mount the file system for the target

Booting up Target(cont)

- /etc/dhcpd.conf

```
allow bootp; ddns-update-style ad-hoc;
subnet 192.168.1.0 netmask 255.255.255.0{
    default-lease-time 1209600 ;
    max-lease-time 31557600 ;
    option routers 192.168.1.1;
    group {
        host localhost.locaLdomain {
            hardware ethernet 00:0E:99:02:03:3F;
            option routers 192.168.1.1;
            fixed-address 192.168.1.122;
            filename "bp_nfs";
            option root-path "/opt/myfilesystem";}}}
```

Booting up Target(cont)

- /etc/exports
opt/myfilesystem *(rw,sync,no_root_squash,no_all_squash)
- /etc/xinetd.d/tftp

```
service tftp{  
    disable  = no  
    socket_type  = dgram  
    wait        = yes  
    user        = root  
    log_on_success += USERID  
    log_on_failure += USERID  
    server      = < should be tftpd server name >  
    server_args  = /tftpboot  
    protocol     = udp}
```

BUSYBOX

- **The Swiss Army Knife of Embedded Linux**
 - provide many UNIX utilities
 - cross-compilation
 - size-optimization
 - Not as extensive as GNU
 - Modular and configurable

LINUX CONFIGURATION

LINUX CONFIGURATION

- Building linux kernel
 - Linux source code
 - Compilation tools
 - configure
 - make menuconfig
 - compile
 - make all
 - install
 - make modules_install
 - make install

Config Methods

- console based: *make menuconfig*
- (GUI) Qt Based: *make xconfig*
- (GUI) GTK Based: *make gconfig*

OTHER

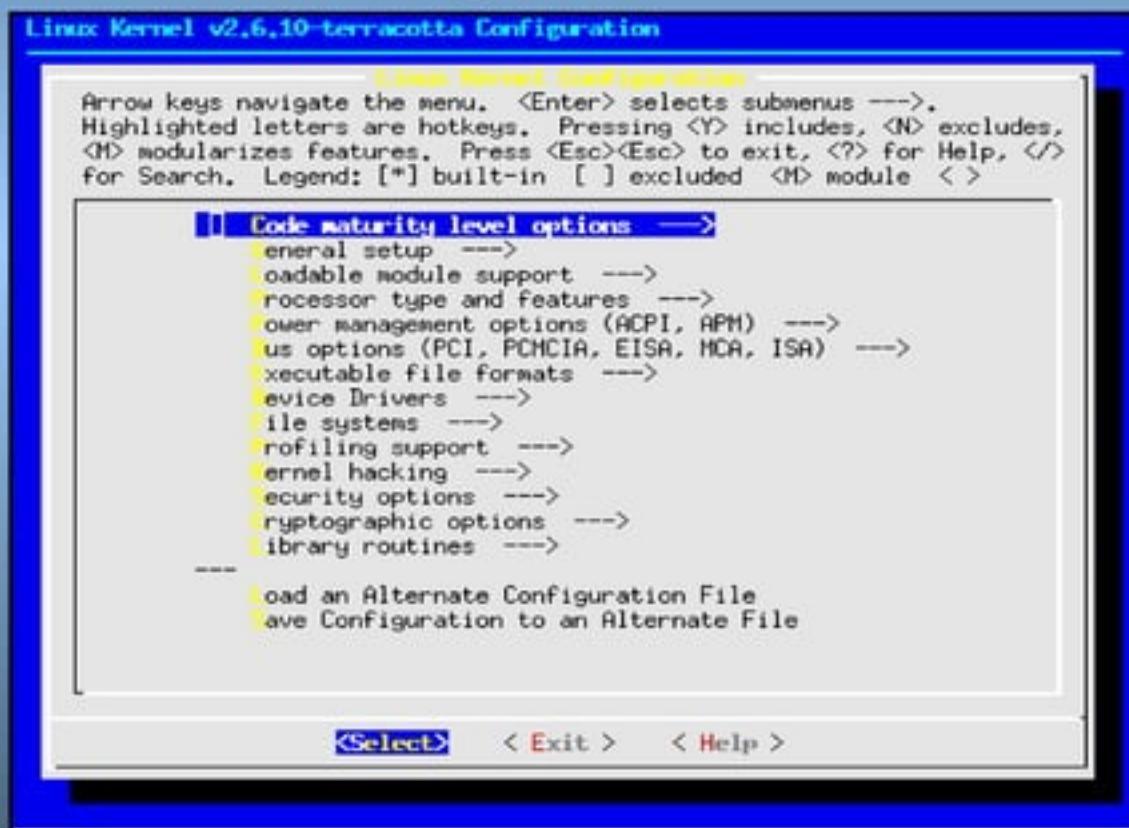
- keep old Kernels settings: *make oldconfig*
- edit the `/src/Linux/.config` file manually

CONFIG Initial steps

- Config files
 - `/arch/<xxx>/configs/ccc_defconfig`
- Main Active config file
 - `/src/linux/.config`
- Configure for respective config file
 - Make `ARCH=<xxx> ccc_defconfig`

Configuration(cont)

- Make menuconfig



Configuration(cont)

- Code maturity level options
- General setup
- Loadable module support
- Block layer
- Processor type and features
- Power management options (ACPI, APM)
- Bus options (**PCI**, **PCMCIA**, **EISA**, **MCA**, **ISA**)
- Executable file formats
- Networking *integrated kernel support for ppp, slip, vpn, ...)*
- Device Drivers (*drivers for sound, video, usb, disk drive, ...*)
- File systems *ext2, ext3, fat, ntfs, ...)*
- Howto configure the Linux kernel/Instrumentation Support?
- Howto configure the Linux kernel/kernel hacking?
- Security Options
- Cryptography options
- Library routines

Configuration(cont)

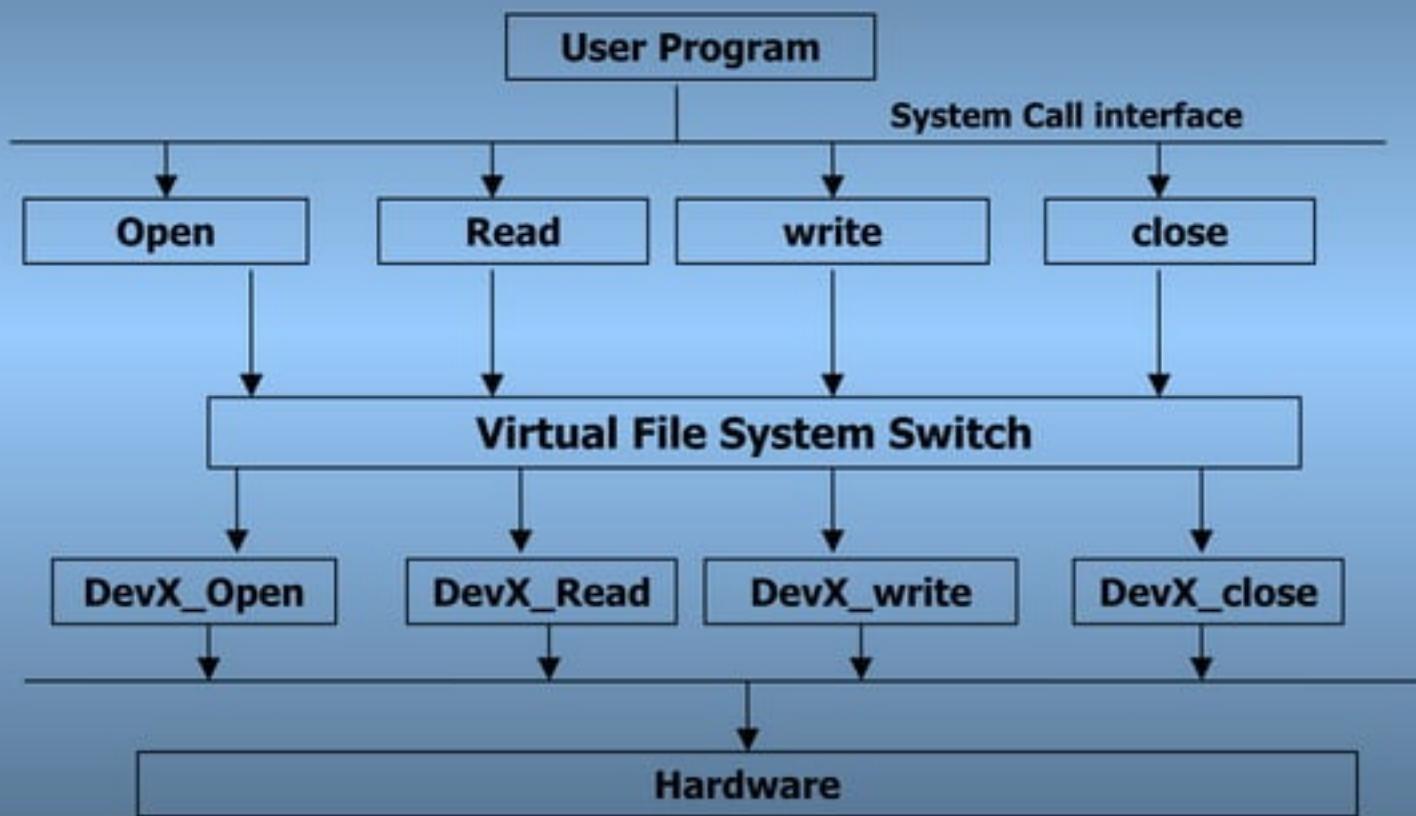
- Kconfig
 - Present in each directory for selecting options
 - Options viewed in menuconfig read from this file for each module
 - Any new modification/addition that you want to be configurable should edit corresponding Kconfig file
- Makefile

Introduction to Device Driver framework

DEVICE DRIVER

- In general Devices can be Physical/Virtual
- These devices are accessed by I/O Operations.
- Device concept in Linux is different to any other Operating system
- Virtually anything is considered as device file, for ex: /dev/hda1, /dev/hda2 etc..
- For each device there exists an entry point in the file system called inode
- The advantage of device file is obvious-we can use the same function and libraries of the file system to access and manipulate it. Ex: open(path, permissions);

Device Driver Interface



Memory/I/O mapped

- Every device is accessed through I/O operations.
- Control, Status or Data registers of the device is mapped to memory space or I/O space at boot time.
- Memory mapped I/O operations are performed like normal load, store operations, whereas I/O –I/O Mapped devices are accessed by IN/OUT or inw/outw instructions.
- I/O-I/O mapped devices are limited in space .. Used especially for x86 architecture
- Memory Mapped I/O. .. Another way of accessing devices

Kernel/User Space

- Kernel space which is flat address space being mapped from $0x100000$ to $PHY_OFFSET + 0x100000$ when Paging is turned ON during the process of Initialization.
- Kernel and user address space are accessed by virtual addressing concept.
- The user space and kernel can be configured based on user process size, 1:3 or 2:2.
- User process running in user space have limited privilege, and they access kernel space using system calls.

Kernel/User Space(cont)

- Kernel space cannot be swapped like user process.
- User space cannot access directly physically addresses.
- Kernel modules which uses logical addressing concepts, access physical memory by mapping physical memory to virtual address using the Macro and API's provided by the kernel.
- Example: ioremap, readb etc..

Kernel Space Activities

- Tasklets
- Kernel threads
- Soft IRQs
- Exceptions
- ISRs
- Device Drivers
- Kernel itself

User Space Activities

- Processes
- Shared libraries
- Daemons/Services
- Threads

Kernel modules

- Kernel Modules: Dynamically-loaded drivers are called as kernel modules.
- Linux driver can be loaded dynamically or statically.
- Drivers those loaded at boot time are called static drivers.
- Typically dynamic drivers are loaded by insmod/modprobe after kernel boot up.
- This Kernel modules are not linked completely unless they are loaded by the above commands.

Module Example

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Module Example (cont)

- Module_init gives the entry point of the module for rest of the driver.
- Module_exit de-registers the driver from kernel.
- insmod/modprobe looks for the Module_init within the object module to register driver.
- rmmod command looks for module_exit/clean_module that de-registers the module.

Compilation

- Makefile contains the following:

```
obj-m += hello.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
```

- In the command prompt, type 'make'
- <root>#make
- In Makefile 'uname -r' gives linux version number.(eg 2.6.20).
- This creates hello.ko, which can be loaded by insmod

To know more

- Documentation/kbuild
- Documentation/changes

Dynamic loading/Unloading

- Insmod
- Modprobe
- Modinfo
- depmod
- Rmmod
- Ismod

Inserting the Module

- insmod [objectfile], inserts module in to kernel and name of the file is registered in to modules
- Insertion problems are:
 - Version mismatch
 - Device busy
 - Symbols unresolved
 - Kernel tainted.

Insertion problem

- Version mismatch
 - This occurs due to compile version and kernel version mismatch.
 - Possible solution is to change the UT_RELEASE macro in the version.h file to the kernel version.

Insertion problem(cont)

- Device busy
 - This is basic and simple problem that has to be solved.
 - The insertion is not successful and thus exits.
 - Problem is due to the return value which happens to be not successful for kernel.

Insertion problem(cont)

- Symbols unresolved
 - Very Common problem
 - Symbol being used in this module depend on the other.
 - Dependent module do not exist in the kernel, hence it should be inserted first before this

Insertion problem (cont)

- Kernel tainted
 - Basic Licenses problem.
 - Kernel module is said to be tainted if the module loaded with out approved module.
 - Hence the release module should be valid one.
 - `MODULE_LICENSE()`, defines the license of the module

Module Dependency

- Program to generate modules dependency and map files, Linux kernel modules can provide services called symbol to use(using EXPORT_SYMBOL in the code), if a second module uses this symbol it is therefore dependent on first module that generates it.
- Depmod creates a list of module dependence, by reaching each module under /lib/module/version and determine what symbols it exports and what it depend on, thus generating the module.dep.
- Depmod, also generate version map file in this directory.

Module Info

- Modinfo a command in linux which examines object file `module_file` to read the respective attribute of the module, for example,
 - `-a` Author name
 - `-l` License
 - `-p` Parameters

Dynamic Module dependency loading:MODPROBE

- Intelligently loads the module
- /lib/modules/`uname-r`
- Modules.dep

Removing the module:Rmmod

- Unload the module under certain condition
 - No user of concerned module

Insmod Call

- Insmod
 - Sys_init_module
 - Vmalloc
 - Copy text
 - Resolve kernel reference
 - Module_init function

MODULE MACROS

- MODULE_LICENSE : tells kernel that module bears free license
- MODULE_AUTHOR
- MODULE_DESCRIPTION
- MODULE_VERSION
- MODULE_ALIAS
- MODULE_DEVICE_TABLE

Important Points

- Linked only to the kernel
- Concurrency
- Current process
- Exporting the symbols
 - EXPORT_SYMBOL

Module Parameters

- Insmod hello.ko xx=2 cc="ccc"

```
Static int xx=1;
```

```
Static char *cc="aa";
```

```
Module_param(xx , int , S_IRUGO);
```

```
Module_param(cc , charp , S_IRUGO);
```

- Module_param_array()

TYPES OF DEVICE DRIVER

- Character Driver
- Block Driver
- Network Driver

CHAR DRIVER

- Character driver are hardware components that read one character at a time in a sequence
 - They access one byte at a time
 - Tape devices, Serial devices and Parallel port devices are character devices
 - Character device driver do not use buffers
 - Basic functions need to access device are similar to any other devices.
 - Every character driver uses Major and Minor Number to access the device

BLOCK DRIVER

- Block devices are hardware devices in which data is read in Block, for example block size of 512byte etc..
 - Block device implements Buffering.
 - They have the same functions as character devices to access the device.
 - Block devices are of type Hard disk.
 - Even these devices posses major and Minor Numbers
 - They provide the foundation for all the file Systems.

Module working

- Register the device
 - Insmod (module_init function)
 - Registers Device file operations
 - Entry in /proc/<xxx>
- Building Device file node
 - Mknod
 - Major/minor number(dynamic / static)

NETWORK DRIVER

- Unlike character and block drivers that have a node in /dev, network drivers have a separate namespace and set of commands
 - The concept of “everything is a file” breaks down with network devices
- Like block drivers, network drivers move blocks of data
 - However, block drivers work on behalf of a user’s request whereas network drivers operate asynchronously to the user

Network Driver (cont)

- Network drivers also need to support:
 - Administrative tasks such as setting addresses, MTU sizes and transmission parameters
 - Collection of network traffic and error statistics

Driver-Hardware Communication

- Communicating with hardware
 - I/O ports
 - Request_region
 - Release_region
 - check_region
 - Inb,inw,outb,outw etc for read/write
 - I/O memory
 - Request_mem_region
 - Release_mem_region
 - Check_mem_region
 - ioremap
 - Readb , writeb etc

Character Driver

Registering Drivers

- It is a method of adding driver to the kernel.
- Devices in linux are identified by Major and Minor number.
- Registering a driver means assigning Major number to the device.
- It can be assigned by the user or dynamically by the kernel
 - Prototype
 - `Register_XXXdev(unsigned int MAJOR, char *name, struct file_operations *fops);`

Where MAJOR is 0 or any user assigned number, 0 requests the kernel to assign dynamically and return it.

Name is the name of the device used during mknod command.

Fops, device file operations like open,read, write etc.

In 2.6 Kernel argument file_operation pointer is optional

Registering Drivers (Cont..)

Internal representation of Device Numbers:

`Dev_t` type (`linux/types.h`) used to hold device numbers(major and minor parts).
Use the following macro to obtain major and minor parts of a `dev_t`.

```
MAJOR(dev_t dev);  
MINOR(dev_t dev);
```

Having major and minor numbers and need to turn them into a `dev_t` use below macro:

```
MKDEV(int major, int minor);
```

Allocating and freeing Device Numbers:

```
int register_chardev_region(dev_t first, unsigned int count, char *name);
```

The above prototype for to get one or more character device numbers.

`first` is the beginning device number of the range you would like to allocate (minor number portion is often 0).

`count` is total number of contiguous device numbers you are requesting.

`name` is name of device.

Registering Drivers (Cont..)

The below prototype allocate major number

```
int alloc_chardev_region(dev_t *dev, unsigned int firstminor, unsigned int count,  
                         char *name);
```

dev is an output-only parameter holds first
number.

firstminor is requested first minor number.

count and *name* are same as previous prototype

For unregister:

```
void unregister_chardev_region(dev_t first, unsigned int count);
```

```
int unregister_chardev (unsigned int major, const char *name);
```

Character Device Registration

The following is newer method of allocating and registering character device.

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;

void cdev_init( struct cdev *cdev, struct
    file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num,
    unsigned int count);
```

Private Data

The following prototype is for getting private data in file handler. This is performed in open method.

```
int (*open) (struct inode *inode, struct file *filp);
```

```
container_of(pointer, container_type, container_field);
```

Sample Code:

```
struct scull_dev *dev;  
dev = container_of(inode->i_cdev, struct scull_dev, cdev);  
filp->private_data = dev;
```

Major and Minor Number

- Major number is a method of identifying device
- It is of 12 bits in size
- Up to (4096-1) devices can address because of its size.
- If the same device shares the major number, minor number is defined to recognize it.
- Minor number again a 20 bits in size and hence it can address up to (1,048,576-1) devices.
- Most of the devices are associated with fixed number.

Major and Minor Number

- Major device assignment to the device is done either dynamically by kernel or by passing value during registration.
 - Registration process creates an entry for the device by inserting Name and File operations (optional) at the particular Major Number Index.
 - Table can be character or block devices table.
 - The name of the device will appear on the /proc/devices on successful registration.

File Operations

- Device driver interface routines, File operations is a structures which holds the function pointers for system call interface, basic operations are:
 - **Open**
 - **Release**
 - **Read**
 - **Write**
 - **ioctl**

User Space Access

```
unsigned long copy_to_user(void __user *to,  
    const void *from, unsigned long count);  
put_user(datum, ptr);  
__put_user(datum,ptr);
```

```
unsigned long copy_from_user(void *to, const void  
    __user *from, unsigned long count);  
get_user(local,ptr);  
__get_user(local,ptr);
```

Skeleton of Character Driver (Cont..)

- Below is skeleton structure for character driver.

```
• static struct file_operations fops = {  
•     .read = device_read,  
•     .write = device_write,  
•     .open = device_open,  
•     .release = device_release  
• };  
• int init_module(void)  
• {  
•     .....  
•     Major = register_chrdev(0, DEVICE_NAME, &fops);  
•     .....  
• }
```

Skeleton of Character Driver (Cont..)

```
• void cleanup_module(void)
• {
•     /* Unregister the driver and free the device(s) */
•     int ret = unregister_chrdev(Major, DEVICE_NAME);
• }

• static int device_open(struct inode *inode, struct file *file)
• {
•     /* Do the Character Device Initialization and assign private data to file handler */
• }
• }

• static int device_release(struct inode *inode, struct file *file)
• {
•     /* Unassign private data from file handler */
• }
```

Skeleton of Character Driver (Cont..)

```
• static ssize_t device_read(struct file *filp, /* see include/linux/fs.h */
•                           char *buffer,          /* buffer to fill with data */
•                           size_t length,         /* length of the buffer */
•                           loff_t * offset)
• {
•     .....
•     put_user(*(msg_Ptr++), buffer++);
•     .....
• }
•
• static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
• {
•     .....
•     copy_from_user(wrt_buf,buff,leng);
•     .....
• }
```

Skeleton of Character Driver (Cont..)

Do the following commands at command line.

```
<root>#make  
<root>#insmod <module-name>
```

Refer /var/log/messages or type 'dmesg' for to create device file node (mknod /dev/chardev c 253 0). 253 is dynamic major number.

Type the following:

```
<root># cat /dev/chardev  
<root># echo "C" > /dev/chardev
```

Skeleton of Character Driver with ioctl

Below is skeleton of character driver with ioctl and user level program for character driver. The user level program is for doing read and write through ioctl numbers:

- IOCTL_SET_MSG
- IOCTL_GET_MSG
- IOCTL_GET_NTH_BYTE

Skeleton of Character Driver with ioctl (Cont..)

```
• int device_ioctl(struct inode /* see include/linux/fs.h */  
•                 struct file /* ditto */  
•                 unsigned int ioctl_num,      /* number and param for ioctl */  
•                 unsigned long ioctl_param)  
• {  
•     switch (ioctl_num) {  
•         case IOCTL_SET_MSG:  
•             /*  
•              * Receive a pointer to a message (in user space) and set that  
•              * to be the device's message. Get the parameter given to  
•              * ioctl by the process.  
•             */  
•             get_user(ch, temp);  
•             device_write(file, (char *)ioctl_param, i, 0);  
•             break;  
•     }
```

Skeleton of Character Driver with ioctl (Cont..)

```
case IOCTL_GET_MSG:  
    /* Give the current message to the calling process -  
     * the parameter we got is a pointer, fill it. */  
  
    device_read(file, (char *)ioctl_param, 99, 0);  
    put_user('\0', (char *)ioctl_param + i);  
    break;  
  
case IOCTL_GET_NTH_BYTE:  
    /* This ioctl is both input (ioctl_param) and  
     * output (the return value of this function) */  
    break;  
}  
}
```

Skeleton of Character Driver with ioctl (Cont..)

```
• /*
•  * This structure will hold the functions to be called
•  * when a process does something to the device we
•  * created. Since a pointer to this structure is kept in
•  * the devices table, it can't be local to
•  * init_module. NULL is for unimplemented functions.
•  */
• struct file_operations Fops = {
•     .read = device_read,
•     .write = device_write,
•     .ioctl = device_ioctl,
•     .open = device_open,
•     .release = device_release,    /* a.k.a. close */
• };


```

User Program for ioctl (Cont..)

- The following skeleton structure for user space for char driver.
- ```
ioctl_set_msg(int file_desc, char *message)
{

 ioctl(file_desc, IOCTL_SET_MSG, message);

}
```
- ```
ioctl_get_msg(int file_desc)
{
    .....
    ioctl(file_desc, IOCTL_GET_MSG, message);
    .....
}
```

User Program for ioctl (Cont..)

```
•     ioctl_get_nth_byte(int file_desc)
•     {
•         .....
•         ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);
•         .....
•     }
•     main()
•     {
•         file_desc = open(DEVICE_FILE_NAME, 0);
•
•         ioctl_get_nth_byte(file_desc);
•         ioctl_get_msg(file_desc);
•         ioctl_set_msg(file_desc, msg);
•
•         close(file_desc);
•     }
```

Skeleton of Character Driver with ioctl (Cont..)

Type the following command.

```
#insmod <module-name>
```

```
#gcc -o <binary> <user_program>
```

Execute the binary

- I2c driver example

Kernel features used for driver development

- Time , delays , timeout , timers
- waitqueues
- Interrupt
- Allocating memory
- Communicating with hardware
- Data types in kernel
- Concurrency
- Mapping device memory
- DMA
- Debugging

LSP

- Stands for **Linux Support Package**
- Includes
 - Linux kernel
 - Compiler tool-chain (gnu)
 - File system.
- Specific to a board, so equivalent to BSP.
- Linux supports most of the COTS (Commercially Of The Shelf) boards or Reference boards.
 - Intel ixdp425, TI OMAP 1510 etc.
- Need to port Linux for custom board.

Choosing the kernel

- Latest, the greatest!!
 - Take the latest kernel from www.kernel.org
 - contain recent bug fixes and enhancements
- Check support for your
 - Architecture (like arm)
 - Processor/SoC (AT91RM)
 - Reference Board (AT91RM9200DK)
- Configure with the nearest “*defconfig*” file available.

Development Environment

- Cross development environment
- Host with Ethernet, serial
- Install the cross-compiler
 - Binaries available in the net
 - Compile from the source
 - Depends on the Linux kernel version.
- JTAG/ICE helpful for initial bring up/debug.

Where to start

- Follow the standard formula
 - Most systems are variants of existing ones
 - Copy the base platform files and modify
 - Add drivers for custom hardware devices

First Step-Machine type

- Register your machine type
 - Provides a unique numerical identifier for your machine
 - Provides a configuration variable for your machine
 - CONFIG_ARCH_\$MACHINE
eg: CONFIG_ARCH_STR9100
 - Provides runtime machine-check
 - machine_is_xxx()
eg: machine_is_str9100
Defined in “include/asm-arm/mach-types.h”
 - <http://www.arm.linux.org.uk/developer/machines/>
 - This information ends up in
 - arch/arm/tools/mach-types

include/asm-arm/mach-types.h

```
#define MACH_TYPE_EBSA110          0
#define MACH_TYPE_RISCPC           1
#define MACH_TYPE_NEXUSPCI         3
*
*
#define MACH_TYPE_INNOKOM          258
#define MACH_TYPE_BMS              259
#define MACH_TYPE_IXCDP1100         260
#define MACH_TYPE_PRPMC1100         261
#define MACH_TYPE_AT91RM9200DK      262
#define MACH_TYPE_ARMSTICK          263
#define MACH_TYPE_ARMONIE           264
#define MACH_TYPE_MPORT1            265
*
*
#define MACH_TYPE FUJITSU_WIMAXSOC   956
#define MACH_TYPE_DUALPCMODEM        957
#define MACH_TYPE_GESBC9312          958
```

Firmware Requirements

- ARM Linux requires certain tasks from firmware
 - Initialize all memory controller and system RAM
 - Initialize a serial port for early boot messages(Optional)
 - Disable MMU
 - Disable all caches (especially data cache)
 - Disable All interrupts (IRQs and FIQs)
 - Quiesce all DMA capable devices
 - Provide kernel parameter ATAG list
 - CPU Registers settings:
 - r0 = 0
 - r1 = machine type number
 - r2 = &(ATAG list)

ATAG Parameter List

- Data structure for providing machine details to kernel
 - ATAG_CORE
 - ATAG_MEM
 - Memory size and location
 - One per memory bank
 - ATAG_CMDLINE
 - Kernel command line string
 - ATAG_NONE
 - Signifies end of parameter list
- Usually located in first 16KiB of RAM
 - Most common is @ RAM base + 0x0100
 - Must not be overwritten by decompressor or initrd
- ATAG defined in include/asm-arm/setup.h

ATAG Parameter List

Tag name	Value	Size	Description
ATAG_NONE	0x00000000	2	Empty tag used to end list
ATAG_CORE	0x54410001	5 (2 if empty)	First tag used to start list
ATAG_MEM	0x54410002	4	Describes a physical area of memory
ATAG_VIDEOTEXT	0x54410003	5	Describes a VGA text display
ATAG_RAMDISK	0x54410004	5	Describes how the ramdisk will be used in kernel
ATAG_INITRD2	0x54420005	4	Describes where the compressed ramdisk image is placed in memory
ATAG_SERIAL	0x54410006	4	64 bit board serial number
ATAG_REVISION	0x54410007	3	32 bit board revision number
ATAG_VIDEOLFB	0x54410008	8	Initial values for vesafb-type framebuffers
ATAG_CMDLINE	0x54410009	2(length_of_cmdline + 3) / 4	Command line to pass to kernel

Directory/File Structure

- *arch/arm/*
 - *mm*
 - Cache/TLB/page fault/DMA handling
 - *kernel*
 - core kernel setup, APIs, and syscall handling
 - *lib*
 - low-level helper functions (mostly ASM)
 - *common*
 - code shared across various machine types
 - *mach-\$MACHINE*
 - Machine-type specific code (mach-at91rm9200, -ixp4xx, etc)
 - *configs/\$PLATFORM_defconfig*
 - Default configuration for \$PLATFORM (at91rm9200dk, ixp4xx, etc)
- *include/asm-arm/arch-\$MACHINE (include/asm/arch)*
 - Machine-specific headers (arch-at91rm9200, ixp4xx, etc)

Early init

- Early serial during decompression
 - arch_decomp_setup()
 - putstr()
 - include/asm-arm/arch \$MACHINE/uncompress.h
- Debug Serial Output
 - addruart, rx
 - Provide UART address in \rx
 - senduart rd, rx
 - Send character in \rd (@ address \rx)
 - busyuart rd, rx
 - Wait until UART is done sending
 - waituart rd, rx
 - Wait for Clear to Send
 - Found in arch/arm/kernel/debug.S

CPU Detection

- Large number of ARM CPU variants in production
 - Each one has different methods of managing cache, TLB
 - Sometimes need to build kernel to boot on various CPUs
- Kernel contains table of CPUs it supports
 - arch/arm/mm/proc-\$CPUTYPE
 - include/asm-arm/procinfo.h
 - First thing kernel does is check CPUID with table of CPUs
 - Table contains a mask and expected value
 - If (`cpuid & cpu_mask`) == `cpu_val` we are OK
 - Otherwise we can't run on this CPU
 - If OK, call CPU's `_setup` function

Low Level CPU APIs

- Processor-specific functions
 - Data abort, CPU init, reset, shutdown, idle
 - include/asm-arm/cpu-multi32.h
- TLB handling
 - Flush user and kernel TLB entries
 - include/asm-arm/tlbflush.h
- Cache functions
 - Flush and clean kernel and user range from cache
 - Sync icache and dcache for new text
 - Sync dcache with memory for DMA operations
 - include/asm-arm/cacheflush.h
- User data functions
 - Clear and copy user page
 - include/asm-arm/page.h

Architecture Specific Macros

- **BOOT_MEM(pram,pio,vio)**
 - 'pram'
 - Physical start address of RAM.
 - Must always be present, and should be the same as PHYS_OFFSET.
 - 'pio'
 - Physical address of an 8MB region containing IO for use with the debugging macros in arch/arm/kernel/debug-armv.S.
 - 'vio'
 - the virtual address of the 8MB debugging region.
- **BOOT_PARAMS**
 - Physical address of the struct param_struct or tag list
 - Gives kernel various parameters about its execution environment.
- **FIXUP(func)**
 - Machine specific fixups, run before memory subsystems have been initialized.
- **MAPIO(func)**
 - Machine specific function to map IO areas (including the debug region above).
- **INITIRQ(func)**
 - Machine specific function to initialize interrupts

Machine Detection

- If CPU is detected OK, check machine type
- Each platform has a machine descriptor structure:

```
MACHINE_START(AT91RM9200DK, "Atmel AT91RM9200-DK")
    MAINTAINER("SAN People / ATMEL")
    BOOT_MEM(AT91_SDRAM_BASE, AT91C_BASE_SYS,
              AT91C_VA_BASE_SYS)
    BOOT_PARAMS(AT91_SDRAM_BASE + 0x100)
    .timer      = &at91rm9200_timer,
    MPIO(dk_map_io)
    INITIRQ(dk_init_irq)
    INIT_MACHINE(dk_board_init)
MACHINE_END
```

MACHINE_START

- MACHINE_START macro expands to a statically built data structure
- i.e. one that is there when the compiler builds the file (Linux Kernel image).
- It expands to:

```
const struct machine_desc __mach_desc_(type)
__attribute__((__section__(".arch.info"))) = {
    nr:      MACH_TYPE_(type),
    name:   (name)
```

- (type) is the first argument passed to the MACHINE_START macro, and (name) is the second.
- The various other definitions set various machine_desc structure members using named initializers
- Finally MACHINE_END provides the closing brace for the structure initializer. (see include/asm-arm/mach/arch.h)

machine_desc

```
struct machine_desc {
/*
 unsigned int nr; /* architecture number */
 unsigned int phys_ram; /* start of physical ram */
 unsigned int phys_io; /* start of physical io */
 unsigned int io_pg_offset; /* byte offset for io
 * page table entry */
 const char *name; /* architecture name */
 unsigned int param_offset; /* parameter page */
 unsigned int video_start; /* start of video RAM */
 unsigned int video_end; /* end of video RAM */
 unsigned int reserve_lp0 :1; /* never has lp0 */
 unsigned int reserve_lp1 :1; /* never has lp1 */
 unsigned int reserve_lp2 :1; /* never has lp2 */
 unsigned int soft_reboot :1; /* soft reboot */
 (*fixup)(struct machine_desc *,
 struct tag *, char **,
 struct meminfo *);
 void (*map_io)(void); /* IO mapping function */
 void (*init_irq)(void);
 struct sys_timer *timer; /* system tick timer */
 void (*init_machine)(void);
};
```

Machine Descriptor Macros

```
#define BOOT_MEM(_pram,_pio,_vio) \
    .phys_ram      = _pram, \
    .phys_io       = _pio, \
    .io_pg_offset  = ((_vio)>>18)&0xffffc, \
    \
#define BOOT_PARAMS(_params) \
    .param_offset  = _params, \
    \
#define VIDEO(_start,_end) \
    .video_start   = _start, \
    .video_end     = _end, \
    \
#define DISABLE_PARPORT(_n) \
    .reserve_ip##_n = 1, \
    \
#define SOFT_REBOOT \
    .soft_reboot   = 1, \
    \
#define FIXUP(_func) \
    .fixup         = _func, \
    \
#define MAPIO(_func) \
    .map_io        = _func, \
    \
#define INITIRQ(_func) \
    .init_irq      = _func, \
    .init_machine  = _func,
```

“.arch.info”

- machine_desc structures are collected up by the linker into the .arch.info ELF section on final link, which is bounded by two symbols,
 - __arch_info_begin
 - __arch_info_end.
- These symbols do not contain pointers into the .arch.info section, but their address are within that section.
- System.map

c001ddf4 T __arch_info_begin

c001ddf4 T __mach_desc_AT91RM9200DK

c001ddf4 T __proc_info_end

Static I/O Mapping

- Certain devices needed before VM is fully up and running
 - Interrupt controllers, timer tick
- Certain devices require large VM areas
 - Static mapping allows usage of 1MB sections
 - ioremap() uses only 4K pages
 - Larger TLB footprint
- Call mdesc->map_io()
- Call create_mapping() with static I/O mappings

Kernel Memory Layout on ARM Linux

- The ARM CPU is capable of addressing a maximum of 4GB virtual memory space
- Shared between
 - user space processes
 - Kernel
 - Hardware devices
- Kernel memory is in upper 1GB
- Static mapping fit in `VMALLOC_END-0xFFFFFFF`
- `VMALLOC_END` is defined by you include/asm-arm/arch-\$MACHINE/vmalloc.h

Decompressor Symbols

- **ZTEXTADDR**
 - Start address of decompressor.
 - MMU will be off at the time when you call the decompressor code.
 - Doesn't have to be located in RAM
 - Can be in flash or other read-only or read-write addressable medium.
- **ZBSSADDR**
 - Start address of zero-initialised work area for the decompressor.
 - Must be pointing at RAM.
 - MMU will be off.
- **ZRELADDR**
 - Address where the decompressed kernel will be written and executed.
 - `__virt_to_phys(TEXTADDR) == ZRELADDR`
- **INITRD_PHYS**
 - Physical address to place the initial RAM disk.
- **INITRD_VIRT**
 - Virtual address of the initial RAM disk.
 - `__virt_to_phys(INITRD_VIRT) == INITRD_PHYS`
- **PARAMS_PHYS**
 - Physical address of the struct param_struct or tag list
 - Gives the kernel various parameters about its execution environment.

Kernel Symbols

- **PHYS_OFFSET**
 - Physical start address of the first bank of RAM.
- **PAGE_OFFSET**
 - Virtual start address of the first bank of RAM.
 - During the kernel boot phase, virtual address PAGE_OFFSET will be mapped to physical address PHYS_OFFSET, along with any other mappings you supply.
 - Should be the same value as TASK_SIZE.
- **TASK_SIZE**
 - The maximum size of a user process in bytes.
 - Since user space always starts at zero, this is the maximum address that a user process can access+1.
 - The user space stack grows down from this address.
 - Any virtual address below TASK_SIZE is deemed to be user process area
 - Managed dynamically on a process by process basis by the kernel.
 - User segment
 - Anything above TASK_SIZE is common to all processes
 - the kernel segment

Kernel Symbols(Contd.)

- TEXTADDR
 - Virtual start address of kernel, normally PAGE_OFFSET + 0x8000. This is where the kernel image ends up.
 - With the latest kernels, it must be located at 32768 bytes into a 128MB region. Previous kernels placed a restriction of 256MB here.
- DATAADDR
 - Virtual address for the kernel data segment.
 - Must not be defined when using the decompressor.
- VMALLOC_START
- VMALLOC_END
 - Virtual addresses bounding the vmalloc() area.
 - There must not be any static mappings in this area; vmalloc will overwrite them.
 - The addresses must also be in the kernel segment.
- VMALLOC_OFFSET
 - Offset normally incorporated into VMALLOC_START to provide a hole between virtual RAM and the vmalloc area.
 - Normally set to 8MB.

Kernel Memory Layout on ARM Linux-1

Reserved for vectors, DMA buffers, etc	0xffffffff
Static I/O	0xfeffff
vmalloc() and ioremap()	VMALLOC_END
Direct Mapped RAM	VMALLOC_START
Kernel Modules	PAGE_OFFSET (0xc0000000)
User Mappings	TASK_SIZE
	0x00000fff
NULL/Vector Trap	0x00000000

Kernel Memory Layout on ARM Linux-2

copy_user_page / clear_user_page use.	FFFFFFFFFF
Reserved. Platforms must not use this address range.	FFFFF7FFF
CPU vector page. The CPU vectors are mapped here if the CPU supports vector relocation (control register V bit.)	FFFF0FFF
DMA memory mapping region. Memory returned by the dma_alloc_xxx functions will be dynamically mapped here	FFFE FFFF
Reserved for future expansion of DMA mapping region.	FFBF FFFF
Free for platform use, recommended.	FEFF FFFF
vmalloc() / ioremap() space. Memory returned by vmalloc/ioremap will be dynamically placed in this region. VMALLOC_START may be based upon the value of the high_memory variable.	VMALLOC_END-1
Kernel direct-mapped RAM region. This maps the platforms RAM, and typically maps all platform RAM in a 1:1 relationship.	VMALLOC_START
0000	high_memory-1
Kernel module space	PAGE_OFFSET-1
Kernel modules inserted via insmod are placed here using dynamic mappings.	
User space mappings	TASK_SIZE-1
Per-thread mappings are placed here via the mmap() system call.	
CPU vector page / null pointer trap CPUs which do not support vector remapping place their vector page here. NULL pointer dereferences by both the kernel and user space are also caught via this mapping.	0000 0FFF
	115
	0000 0000

IRQ

- NR_IRQS defined in *include/asm-arm/arch/irqs.h*
 - Prepares the *iqr* descriptor table depending on NR_IRQS
 - *arch/arm/kernel/irq.c*
 - *struct irqdesc irq_desc[NR_IRQS]*
- IRQ numbering is up to developer
- First level IRQ decoding done in ASM
 - *include/asm/arch-\$MACHINE/entry-macro.S*
 - *get_irqnr_and_base* irqnr, irqstat, base, tmp
 - Linux IRQ number returned in \irqnr
 - Others are for temp calculations

IRQ(Contd.)

- IRQ “Chips”
 - Chip defines a mask, unmask, retrigger, set-type and acknowledge functions.

```
static struct irqchip gpio_irqchip = {  
    .mask      = gpio_irq_mask,  
    .unmask    = gpio_irq_unmask,  
    .set_type  = gpio_irq_type,  
};
```

irqaction

```
struct irqaction {  
    irqreturn_t (*handler)(int, void *, struct pt_regs *);  
    unsigned long flags;  
    cpumask_t mask;  
    const char *name;  
    void *dev_id;  
    struct irqaction *next;  
    int irq;  
    struct proc_dir_entry *dir;  
};
```

System Timer Tick

- Initialized after IRQs
 - Load timer with LATCH
 - $((CLOCK_TICK_RATE + HZ/2) / HZ)$
 - `CLOCK_TICK_RATE` is HW clock freq.
 - Request timer interrupt
 - Set `SA_INTERRUPT`

```
static struct irqaction at91rm9200_timer_irq = {  
    .name          = "AT91RM9200 Timer Tick",  
    .flags         = SA_SHIRQ | SA_INTERRUPT,  
    .handler      = at91rm9200_timer_interrupt  
};
```

- Timer ticks every (1/HZ)s

System Timer Tick(Contd.)

- The interrupt handler “at91rm9200_timer_interrupt” is called by function

```
setup_irq(AT91C_ID_SYS, &at91rm9200_timer_irq);  
in
```

```
void __init at91rm9200_timer_init(void)
```

- System timer function for the processor

```
struct sys_timer at91rm9200_timer = {  
    .init          = at91rm9200_timer_init,  
    .offset        = at91rm9200_gettimeoffset,  
};
```

System Timer Tick(Contd.)

- Timer interrupt:
 - Call `timer_tick()`
 - Reload timer source if needed
- `gettimeoffset()` function
 - Returns number of *usec* since last timer tick

Board Level Device Initialization

- After core subsystems are initialized,
`board_init()` is called
 - Do any last needed fixups for the platform
 - Add platform devices (flash, I2C, etc)
 - Very board/platform specific

Platform bus

- Pseudo-bus used to connect devices on busses with minimal infrastructure
- Like those used to integrate peripherals on many system-on-chip processors, or some "legacy" PC interconnects
- Look in
 - include/platform_device.h

Platform devices

- Devices that typically appear as autonomous entities in the system.
- Includes
 - Legacy port-based devices
 - Host bridges to peripheral buses
 - Most controllers integrated into SoC platforms
- Have in common
 - Direct addressing from a CPU bus
 - Rarely, will be connected through a segment of some other kind of bus
 - But its registers will still be directly addressable.
- Are
 - Given a name
 - Used in driver binding
 - A list of resources
 - such as addresses and IRQs.

Platform Device (Contd.)

```
struct platform_device {  
    const char      *name;  
    u32              id;  
    struct device    dev;  
    u32              num_resources;  
    struct resource  *resource;  
};
```

Eg:

```
static struct platform_device at91rm9200_eth_device = {  
    .name      = "at91_ether",  
    .id        = -1,  
    .dev       = {  
        .dma_mask          = &eth_dmamask,  
        .coherent_dma_mask = 0xffffffff,  
        .platform_data     = &eth_data,  
    },  
    .num_resources = 0,  
};  
  
platform_device_register(&at91rm9200_eth_device);
```

Platform drivers

- Follow the standard driver model convention
- discovery/enumeration is handled outside the drivers
- provide probe() and remove() methods.
- They support power management and shutdown notifications using the standard conventions.

```
struct platform_driver {  
    int (*probe)(struct platform_device *);  
    int (*remove)(struct platform_device *);  
    void (*shutdown)(struct platform_device *);  
    int (*suspend)(struct platform_device *, pm_message_t state);  
    int (*suspend_late)(struct platform_device *, pm_message_t state);  
    int (*resume_early)(struct platform_device *);  
    int (*resume)(struct platform_device *);  
    struct device_driver driver;  
};
```

Platform data

- Platform data specific to the device.
- Uses `platform_data` to point to board-specific structures describing devices and how they are wired
- Include
 - what ports are available
 - Chip variants
 - Which GPIO pins act in what additional roles
- Shrinks the "Board Support Packages" (BSPs)
- Minimizes board-specific #ifdefs in drivers

```
static struct at91_eth_data __initdata dk_eth_data = {  
    .phy_irq_pin  = AT91_PIN_PC4,  
    .is_rmii     = 1,  
};
```

Linux Flow

- Uncompressed kernel
 - arch/arm/boot/compressed/head.S
 - start:
 - arch/arm/boot/compressed/head-<target>.S
 - arch/arm/boot/compressed/misc.c
 - Decompressor
 - r1 - architecture
 - r2 – atags pointer
 - r4 - zreladdr

Linux Flow(Contd.)

- ARM-specific kernel code
 - arch/arm/kernel/head.S: stext
 - lookup_processor_type
 - list in proc_info_list structs
 - defined in arch/arm/mm/proc-arm926.S and corresponding proc-*.S files
 - linked into list by section declaration: .section ".proc.info.init"
 - return pointer to proc_info_list struct corresponding to processor if found, or loop in error if not
- call __lookup_machine_type
 - defined in arch/arm/kernel/head.S
 - search list of supported machines from machine_desc structs
- call __create_page_tables to set up initial page tables for Linux kernel

Linux Flow (Contd.)

- set the following used for jumps after the following calls
 - r13 to `switch_data`
 - address of the first function to be ran after MMU is enabled
 - lr to `enable_mmu` function address
- enable MMU - virtual addresses
- jump using address of `__switch_data`, whose first field is address of `__mmap_switched`
 - copy data segment to RAM
 - zero BSS
 - branch to `start_kernel`

Linux Flow (Contd.)

- Architecture independent code
 - Init/main.c – start_kernel
 - Process 0
 - Initializes all other stuffs pages, file system, driver, spans the ‘init’.

Kernel command line Parameters

- The kernel can accept information at boot time
- Used to supply the kernel with information
 - About hardware parameters that the kernel would not be able to determine on its own
 - To avoid/override the values that the kernel would otherwise detect
- Can be given by
 - During kernel compilation
 - Boot-loader environment variable
 - With 'boot' command
- How the Kernel Sorts the Arguments
 - Most of the boot args take the form of:
 - name[=value_1][,value_2]...[,value_11]
 - 'name' is a unique keyword
 - Multiple boot arguments separated by space
 - Limited to 11 parameters per keyword

General Non-Device Specific Boot Args

- Root Filesystem options
 - The 'root=' Argument
 - Tells the kernel what device is to be used as the root file system while booting.
- Valid root devices are any of the following devices:
 - /dev/hdaN to /dev/hddN
 - Partition N on ST-506 compatible disk 'a to d'.
 - /dev/sdaN to /dev/sdeN
 - Partition N on SCSI compatible disk 'a to e'.
 - /dev/xdaN to /dev/xdbN
 - Partition N on XT compatible disk 'a to b'.
 - /dev/fdN
 - Floppy disk drive number N. Having N=0 would be the DOS 'A:' drive, and N=1 would be 'B:'.
 - /dev/nfs
 - Which is not really a device, but rather a flag to tell the kernel to get the root fs via the network.
 - /dev/ram
 - The RAM disk.
- Disk devices in major/minor format is also accepted.
 - /dev/sda3 is major 8, minor 3, so you could use root=0x803

General Boot Args(Contd.)

- The 'rootflags=' Argument
 - Pertaining to the mounting of the root filesystem.
 - giving the noatime option to an ext2 fs.
- The 'rootfstype=' Argument
 - A comma separated list of fs types that will be tried for a match when trying to mount the root filesystem.
- The 'ro' Argument
 - Tells the kernel to mount the root file system as 'readonly'
- The 'rw' Argument
 - Tells the kernel to mount the root filesystem as read/write.
 - The default is to mount the root filesystem as read only.
- The 'nfsroot=' Argument
 - Tells the kernel which machine, what directory and what NFS options to use for the root filesystem.
- The 'ip=' or 'nfsaddrs=' Argument
 - This boot argument sets up the various network interface addresses that are required to communicate over the network.
 - If this argument is not given, then the kernel tries to use RARP and/or BOOTP to figure out these parameters.

RAM Disk Management

- **initrd**
 - Specify the location of the initial ramdisk
 - Size of the initrd.
 - `initrd=0x20410000,3145728`
- **noinitrd**
 - Tells the kernel not to load any configured initial RAM disk.
- **ramdisk_size**
 - Sizes of RAM disks in kilobytes
- **rdinit**
 - Format: `<full_path>`
 - Run specified binary instead of `/init` from the ramdisk

Related to Memory Handling

- Arguments alter how Linux detects or handles the physical and virtual memory of the system
- `mem='
 - specify the amount of installed memory
 - mem=64M@0xC0000000 mem=1023M@1M
- `swap='
 - Allows the user to tune some of the virtual memory (VM) parameters that are related to swapping to disk.
 - MAX_PAGE_AGE PAGE_ADVANCE
 - PAGE_DECLINE PAGE_INITIAL_AGE
 - PAGE_CLUSTER_FRACT PAGE_CLUSTER_MI
 - PAGEOUT_WEIGHT BUFFEROUT_WEIGHT
- `buff='
 - Allows the user to tune some of the parameters related to buffer memory management.
 - MAX_BUFF_AGE BUFF_ADVANCE
 - BUFF_DECLINE BUFF_INITIAL_AGE
 - BUFFEROUT_WEIGHT BUFFERMEM_GRACE

Misc. Kernel Boot Arguments

- `console='` Argument
 - Specify the console for the Linux
 - Ex. `console=ttyS1,9600`
- `'init='` Argument
 - Specify the 'init' to be executed
 - Ex. `init=/bin/sh`
- PCI, sound, video, SCSI are available

Debugging Linux Kernel With KGDB

- KGDB- Kernel Gnu Debugger
- A remote host Linux kernel debugger through gdb
- Provides a mechanism to debug the Linux kernel using gdb on the host.
- <http://sourceforge.net/projects/kgdb>

Procedure for KGDB

- Need KGDB support in kernel or KGDB patch.
- Select the 'Remote GDB kernel debugging' listed under "Kernel hacking".
 - make menuconfig
 - Enable KGDB
 - Console on KDGB (if second serial port not available for console)
 - Additional options
 - Source debug "-g"
 - May need to add gdb options to kernel command line
 - gdbbaud
 - gdbtty
- Recompile the kernel
 - make clean zImage/uImage/all

Procedure for KGDB(Contd.)

- Connect host-target with serial Null-modem cable
- Boot with the new image
- Booting will stop after “Un-compressing kernel” for a gdb connection.
- Quit minicom/hyperterminal
- Start the cross gdb in the host
- #arm-linux-gdb
 - gdb>set
 - gdb>set remotebaud <baudrate>
 - gdb>file vmlinux
 - gdb>target remote <serial port on host connected target>

Advantages of KGDB

- Source level debugging
- Can debug a running kernel through serial/Ethernet
- Can debug static drivers and dynamic modules
- Single-step/break-points are supported
- Target system doesn't require a VT-capable processor.

Disadvantages of KGDB

- KGDB needs two systems
 - one running the Linux kernel
 - another controlling it.
- Need a dedicated serial port
- KGDB also needs a KGDB enabled/ patched Linux kernel.
- If the KGDB patch for the desired kernel is not available, then there will be more effort to port the KGDB patch to the desired Linux kernel

Debugging With JTAG

- Using Abatron BDI2000
- Compile kernel with ‘-g’ option
- Get the required address from System.map
- Put the breakpoint
- Use gdb/ddd for source level debugging

Creating Downloadable Image

- 'make' is available with Linux kernel
 - make zImage
 - make ulimage
 - make vmlinux
 - make all

Different types of Linux kernel Images

- **zImage** .. Combination of several pieces
 - vmlinuz + Linux Boot Loader + Optional ramfs
- **uImage**
 - vmlinuz + VBL + with headers an uboot loader can understand + Optional ramfs
- **vmlinux**
 - Uncompressed image which has debug information (with -g option)

Kernel

User Processes

- User applications each have their own memory address space
- Linux maintains a `task_struct` for each piece of user code
 - Contains a pointer to the virtual memory map for the process
- The `task_struct` coupled with the memory map are typically considered a “process”



User Threads

- In many UNIX-like operating systems, the user process context is fairly large
- To improve performance, many of these O/S implementations introduced a “light-weight” process frequently referred to as a thread
- Threads usually share the memory map of a parent process

Threads Library

- NPTL
 - Native posix thread library
 - Pthreads
 - Lpthread
 - Pthreads API

Linux “Tasks”

- Linux doesn't really distinguish between processes and threads
 - Both use the same `task_struct`
 - Both are scheduled globally in the CPU
 - Essentially, everything's a task
- In Linux, a process is really just a “main” task that serves as the anchor for a virtual memory address (VMA) space

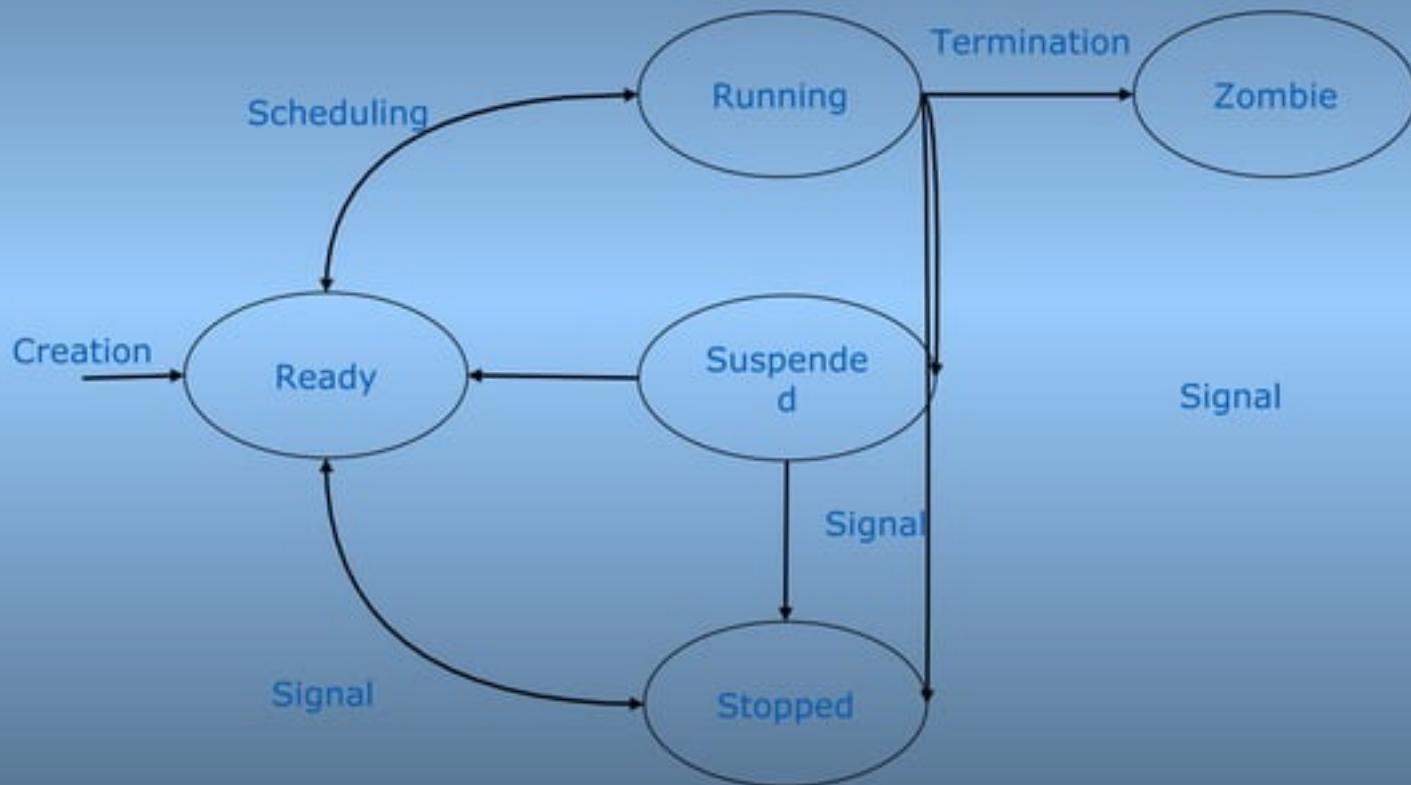
Kernel Daemons

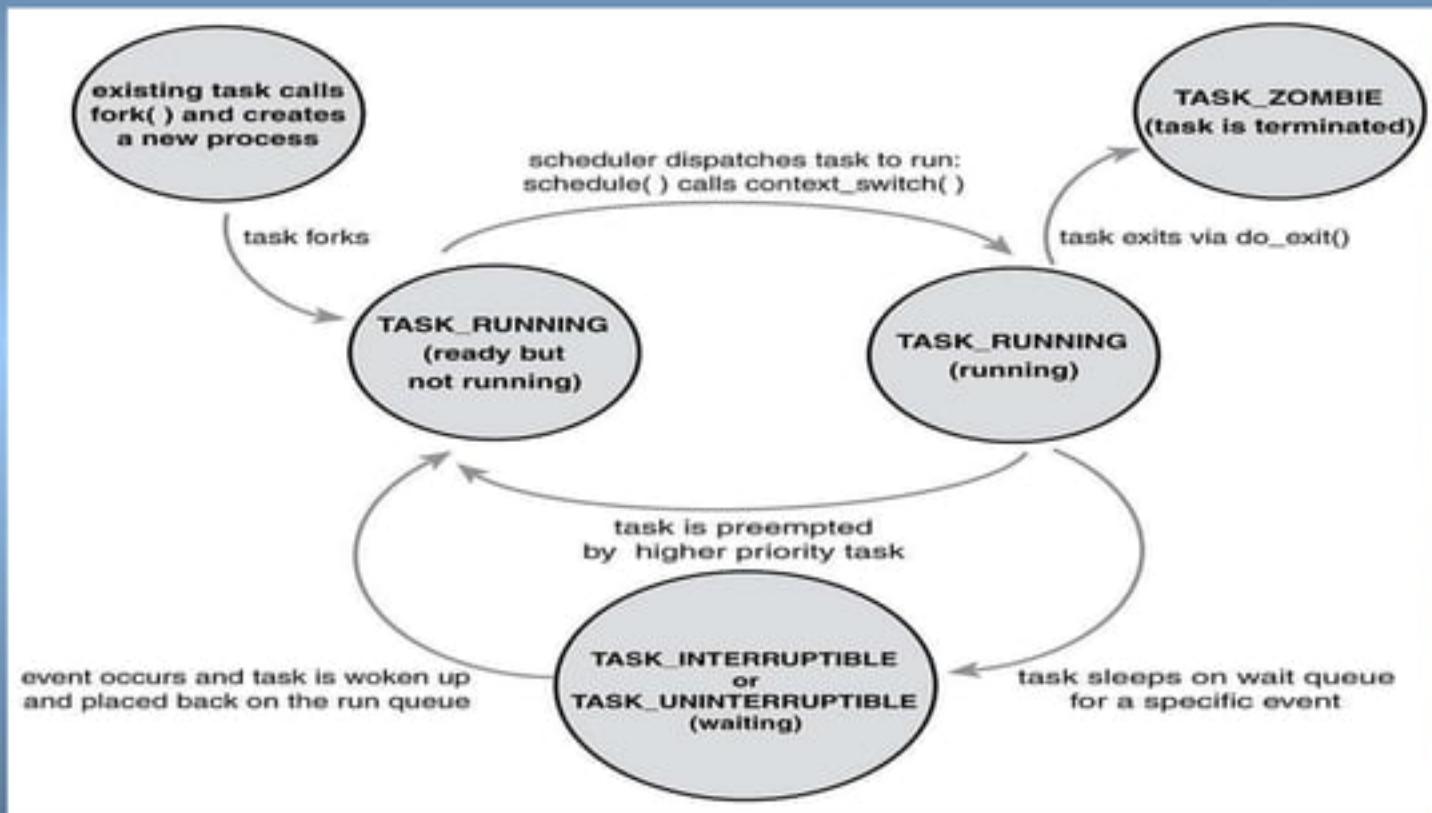
- Kernel thread to be created by other kernel threads
- Threads have own PID
- Threads share resources
- Kernel threads operate in kernel address space
- `Int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`

Kernel threads

- Process 0 – idle process
- Process 1 – Init
- Keventd
- Kapmd
- Ksawpd
- Pdflush
- Kblockd
- ksoftirqd

Process States





Scheduler

- Time sharing (time-slice/quantum)
 - Depends on timer interrupt
- Priority
- 2 main kinds of processes
 - Interactive
 - Real-time
- Scheduler based on interactivity estimator
 - CPU bound or I/O bound
- Runqueues per CPU

Process priority

- Static priority
 - -20 to +19
- Dynamic priority
 - Scheduler interactivity estimator
- Real-time priority
 - 1-99 Real-time priority
- 140 priority levels

SCHEDULING POLICY

- LinuxThreads/NPTL/Processes supports 3 POSIX scheduling policies simultaneously
 - SCHED_FIFO
 - Preemptive-priority based FIFO
 - SCHED_RR
 - Preemptive-priority with Round-robin
 - SCHED_OTHER
 - Conventional Round-robin
- SCHED_FIFO/SCHED_RR allow you to assign a real-time priority
- SCHED_OTHER is locked at priority 0

Preemptive-Priority Based

- Real-time priorities from 1-99
 - 1 is lowest, 99 is highest
 - Default range can be changed when the kernel is built
- Highest priority thread gets the CPU and holds it
 - No attempt to be fair
- Can be preempted by:
 - ISR
 - Kernel
 - Higher priority thread

Process Queues

- Runqueue
 - For task in running state
- Wait queue
 - Task in sleeping state
 - Define wait queue
 - Add task in wait queue for some condition to be true
 - Task moves in waitqueue
 - Wake-up task and make condition true from outside
 - Task is moved from waitqueue to runqueue

Process Creation

- FORK
- VFORK
- Kernel_thread
- pthread

System Calls related to Scheduling

- nice
- getpriority/setpriority
- sched_getaffinity / sched_setaffinity
- sched_getscheduler / sched_setscheduler
- sched_yield
- sched_get_priority_min /
 sched_get_priority_max
- sched_rr_get_interval

System Calls

- System calls are explicit requests to the kernel made via software interrupts (int \$0x80).
- When a user mode process invokes a system call the CPU switches from user mode to kernel mode, and starts executing functions.
- Each system call is assigned a ‘syscall’ number
- List of registered system calls are in ‘sys_call_table’ in ‘entry.S’ file
- Parameters are stored in Registers and an exception is generated

System call Implementation

- System call API
 - Malloc etc
 - Implemented in libc
 - Internally uses system calls like brk

SYSTEM CALL (Cont)

- Let's write a system call 'foo'
- `asmlinkage long sys_foo(void)`
- `printf("\n I am in system call foo \");`
- `return -ENOMEM`
- `}`
- Asmlinkage ??

SYSTEM CALL (Cont)

- You need to add this to the list of system call table
 - Open the file 'entry.S' or 'misc.S' in arch/ppc/kernel directory
 - Add the line '.long sys_foo' to the end of table
 - ENTRY(sys_call_table)
- .long sys_restart_syscall /* 9 */
- ..
- ..
- .long sys_foo /* May be sys call # is 268 */
- Open the file include/asm/unistd.h and the macro __NR_foo 268 /* What ever # assigned from the previous table */ for example
- #define __NR_restart_syscall 0
- #define __NR_exit 1
- ..
- ..
- #define __NR_foo 268
- Make sure you compile the code 'sys_foo.c' and rebuild the kernel
- Now Applications can call the system call foo()

- Pass Parameters
 - Mmap , write etc
- Verifying parameters

Memory manager

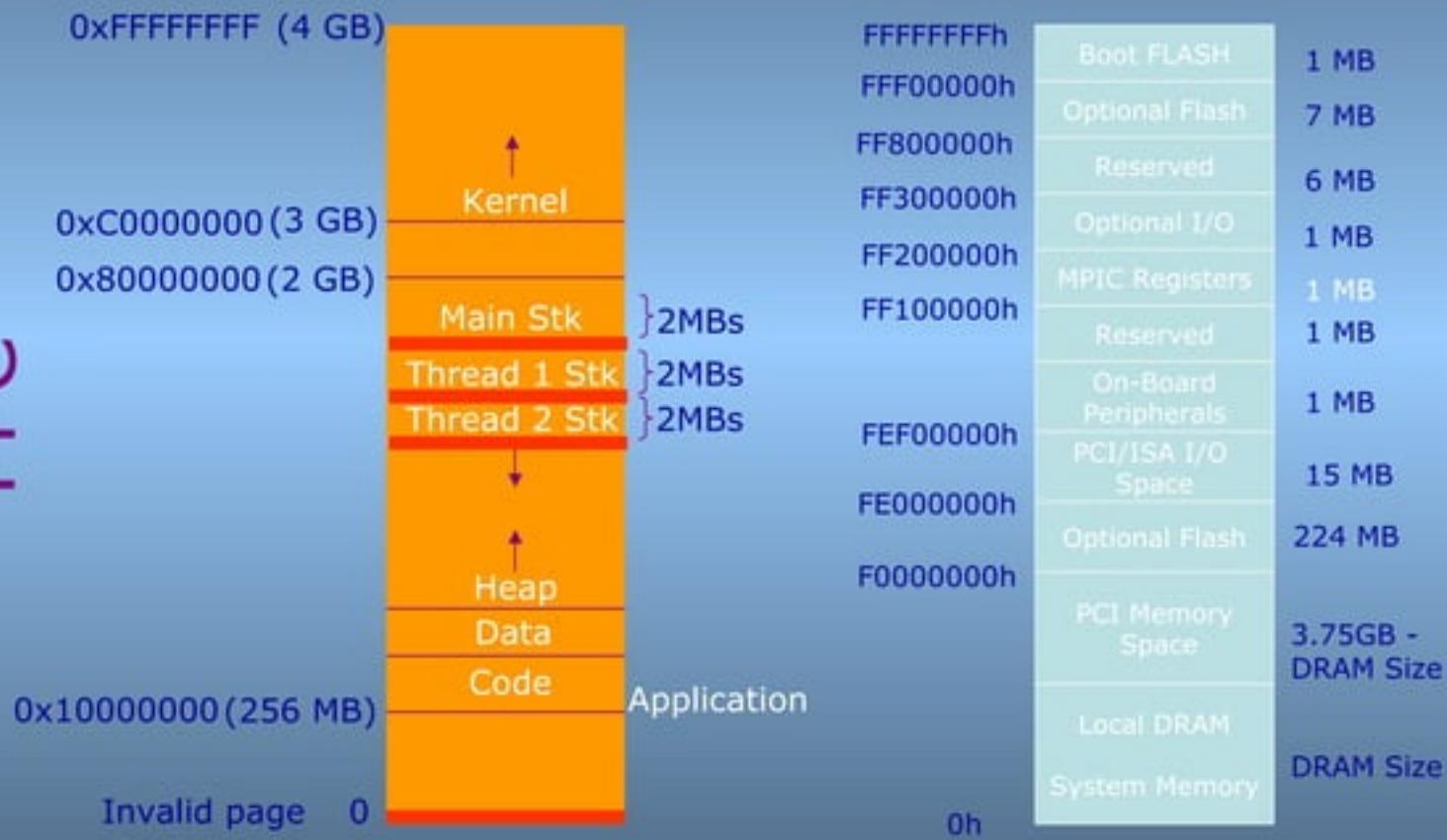
- Memory Addressing
- Memory Management

Virtual memory

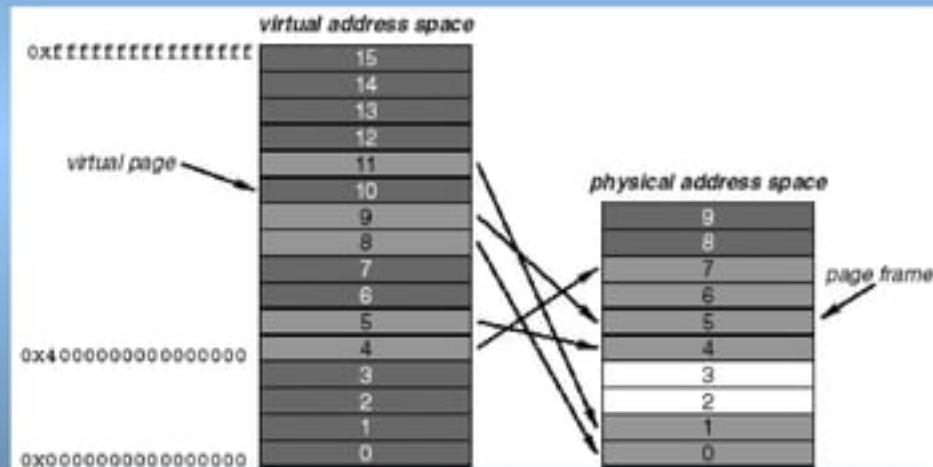
- Resource Virtualization
- Information Isolation
- Fault Isolation

Virtual vs. Physical Addresses

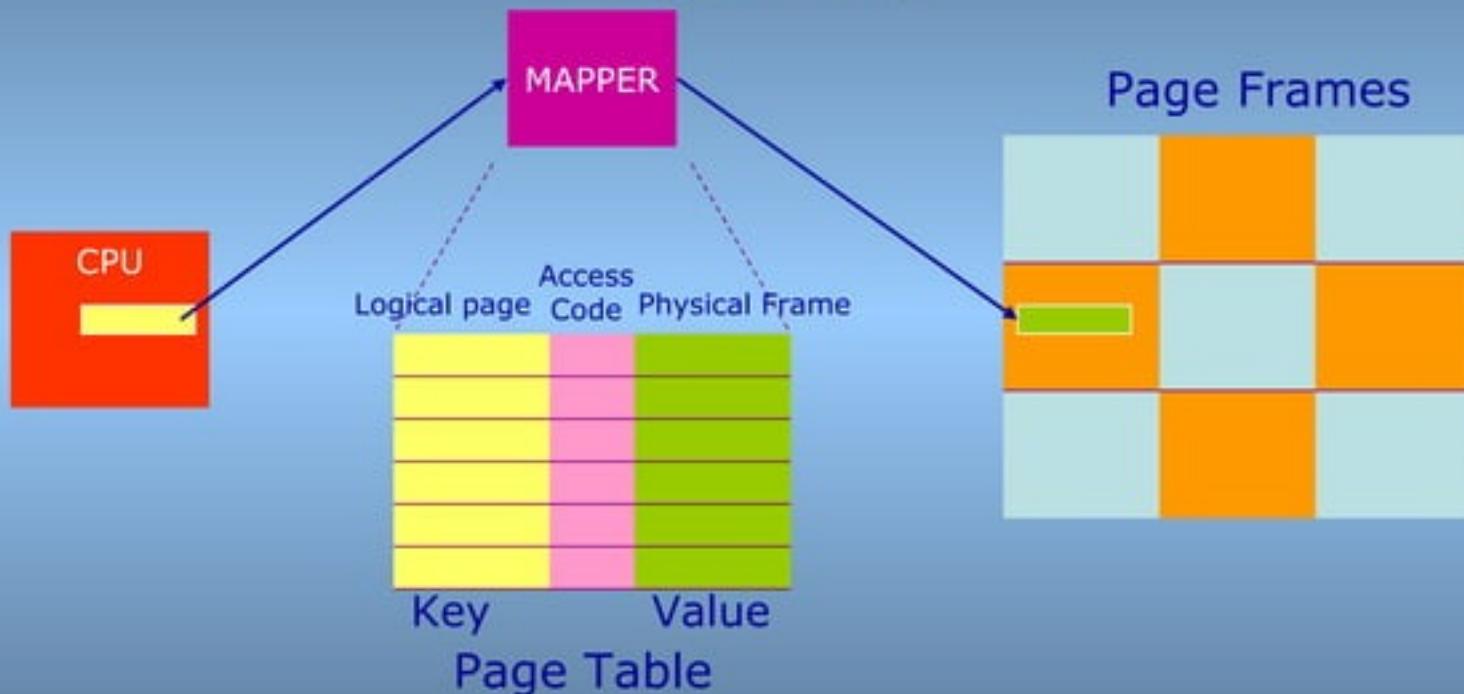
PPC



Virtual - Physical



Memory Mapping(virtual to Physical)



- RAM is divided into a collection of “pages”

Main Components

- Pages
 - Linux can be configured to support 4,8,16 KB
- Page Table
 - Mapping between virtual page frame and physical page
 - Maintained by linux in RAM
- MMU
 - To translate virtual address to physical address using page table

Concepts used in VM Domain

- Demand paging
- Paging and swapping
- Protection

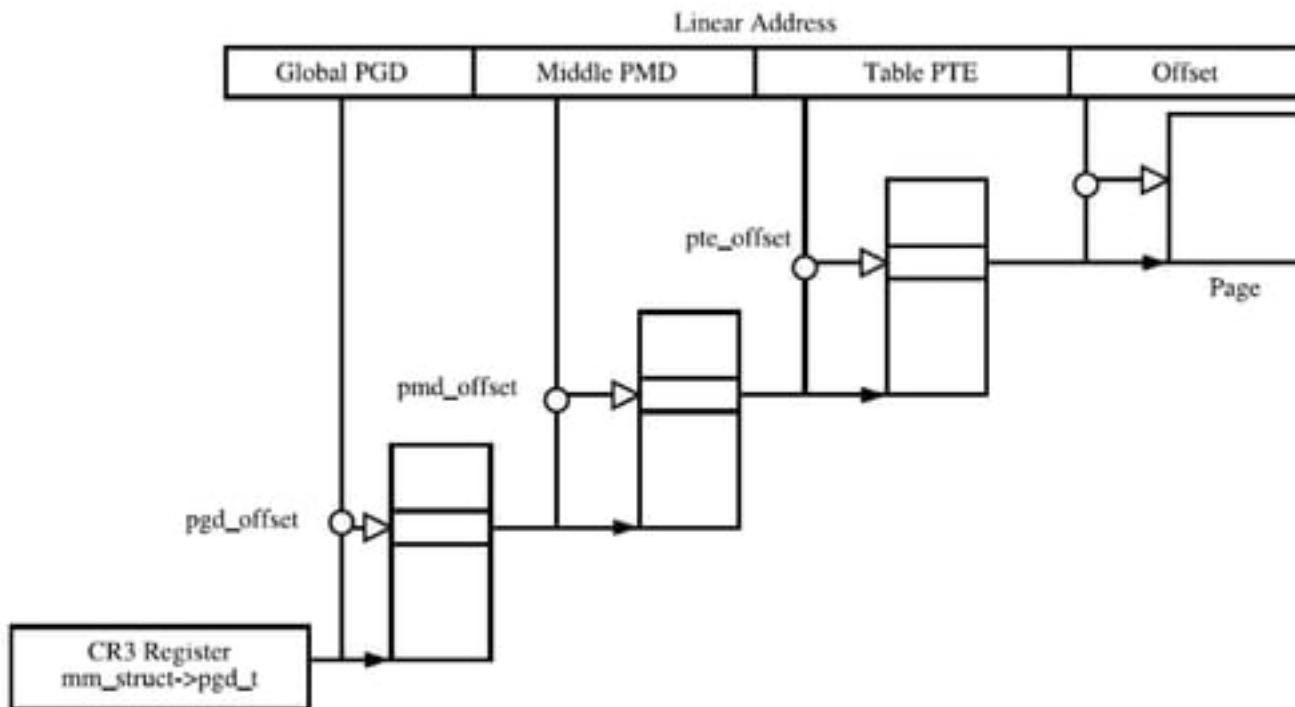
Demand Paging

- Physical pages not allocated when virtual memory is created
- Page table entries for virtual memory not present
- Page fault
- Allocates physical page
- Updates the page table

Memory Protection

- Protection bits for each page table entry
 - User
 - Read
 - Write
 - Execute
- Other bits
 - Page accessed
 - Page dirty
 - Page present

Page table



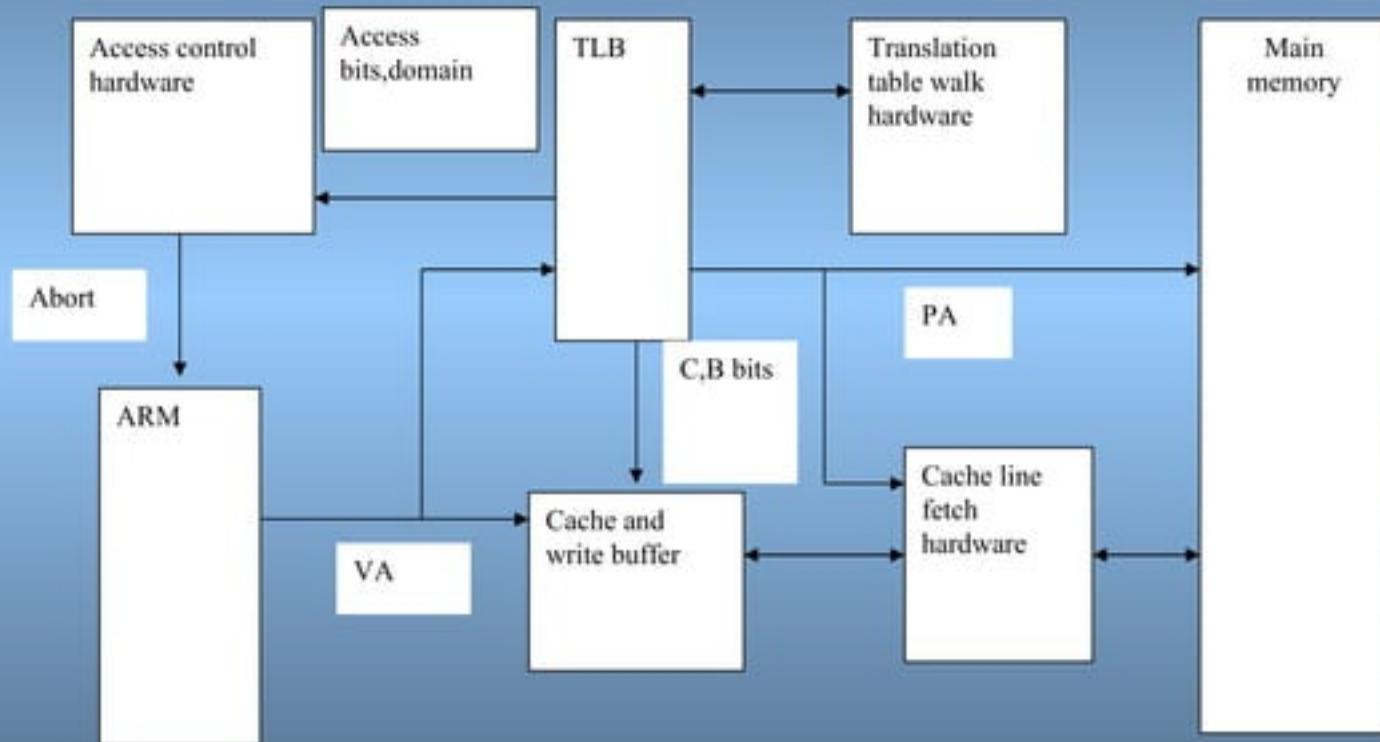
Page Table (cont)

- Each user process has page table
 - Mm_struct -> pgd
- kernel uses a separate page table
- In contrast to user space, one page table per process
- Single kernel page table

TLB

- A cache of page table entries is kept in the MMU's translation look-aside buffers (TLBs)
 - There are a limited number of TLBs on a processor
 - There is a big performance penalty for flushing a TLB
 - You may have to do that if you run out of TLBs when doing a context switch.
- Flushing a TLB may cause the instruction/data cache to be flushed
- The use of threads minimizes TLB/cache flushing
- Other mechanisms supported by hardware and software to avoid TLB flushing penalty

ARM-Memory Management



Points to remember for memory management

- Enable/disable MMU
- Page sizes
- Translation table base
- Access permission
- MMU faults
- Memory Coherency
- Page table macros/api
- Address space
- Context switching

KERNEL ADDRESS SPACE

- Linux memory is broken into two distinct regions
 - Kernel address space
 - User address space
- Kernel space is a “flat” memory region
 - Can address all physical RAM
 - Still uses “logical” (aka “virtual”) addresses
 - Macros provide translations from physical/bus addresses to logical addresses and back
 - Kernel memory is never “swapped” to disk

USER ADDRESS SPACE

- Unlike kernel space, user space cannot directly access physical addresses
- Each user process has its own virtual address map
- User memory can be swapped to disk unless it is locked into memory via `mlock()`
- All memory defaults to being private to the user process

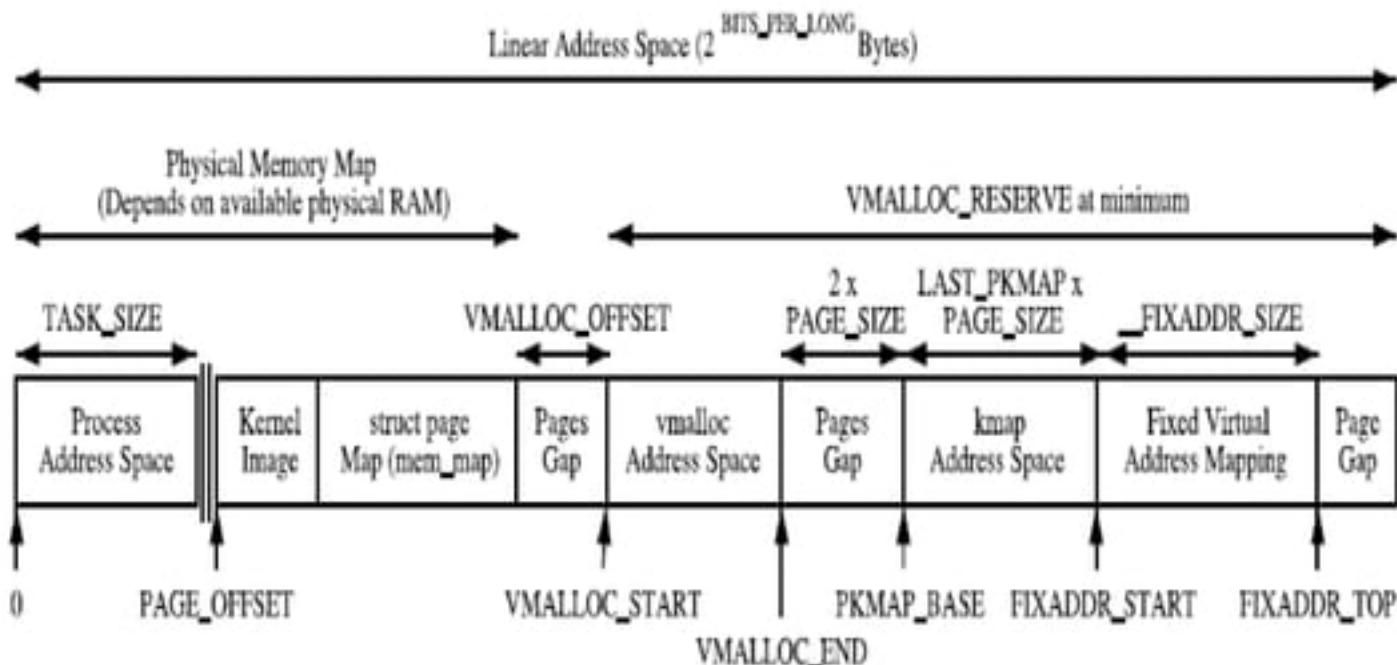
Code Executing in Kernel Space

- The kernel itself
- Device drivers/Loadable Modules
 - A loadable module is code that is not compiled into the kernel (that code could be a driver)
 - Typically executes on behalf of a user process but in kernel space
- Interrupt Service Routines
 - Usually installed by a device driver
- Exception handlers
- Kernel threads
- Tasklets/Bottom halves/Soft IRQs

Code Executing in User Space

- User processes
 - The programs you run
- User threads
 - Sub-tasks run within a process address space
- Daemons/Servers
 - portmap, nfs, xinetd, etc.
- Shared Libraries
 - A shared library contains functions that are used at run time, potentially by many programs
 - Dynamically loaded/unloaded

Address Space



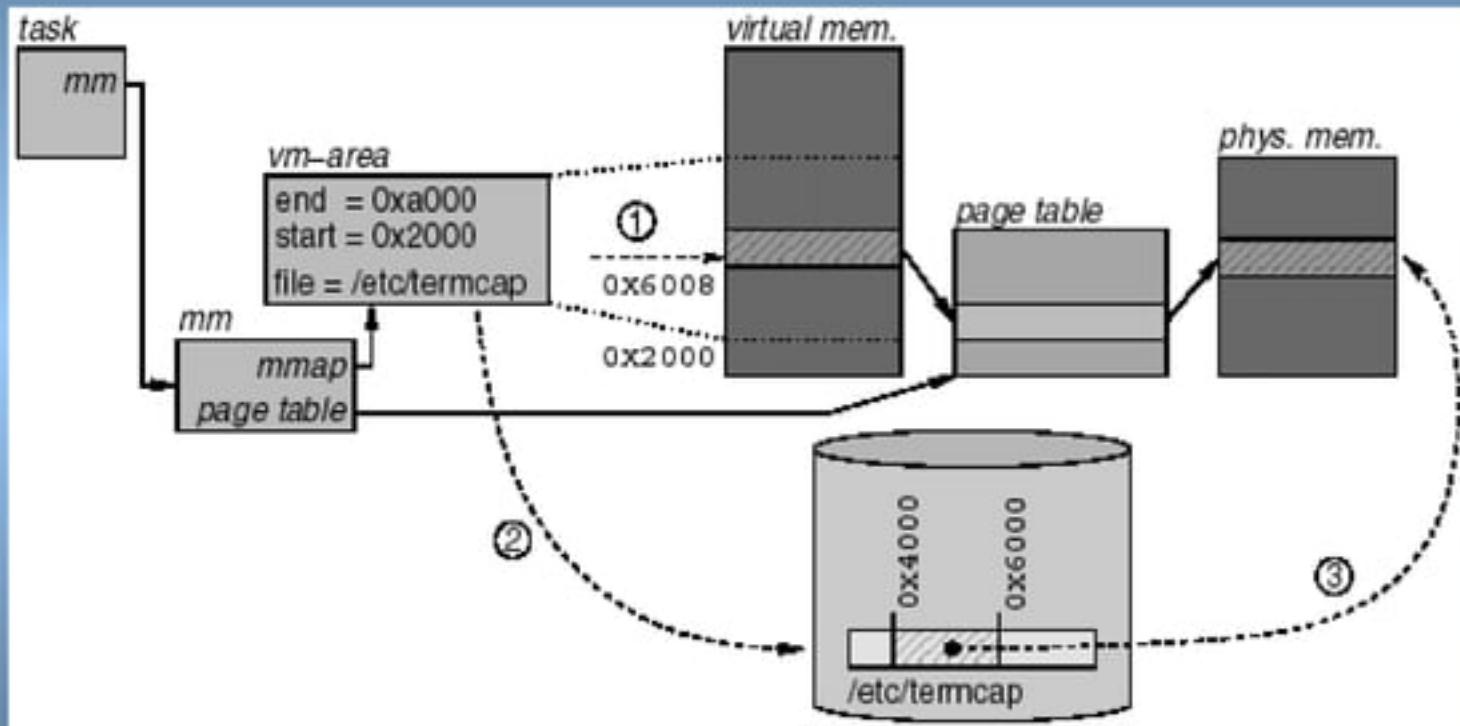
Process Address space

- Mm_structure
- Each task_structure has pointer for mm_structure
- Process can share the address space
- Kernel processes run in single address space
- All kernel process task_structure ,contains mm_pointer → NULL

MM Structure

- PgD
 - Process Page table directory required to manage virtual address space mapping to physical address
 - Page table
- Vm_area_structure
 - Divide the address space into contiguous ranges of pages handled in same fashion

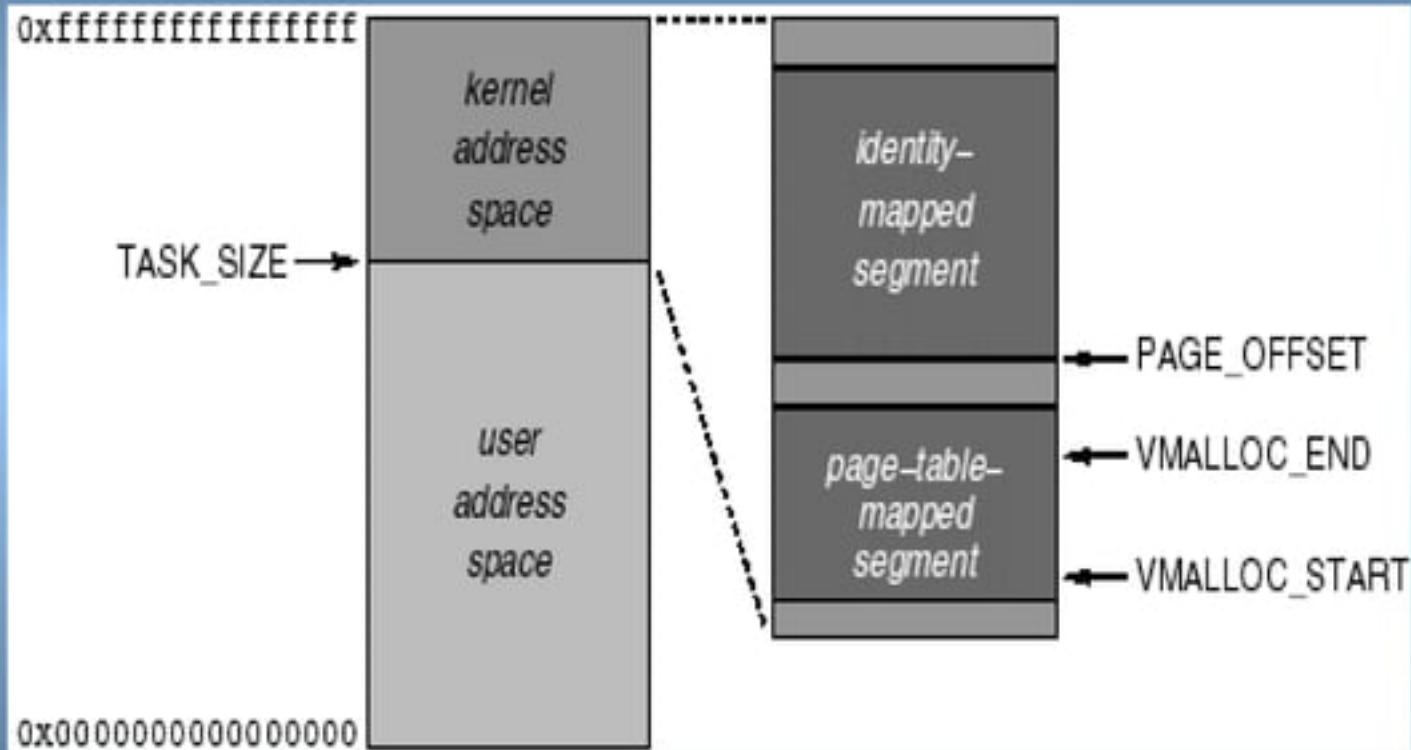
Example



Process address space(cont)

- Mmap
 - Providing the user process access to the device memory
 - Creates new `vm_area_struct` defining the virtual address range
 - Create page table entry for the corresponding virtual address mapping to device memory
- Shared memory

Kernel Address Space



Kernel address space

- Identity mapped
 - Starts at PAGE_OFFSET(0xc0000000)
 - Direct mapping
 - Unsigned long _pa(vaddr)
 - Void * __va(paddr)
- Page-table mapped
 - Kernel page table
 - Vmalloc area
 - Master page table directory(init_mm->pgd)

Page Fault

- When process access virtual page for which there is:
 - No PTE entry in page table
 - Page not present
 - Access right

Page Fault

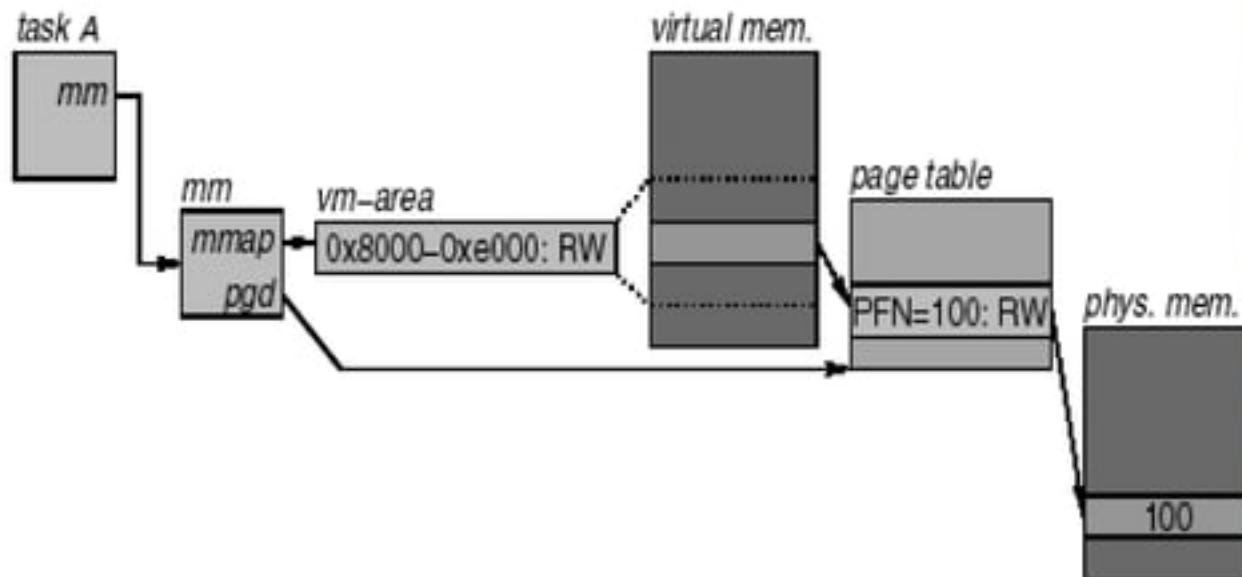
- Page fault handler (do_page_fault)
 - Not valid page access
 - Segmentation fault
 - Page accessed for first time
 - Demand fetching
 - Page swapped out
 - swapping
 - Other page fault optimization
 - COW (copy-on-write)

Page fault (cont)

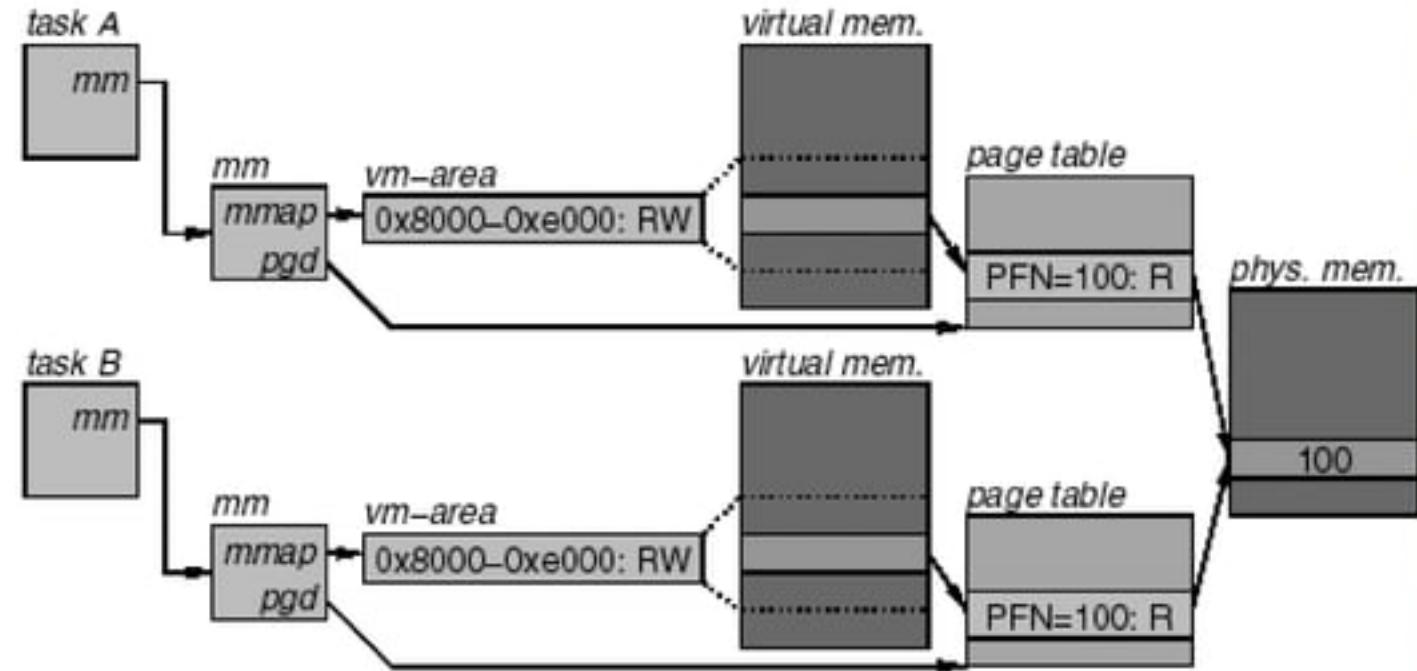
- Actions performed on page fault
 - Segmentation fault
 - Demand_page_faults (first time access)
 - Page swapped out
 - Page fault based optimization (e.g COW)

Page Fault (COW)

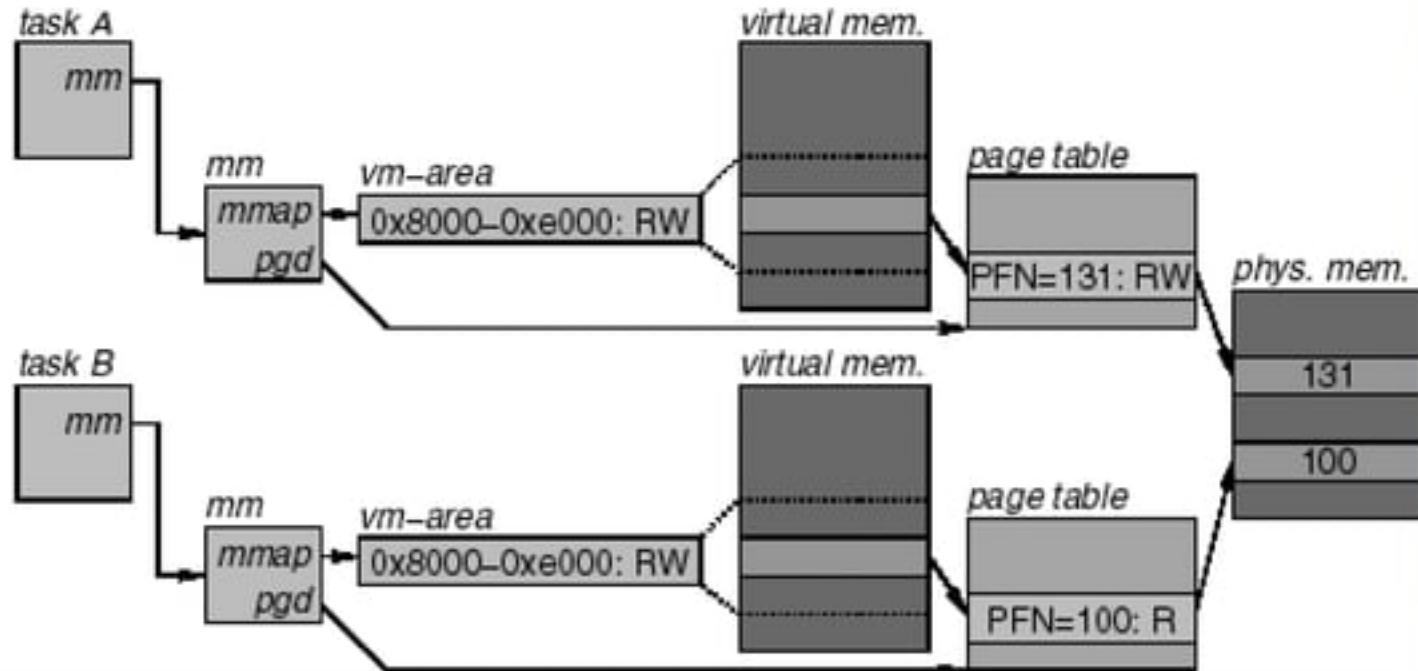
(a) before *fork()*:



(b) after *fork()*:



(c) after write access by task A:



Memory manager components

- Page allocator
- Slab allocator
- Mempool
- Kmalloc
- Vmalloc
- Kswapd
- Page cache
- Etc...

Kernel dynamic memory allocation

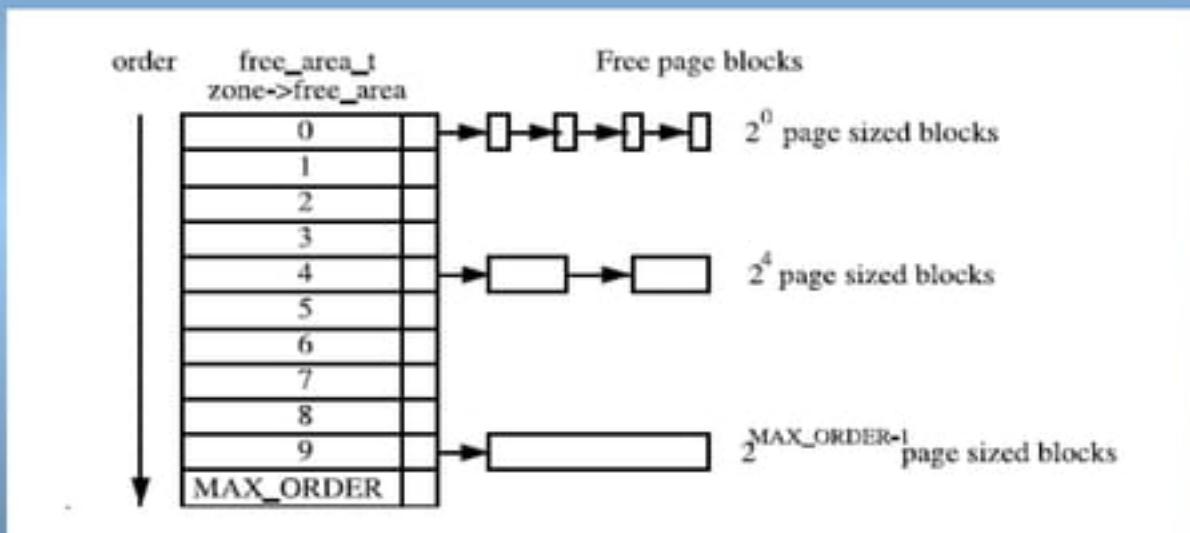
- Kmalloc
 - Allocates memory in kernel space
 - `ptr = kmalloc(size,flags)`
 - size is size in bytes flags are defined as below
 - Flags is broken to 3 categories
 - Action modifiers, zone modifiers & Types
 - Action modifiers specify how kernel is suppose to allocate memory
 - Zone modifiers tell from which zone memory is required
 - Types indicate combination of action & zone modifiers
 - 128KB memory allocation

Non-Contiguous memory allocation

- `vmalloc` – non-contiguous physical but contiguous virtual memory
- `VMALLOC_START` – `VMALLOC_END`
- Page table to be adjusted with page allocator
- Used mainly for storing swap map information / loading kernel modules into memory

Physical page allocator

- Page allocator



Physical Page allocator(cont)

- `alloc_page(unsigned int gfp_mask)`
Allocate a single page and return a struct address
- `alloc_pages(unsigned int gfp_mask, unsigned int order)`
Allocate 2^{order} number of pages and returns a struct page
- `get_free_page(unsigned int gfp_mask)`
Allocate a single page, zero it and return a virtual address
- `__get_free_page(unsigned int gfp_mask)`
Allocate a single page and return a virtual address

Physical Page allocator(cont)

- `__get_free_pages(unsigned int gfp_mask, unsigned int order)`
Allocate 2^{order} number of pages and return a virtual address
- `__get_dma_pages(unsigned int gfp_mask, unsigned int order)`
Allocate 2^{order} number of pages from the DMA zone and return a struct page

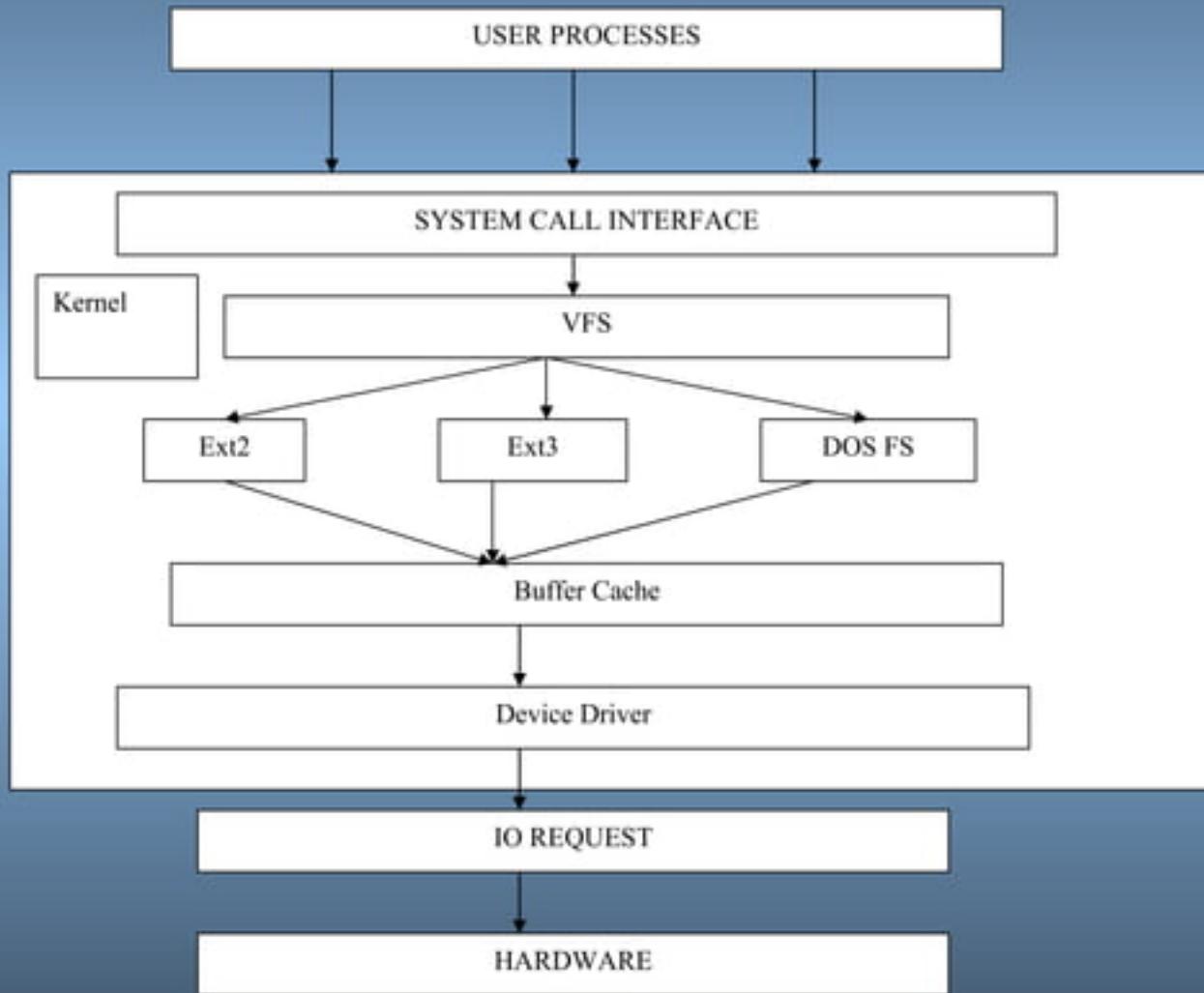
GFP Flags

- GFP_KERNEL is the default flag. Sleeping is allowed.
- GFP_ATOMIC is used in interrupt handlers. Never sleeps.
- GFP_USER for user mode allocations. Low priority, and may sleep. (Today equal to GFP_KERNEL.)
- GFP_NOIO must not call down to drivers (since it is used from drivers).
- GFP_NOFS must not call down to filesystems (since it is used from filesystems -- see, e.g., dcache.c:shrink_dcache_memory and inode.c:shrink_icache_memory).

Slab Allocator

- The slab allocator (SA) provides an efficient mechanism for allocating sub-page pieces of memory and re-usable objects with static structure.
- It benefits the system by helping to speed up these types of allocation requests and, in addition, decrease internal fragmentation.
- A series of caches are kept, each responsible for managing "slabs" of like objects defined by various properties including an optional constructor/deconstructor pair. A slab is a series of contiguous physical pages in which objects are kept.
- Slab allocator allocate/free small pieces of memory using `kmalloc()` and `kfree()`.

VFS



What is VFS

- It is a kernel software layer that handles all system calls related to a standard Unix filesystem.
- Filesystem supported by the VFS may grouped into 3 classes
 - Disk-based filesystem
 - Network filesystem
 - Special filesystem

4 Basic FS abstractions

- Files
- directory entries
- Inode
- Mount points

Main features for FS to work with VFS

- Superblock
- Inode
- Directories as files
- Mount

4 Primary objects of VFS

- **Superblock**
 - Information about specific mounted fs
- **Inode**
 - Information about specific file
 - Each file represented by inode number
- **dentry**
 - Information for specific directory entry
- **files**
 - represents an open file as associated with process

Root File-system

- Starting with installing file-system
 - Swap partition
 - Root partition
- Mount root-filesystem
 - Provide major number of disk
 - Provide root argument (root=/dev/hda1)

Root File-system (cont)

- Mount of rootfs in 2 stages
 - Mount special rootfs, empty directory providing initial mount point
 - Mount real root filesystem in the empty directory
- Mount needs to provide the file-system type supported by VFS(ext2,ext3,crampfs etc..)

Initrd

- Initial Ram-Disk
- temporary root file system that is mounted during system boot to support two-state boot process
- In many embedded systems, initrd is the final root file systems

File-System on system

- /boot/initrd-2.6.xxx.img.gz
 - Compressed image
 - **gunzip initrd-2.6.14.2.img.gz**
 - **cpio -i --make-directories < initrd-2.6.14.2.img**
 - Mkinitr – to build initrd image at the time of linux building

Flash-system

- Fdisk
 - Create partition
- Mk2efs.ext2
 - Create new ext2 partition
- mount
 - Mount the file system
- Populate the file-system
 - Directories
 - Important files , utilities
 - /etc/fstab mounting entry to mounting the file-system during system boot-up
 - Use busybox to build up the binaries

Manually building custom init tam-disk

- `#!/bin/bash`
- `# Housekeeping...`
- `rm -f /tmp/ramdisk.img`
- `rm -f /tmp/ramdisk.img.gz`
- `# Ramdisk Constants`
- `RDSIZE=4000`
- `BLKSIZE=1024`
- `# Create an empty ramdisk image`
- `dd if=/dev/zero of=/tmp/ramdisk.img bs=$BLKSIZE count=$RDSIZE`
- `# Make it an ext2 mountable file system`
- `/sbin/mke2fs -F -m 0 -b $BLKSIZE /tmp/ramdisk.img $RDSIZE`
- `# Mount it so that we can populate`
- `mount /tmp/ramdisk.img /mnt/initrd -t ext2 -o loop=/dev/loop0`

- # Populate the filesystem (subdirectories)
- mkdir /mnt/initrd/bin
- mkdir /mnt/initrd/sys
- mkdir /mnt/initrd/dev
- mkdir /mnt/initrd/proc
- # Grab busybox and create the symbolic links
- pushd /mnt/initrd/bin
- cp /usr/local/src/busybox-1.1.1/busybox
- .ln -s busybox ash
- ln -s busybox mount
- ln -s busybox echo
- ln -s busybox ls
- ln -s busybox cat
- ln -s busybox ps
- ln -s busybox dmesg
- ln -s busybox sysctlpopd

```
•      # Grab the necessary dev files
•      cp -a /dev/console /mnt/initrd/dev
•      cp -a /dev/ramdisk /mnt/initrd/dev
•      cp -a /dev/ram0 /mnt/initrd/dev
•      cp -a /dev/null /mnt/initrd/dev
•      cp -a /dev/tty1 /mnt/initrd/dev
•      cp -a /dev/tty2 /mnt/initrd/dev
•      # Equate sbin with bin
•      pushd /mnt/initrd
•      ln -s bin sbin
•      Popd
•      # Create the init file
•      cat >> /mnt/initrd/linuxrc << EOF
•      #!/bin/ash
•      Echo
•      echo "Simple initrd is active"
•      Echo
•      mount -t proc /proc /proc
•      mount -t sysfs none /sys
•      /bin/ash -login
•      EOF
•      chmod +x /mnt/initrd/linuxrc
•      # Finish up...
•      umount /mnt/initrd
•      gzip -9 /tmp/ramdisk.img
•      cp /tmp/ramdisk.img.gz /boot/ramdisk.img.gz
```

Points to think about:

- NOR flash
 - Block size (64kb-256Kb)
 - Ext2 file system typically support 4kb
 - Erase cycle of NOR flash
 - Cost of NOR flash
 - Compresses file system could be better
 - Journaling file-system
 - logs changes to a journal (usually a circular log in a dedicated area) before committing them to the main file system.
 - less likely to become corrupted in the event of power failure or system crash.

Different file-system

- MINIX
- Ext2
- Ext3
- Jffs2
- Crampfs
- Squashfs
- Etc..

Comparison Parameters

- Crash stability
- File creation time
- File read/write performance
- Maximum file-length
- Allowable characters in directory-entry
- Maximum file-size
- Maximum disk-size
- Metadata
 - Time-stamps
 - Ownership
 - Permissions
 - Checksum/ECC
- Hard-links
- Soft-links
- Journaling
- Case-sensitive
- XIP
- Compression
- Etc...

Special FileSystems

- Bdev
- Futexfs
- Pipefs
- Proc
- Rootfs
- Shm
- Mqueue
- Sockfs
- Sysfs
- Tmpfs
- Usbfs

PROC FILE SYSTEM

- **/proc** directory, referred to as a virtual file system.
- Contains a hierarchy of special files which represent the current state of the kernel.
- A method for User/Kernel Communications.
 - can be opened and read like normal files
 - Entries can be read and parsed by shell scripts, perl, awk/sed, etc.
 - device drivers can add /proc entries that can be used to find out status information without writing ioctl calls
- Allows applications and users to peer into the kernel's view of the system.
- Information detailing the system hardware and any processes currently running.
- Constantly updated.

IPC

- Signal
- Message queue
- Pipe
- Fifo
- Shared memory
- semaphore

signal

- Introduced in UNIX systems to simplify IPC.
- Used by the kernel to notify processes of system events.
- A signal is a short message sent to a process, or group of processes, containing the number identifying the signal.
- No data is delivered with traditional signals.
- POSIX.4 defines i/f for queueing & ordering RT signals w/ arguments.

Signal (cont)

- To make process aware of specific event
- To cause a process to execute a signal handler function

Signal (cont)

- Linux supports 31 non-real-time signals.
- POSIX standard defines a range of values for RT signals:

SIGRTMIN 32 ... **SIGRTMAX (_NSIG-1)**
in **<asm-*/signal.h>**

Signal transmission

- Signal sending
 - Signal can be send by kernel or other process
 - Kernel updates descriptor of destination process
- Signal receiving
 - Kernel forces target process to “handle” signal.
 - *Pending signals* are sent but not yet received.
 - Up to one pending signal per type for each process, except for POSIX.4 signals.
 - Subsequent signals are discarded.
 - Signals can be blocked, i.e., prevented from being received.

Signal Handling

- 3 ways a process responds to signal:
 - Ignore
 - Execute default action
 - Catch the signal by invoking corresponding signal handler

pipes

- A pipe is a one-way flow of data between processes.
- All data written by processes to the pipe is routed by the kernel to another process.
- Pipes can be created by means of ' | ' operator
\$ ls | more

Here the output of the first process (executes ls) is directed to the pipe, the second (executes more) reads the input from the pipe.

Pipes (cont)

- A unidirectional communications path limited to 4K (1 page) of data in the pipe at any point in time
- Both anonymous pipes and named pipes (FIFOs) are supported
- It typically allows multiple writers (senders) but only one reader(receiver)

Pipes (cont)

- When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe.
- One descriptor is used to allow a path of input into the pipe (write), while the other is used to obtain data from the pipe (read).

Named Pipe

- They also known as FIFO
- IT is a special type of file that exists as a name in the file system
- We can create named pipes from the command line
 - mknod filename p
 - mkfifo filename

Named Pipe (cont)

- From a program you can create them using
 - `int mkfifo (const char *filename, mode_t mode)`
 - `int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);`

Named Pipe (cont)

- `cat < /tmp/my_fifo &`
- `echo "abcd" > /tmp/my_fifo`
- You can open the named files from regular ‘open’ call and then do a regular read or write
- You can write client-server program using pipes

Message Queue

- This is also an IPC mechanism where each message generated by a process is sent to an IPC message queue, where it stays until another process reads it
- To send a message, a process invokes the `msgsnd()` function.
- To retrieve a message, process invokes the `msgrcv()` function.

Shared Memory

- This allows two or more processes to access some common data structure by placing them in an IPC shared memory region
- Each process wants to access data structures included in shared memory region must add to its address space a new memory region ,which maps to page frames associated with shared memory region

Semaphore

Used for synchronization which implements a locking primitive that allows waiters to sleep until the desired resource becomes free.

Linux offers two kinds of semaphores

- Kernel semaphore, which are used by kernel control paths
- System V IPC semaphore which are used by user mode processes.

Mutex

- Mutexes are used for synchronization purpose to be sure that two threads do not attempt to access memory at the same time.
- Unlike semaphore it can be acquired by only one thread/process at a time.
- We can use the mutexes for locking by use of following system calls...

`pthread_mutex_lock()`

`pthread_mutex_unlock()`

Kernel Preemption

Some Terminology

- user mode
- kernel mode
- interrupt context
- process context

Current implementation terminology:

- hard interrupt context
- soft interrupt context
- user context

Preemption

- In kernel mode CPU is relinquished :
 - planned way (process in sleep state , waiting for resource etc , self-relinquished)
 - forced way (higher priority process)
- Either way 2 steps are common:
 - switch_to function called
 - scheduler invoked

Non-Preemptive/Preemptive kernel

- Non_Preemptive kernel:
 - Forced Process switch does not take place while in kernel mode. Process switch happens while returning to user mode.
- Preemptive kernel:
 - Process in kernel mode can be replaced by another process while in the middle of kernel function.

How Preemption affects execution

- Process A → exception handler (kernel mode)
IRQ raised → wakes up process B
Switch A → B (forced)
When A is runnable , exception handler continues.
- Process A → exception handler (kernel mode)
IRQ raised → wakes up process B
A continues to run.

How Preemption affects execution(cont)

- Process A → exception handler (kernel mode)
Time quantum expires
Immediate replacement (forced)
- Process A → exception handler (kernel mode)
Time quantum expires
No immediate replacement

Linux support for kernel preemption

- To support kernel preemption :

Prompt_count

- interrupt context (interrupt or deferred function)
- explicitly > 0

Kernel preemption

- Can be preempted:
 - when executing exception handler (system calls)
 - kernel preemption not set explicitly
 - local CPU must have local interrupts enabled
- 2 conditions checked when `preempt_schedule` function called:
 - interrupt enabled
 - `preempt_count == 0`

Synchronization support in kernel

Kernel Synchronization

- Critical Region
 - Must be completely executed by kernel control path that enters it before another path can enter
 - Exception handlers, interrupt handlers, bottom-half, kernel threads
 - Concurrency for drivers (multiple processes trying to access driver at same time)

Why is it Required?

- Sharing between 2 interrupt handlers on single CPU
 - Disable interrupts
- Sharing between service routines of system calls on single COU
 - Disable kernel preemption
- Why sync tools? Multi-processor architecture?

Usage requires to :

- Understand the kernel control paths and their implementation
 - Results in indicating the cases where kernel sync is not necessary
- Some cases:
 - Interrupt line disabled during interrupt handling
 - Interrupt handlers/Bottom half are non-preemptable and non-blocking
 - Interrupt handler cannot be interrupted by kernel control path like system call service routine
 - Same tasklets cannot be executed simultaneously on several CPU

Sync kernel primitives

- Per-cpu variables
- Atomic operations
- Spinlocks
- Read/write spinlocks
- Seqlocks
- RCU
- Semaphore
- Read/write semaphore
- BKL

Atomic operations

- Read-modify-write instructions between multiple CPU's
- Instructions that execute without interruption.
 - Integers
 - `void atomic_add(int i, atomic_t *v)`
 - `void atomic_sub(int i, atomic_t *v)`
 - Individual bit
 - `void set_bit(int nr, void *addr)`
 - `void test_bit(int nr, void *addr)`

Spinlock

Can be held at most by one thread of execution

- If not available, thread will busy loop
 - `spin_lock(spinlock_t &slock)`
 - `spin_unlock(&slock)`
- Same lock can be used in multiple locations.
- Architecture dependent, implemented in Assembly.
- Can be used in Interrupt
 - `spin_lock_irqsave(spinlock_t &slock, ulong flags)`
- Can be used in Interrupt
 - `spin_lock_irqsave(spinlock_t &slock, ulong flags)`
- `spin_unlock_irqresote(&slock,flags)`

What spinlock does?

- Kernel preemption disabled
- In uniprocessor, kernel preemption is disabled

Sqlocks

- In read/write spinlocks , same priority given for read and write, so write has to wait
- Seqlocks gives higher priority to writers
- Readers don't need to disable kernel preemption
- Writers disable kernel preemption

RCU

- Many readers / many writers to proceed concurrently
- There is no shared lock/counter between CPU
- Used for data structures that are dynamically allocated / referred by pointers
- No kernel control path can sleep

RCU flow

- Read
 - Disable preemption
 - Dereference pointer
 - Preemption enabled at the end
- Writer
 - Dereference pointer
 - Modify the copy
 - Change the pointer to new updated copy

Semaphores

- **Sleeping locks.**
 - Suited for locks that can be held a long time.
 - Only in process context, not in interrupt context.
 - Cannot acquire a spinlock while holding a semaphore.
- **Binary/mutex semaphore**
- **Counting semaphore**

Sempahore (cont)

- `down_interruptible()` .. Attempts to acquire the semaphore, it sleeps if it can't acquire the semaphore in “`TASK_INTERRUPTIBLE`” state
- `down()` .. Places the task in “`TASK_UNINTERRUPTIBLE`” state
- `down_trylock()` .. Returns immediately nonzero if lock is already held
- `up(&mysem)` .. `mysem` is a pointer to ‘semaphore’ structure
 - This Releases the semaphore
- Wakes up the waiting task if any

BKL

- **Global spinlock.**
 - Can sleep while holding BKL.
 - Recursive lock
 - Can be used only in process context
 - Preemption disabled
- **Current implementation:**
 - Kernel semaphore (`kernel_sem`)
 - Can be preempted while holding the lock

Interruption!!!

Interrupts and Exceptions

- a signal hardware/software sends when it wants the processor's attention.
 - Software Interrupts
 - Hardware Interrupts

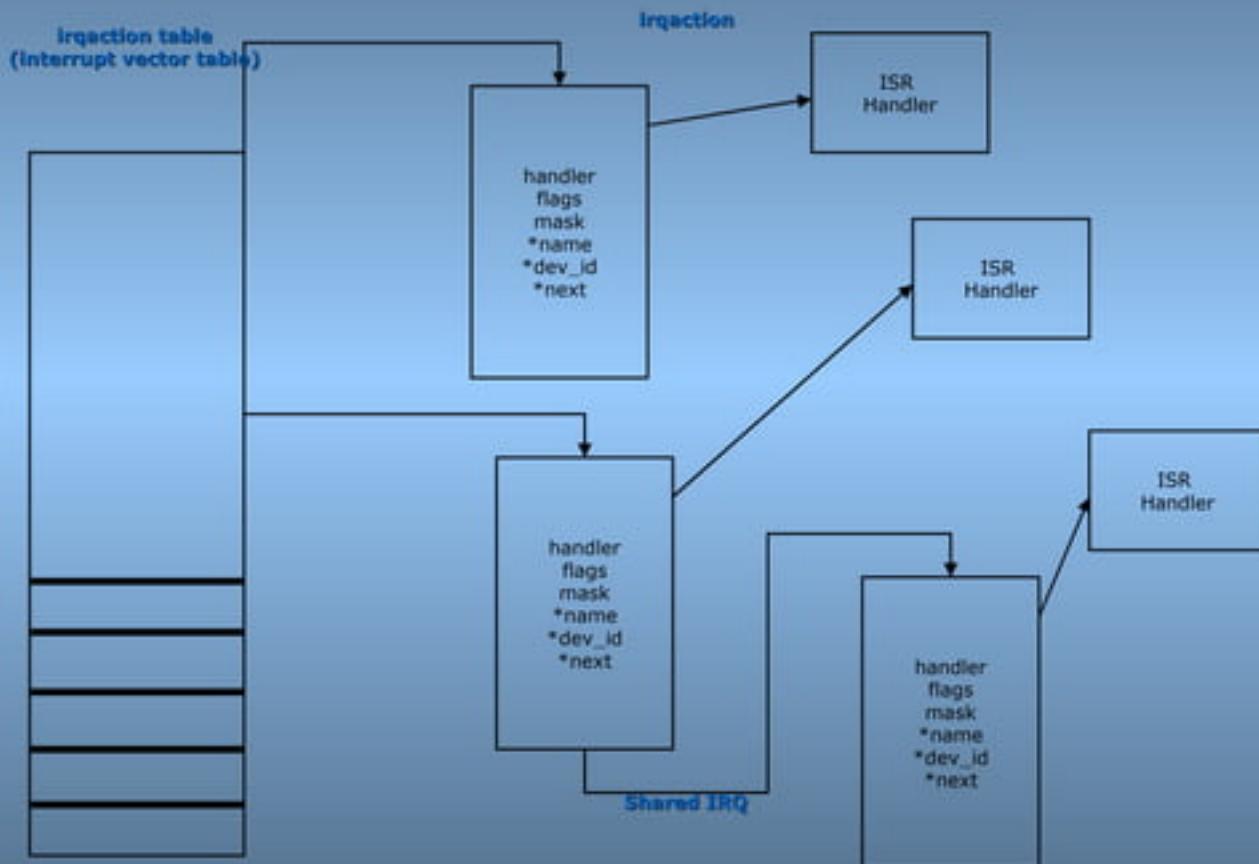
Linux Interrupt

- Interrupt features supported:
 - IRQ sharing
 - IRQ dynamic allocation
- Interrupt handlers divided in following sections:
 - Critical
 - Interrupt disable
 - Non-critical
 - Interrupt enables
 - Non-critical deferrable
 - Interrupt enabled
 - Delayed execution

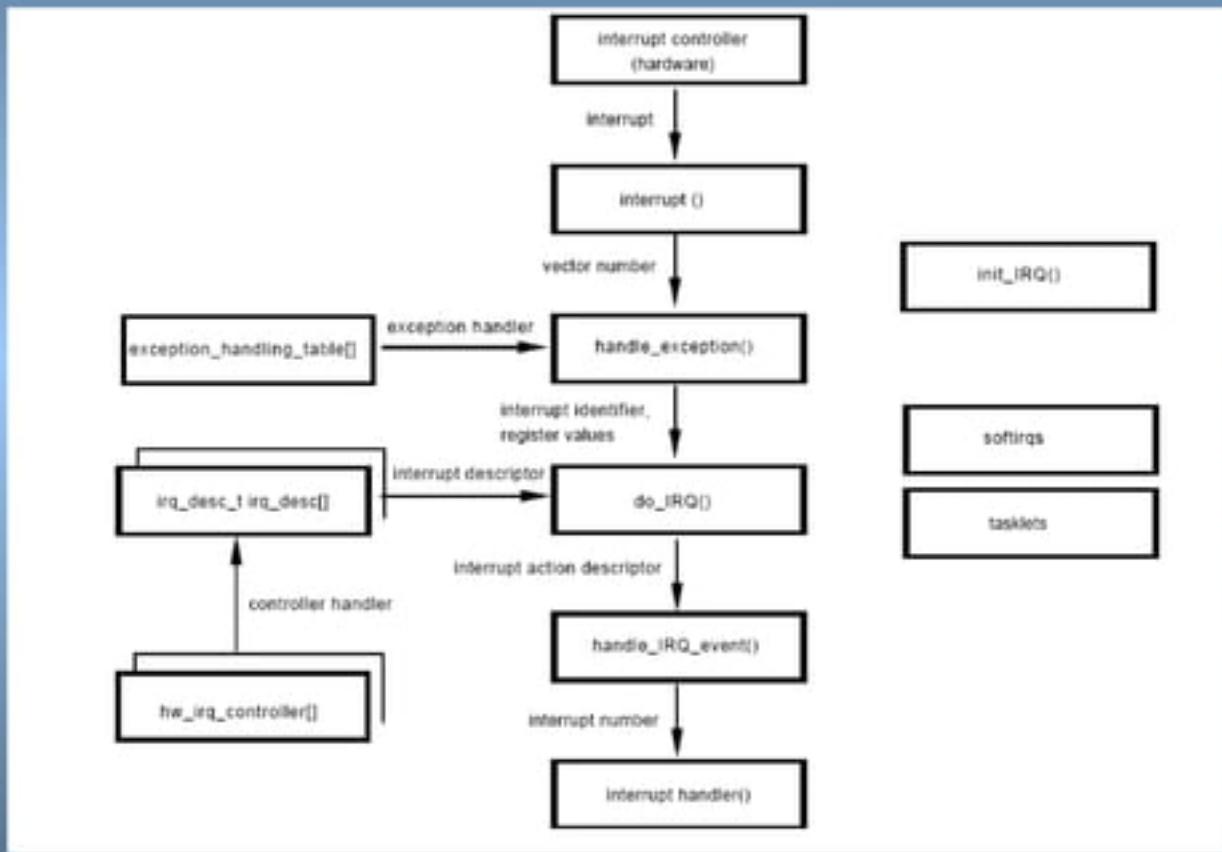
Additional Points:

- Stack usage
 - Current task in running state , kernel stack is used in case kernel stack is 8kB
 - If kernel stack of process is 4kb , separate interrupt stack is allocated
- Interrupt number (0-255)
- /proc/interrupt

Interrupt Registration



Interrupt Flow



Division of Interrupt handler routine

- top half: routine that responds to the interrupt (critical and non-critical)
 - register with request_irq.
 - Saves device data to buffers.
- bottom half: routine that is scheduled by the top half to be executed later, at a safer time (non-critical deferrable)
 - All interrupts are enabled during execution

Bottom-Half

- Bottom half is executed from any of the following points:
 - Local_bh_enable
 - When do_irq (linux irq entry handler) finished and irq_exit is invoked
 - Timer handler finishes handling local timer routine
 - Ksoftirqd/n kernel thread

Bottom-Half(cont)

- Register bottom-half handler
- Activate it from top-half interrupt handler
- `Do_softirq` parses the pending list of bottom-half handlers
- Interrupt occurring at fast pace can activate corresponding handler another instance
- If `do_softirq` keeps iterating few times, awakes `ksoftirq` kernel thread
- Thread gets CPU , executes the pending bottom-half handler

Types of Bottom-Half

- **SoftIRQ**
 - called by Kernel at the end of a system Call or called in a hardware interrupt handler
 - fully-SMP versions of BHs
 - can run on as many CPUs at once as required.
 - need to deal with any races in shared data using their own locks.
 - Much of the real interrupt handling work is done here
 - softirqs can be declared statically.
 - You can't dynamically register and destroy softirqs
 - They are defined in `linux/interrupts.h` & `kernel/softirq.h` files
 - There can be max of 32 softirqs

TASKLETS

They are bottom half mechanism

- They are built on top of softirq
- Two types of Tasklets
 - HI_SOFTIRQ
 - TASKLET_SOFTIRQ

Work-queues

- Defer work into Kernel threads
- Runs in process context
- Schedulable, can sleep

Time!!!

Timer

- Main points performed by linux kernel:
 - Keeping current time and date
 - Maintaining timers
- Timer interrupt generated by the time keeping hardware
 - HZ (100 → 100 interrupts per second)
 - Programmed at the kernel init time

Timer interrupt

- Functionality performed by timer interrupt:
 - Update time elapsed since system boot-up
 - Update time and date
 - Update time-slice of current process
 - Update resource usage statistics
 - Checks software timers list

Kernel structures

- Jiffies
 - Holds number of ticks since system booted
 - Incremented inside timer interrupt
 - Basic packet of kernel time, around 10ms on x86. Related to HZ, the basic resolution of the operating system.
 - The timer interrupt is raised each 10ms, which then performs some h/w timer related stuff, and marks a couple of bottom-half handlers ready to run if applicable.
- Xtime
 - Stores current time and date
 - Tv_sec
 - Tv_nsec

Types of timers

- 2 kinds of timers:
 - Dynamic timers
 - Used by kernel
 - Schedule_timeout
 - Adds a time in the timer list
 - Interval timers
 - Used in the user space
 - Uses SIGALRM , to signal the user process for time expiry
 - Setitimer and alarm

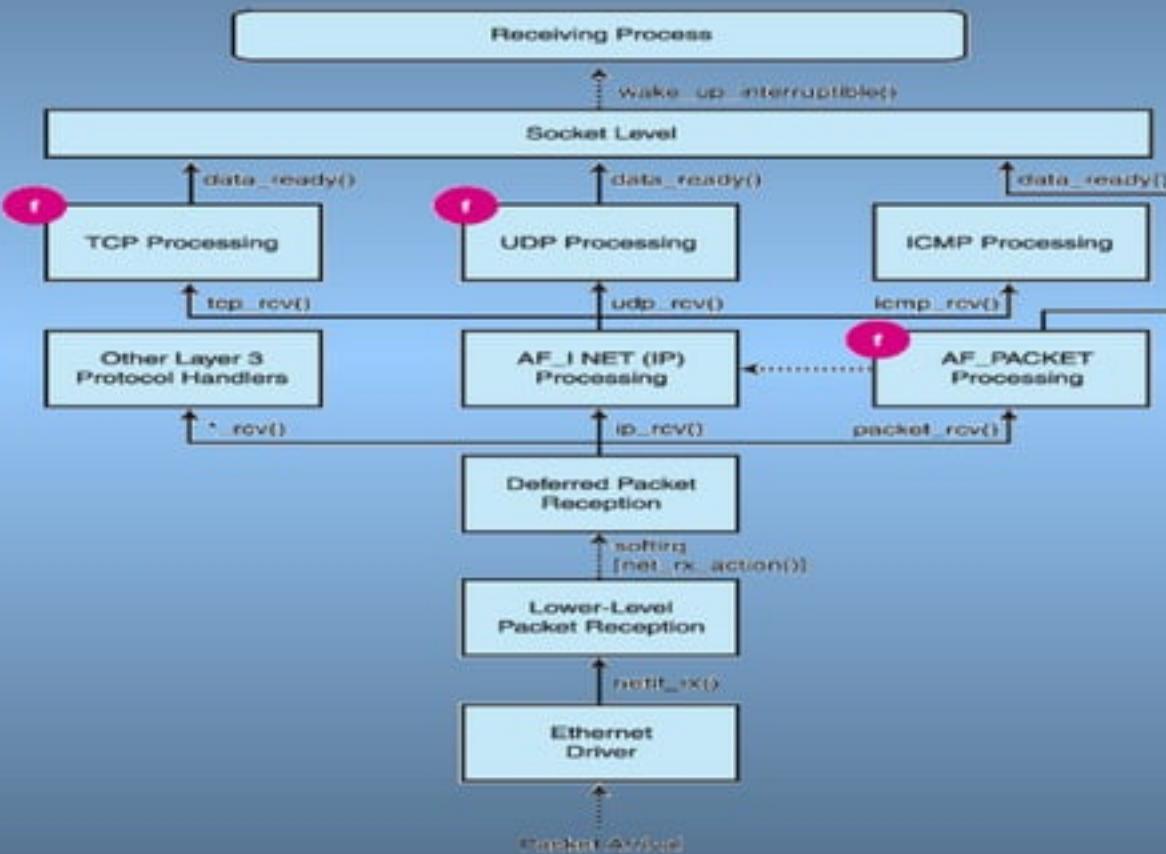
Delay functions

- Timers cannot be used for microseconds
 - Delay functions for kernel
 - Udelay
 - Ndelay
 - Delay functions loops for specified duration
 - User level delay
 - Sleep
 - usleep

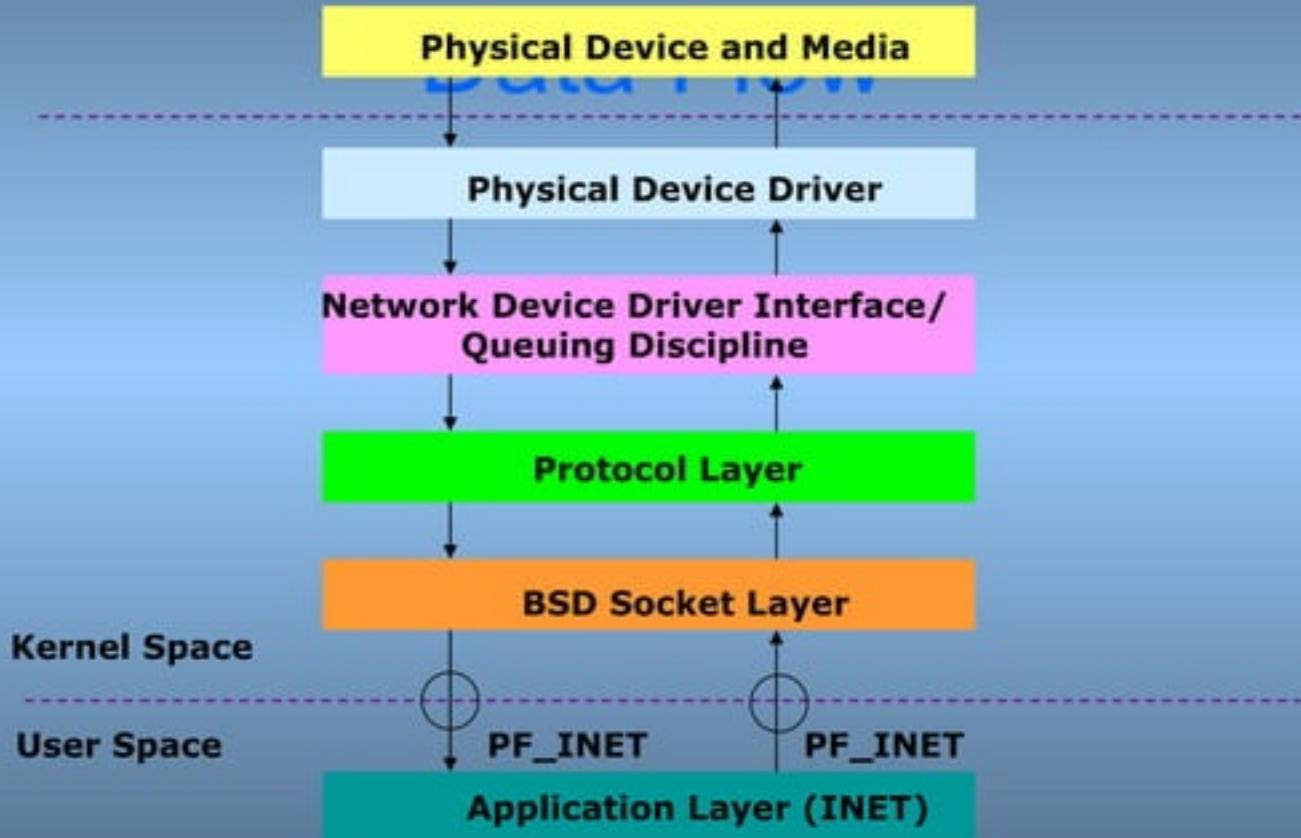
Network

- One of the greatest advantages of Linux is the breadth of its network protocol support
 - - IPv4, IPv6, mobile IPv6, PPP
 - - IPX, DECnet, AppleTalk, Econet
 - - IGMP, IPSEC, DHCP, 802.1Q VLAN, 802.1d Ethernet Bridging, SNMP and more
- Linux also supports a variety of network interface including:
 - - 10/100/1000BaseT, FDDI, Fibrechannel, WAN interfaces, and more

Network Flow



Linux Network



Some file for Configuring network

- LINUX NETWORK FILES:
- /etc/sysconfig/network - Defines your network and some of its characteristics.
- /etc/HOSTNAME - Shows the host name of this host. If your name is "myhost" then that is exactly the text this file will contain.
- /etc/resolv.conf - Specifies the domain to be searched for host names to connect to, the nameserver address, and the search order for the nameservers.
- /etc/host.conf - Specifies the order nameservice looks to resolve names.

COnFIGURING NETWORK(Cont)

- `/etc/hosts` - Shows addresses and names of local hosts.
- `/etc/networks` - Provides a database of network names with network addresses similar to the `/etc/hosts` file. This file is not required for operation.
- `/etc/sysconfig/network-scripts/ifcfg-eth*` - There is a file for each network interface. This file contains the IP address of the interface and many other setup variables.

Network Data structures

- Critical data structures:
 - struct `sk_buff`
 - This is where a packet is stored. The structure is used by all the network layers to store their headers, information about the user data (the payload), and other information needed internally for coordinating their work.
 - struct `net_device`
 - Each network device is represented in the Linux kernel by this data structure, which contains information about both its hardware and its software configuration.
 - struct `sock`
 - stores the networking information for sockets

Points to consider:

- Important points:
 - Elements of data structures
 - Allocation / deallocation
 - Buffer management
 - Network driver structure
 - Network device support API
 - PHY support API

Network driver helper interfaces

- Kernel/user space interaction:
 - Procfs
 - Sysfs
 - ioctl
 - netlink socket

To begin with

- Initialization of NIC:
 - IRQ
 - Memory region mapping

Interaction b/w device and kernel:

- Polling
- Interrupt
 - » reception of frame
 - » transmission failure
 - » transmit buffer space available
 - » dma transfer complete

Configuring Tools

- Tools:
 - Iputilis
 - Net-tools
 - IPROUTE2
 - Ethtool
 - Mii-tools

Loading a Network Module

- Network modules can be loaded via `insmod/modprobe` just like other drivers
- They request resources, probe IRQs, set up `wait_queues` and tasklets just like normal drivers
- However, there is no major/minor device number for a network interface
 - The driver inserts a data structure (`struct net_device`) into a global list of network devices

Network Driver Sample Code

(a)

`xxx_probe/module_init`

```
→ dev=alloc_etherdev(sizeof(driver_private_structure))
  ↳ alloc_etherdev(sizeofpriv, "eth%d", ether_setup)
    → dev=kmalloc(sizeof(net_device)+sizeofpriv+padding)
    → ether_setup(dev)
    → strcpy(dev->name, "eth%d")
    → return(dev)
    ...
    → netdev_boot_setup_check(dev)
    ...
    → register_netdev(dev)
      ↳ register_netdevice(dev)
```

(b)

`xxx_remove_one/module_exit`

```
→ unregister_netdev(dev)
  ↳ unregister_netdevice(dev)
→ .... ...
→ free_netdev(dev)
```

- `xxx_setup`
 - `change_mtu`
 - `set_mac_address`
 - `rebuild_header`
 - `hard_header`
 - `hard_header_cache`
 - `header_cache_update`
 - `hard_header_parse`

- Net_device
 - open
 - stop
 - hard_start_xmit
 - tx_timeout
 - watchdog_timeo
 - get_stats
 - get_wireless_stats
 - set_multicast_list
 - do_ioctl
 - init
 - uninit
 - poll
 - ethtool_ops (*this is actually an array of routines*)

Functions used :

- `alloc_netdev` – dynamic allocation of net device structure for each NIC
- `alloc_etherdev` – allocates eth% device and calls
- `ether_setup` – set up `net_device` struture fill up other `net_device` structure
- `register_netdevice` – register with linux stack
- `request_irq` – register interrupt

Design of network driver

- Different modes:
 - Polling
 - Interrupts
 - NAPI
 - Timer-interrupt
 - Interrupts+BH

Packet Flow:

- Example: Flow for receive
 - Interrupt → hardware registers etc → raise BH → return
 - BH → some actions → allocate and fills in skb buffer → netif_rx
 - Netif_rx → ingress queue → raise net_rx_action (BH)
 - Net_rx_action (BH) → netif_receive_skb → action <drop/route/etc>

Packet Flow (cont)

- Transmission:
 - Netif_start_queue
 - Netif_stop_queue
 - Netif_restart_queue
 - Netif_wake_queue → TX BH

Packet Flow (cont)

- Flow
 - Application socket → <network stack> → `dev_queue_xmit`
 - `Qdisc_restart` → `hard_start_xmit` → driver transmit function

Flow of Network Drivers

- We pass the address of an `init()` function in the `net_device` structure
 - This routine handles probing for the device and establishes the basic callbacks and flags and calls

```
void ether_setup(struct net_device) *dev);
```

to fill in some details

Initialization Example

```
/* Initialize the private device structure. */
if (dev->priv == NULL) {
    dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
    if (dev->priv == NULL)
        return -ENOMEM;
}
memset(dev->priv, 0, sizeof(struct net_local));
np = (struct net_local *)dev->priv;
spin_lock_init(&np->lock);

/* Grab the region so that no one else tries to probe our ioports. */
request_region(ioaddr, NETCARD_IO_EXTENT, cardname);

dev->open          = net_open;
dev->stop          = net_close;
dev->hard_start_xmit = net_send_packet;
dev->get_stats     = net_get_stats;
dev->set_multicast_list = &set_multicast_list;
dev->tx_timeout    = &net_tx_timeout;
dev->watchdog_timeo = MY_TX_TIMEOUT;

/* Fill in the fields of the device structure with ethernet values. */
ether_setup(dev);
```

Key Functions

- The interface is opened when `ifconfig()` activates it
 - Your `open()` method should allocate any resources, turn on the hardware and increment the module usage count
- Your `stop()` method will be called when someone performs an `ifconfig down` on your interface
 - It should undo anything done in the `open()` method

Example open() Method

```
static int net_open(struct net_device *dev) {
    struct net_local *np = (struct net_local *)dev->priv;
    int ioaddr = dev->base_addr;
    /* This is used if the interrupt line can turned off (shared).
     * See 3c503.c for an example of selecting the IRQ at config-time. */
    if (request_irq(dev->irq, &net_interrupt, 0, cardname, dev)) {
        return -EAGAIN;
    }
    /* Always allocate the DMA channel after the IRQ, and clean up on failure.
     */
    if (request_dma(dev->dma, cardname)) {
        free_irq(dev->irq, dev);
        return -EAGAIN;
    }

    /* Reset the hardware here. Don't forget to set the station address. */
    chipset_init(dev, 1);
    outb(0x00, ioaddr);
    np->open_time = jiffies;

    /* We are now ready to accept transmit requests from
     * the queueing layer of the networking. */
    netif_start_queue(dev);
    return 0;
}
```

Key Functions #2

- When a packet is ready to transmit, your `hard_start_xmit()` method will be called
 - The passed socket buffer (`struct sk_buff *`) will have a fully formatted packet with hardware headers ready for DMA
- The `hard_header()` and `rebuild_header()` methods handle creating the hardware headers on reception and transmission respectively
 - You write these to be appropriate to the media being used

Example hard_start_xmit() #1 of 2

```
static int net_send_packet(struct sk_buff *skb, struct
    net_device *dev)
{
    struct net_local *np = (struct net_local *)dev->priv;
    int ioaddr = dev->base_addr;
    short length = ETH_ZLEN < skb->len ? skb->len : ETH_ZLEN;
    unsigned char *buf = skb->data;

    /* If some error occurs while trying to transmit this
     * packet, you should return '1' from this function.
     * In such a case you may not do anything to the
     * SKB, it is still owned by the network queueing
     * layer when an error is returned. This means you
     * may not modify any SKB fields, you may not free
     * the SKB, etc.
    */
}
```

Example start_hard_xmit() #2 of 2

```
/* This is the case for older hardware which takes
 * a single transmit buffer at a time, and it is
 * just written to the device via PIO.
 *
 * No spin locking is needed since there is no TX complete
 * event. If by chance your card does have a TX complete
 * hardware IRQ then you may need to utilize np->lock here.
 */
hardware_send_packet(ioaddr, buf, length);
np->stats.tx_bytes += skb->len;

dev->trans_start = jiffies;

/* You might need to clean up and record Tx statistics here. */
if (inw(ioaddr) == /*RU*/81)
    np->stats.tx_aborted_errors++;
dev_kfree_skb (skb);
return 0;
}
```

Starting/Stopping the Stack

- Once set up, we issue the `netif_start_queue(struct net_device *dev)` command to inform the stack that we're ready to start transmitting packets
- We will use the `netif_stop_queue(dev)` command to stop the stack when we release the device
 - Or, if the transmit queue becomes full
 - Our transmit interrupt will need to restart the queue via `netif_wake_queue(dev)` as soon as there's room for more `sk_buffs`

Key Functions #3

- Your `tx_timeout()` method will be called if the packet transmission fails to complete in a reasonable period of time
 - The assumption is that an interrupt was missed or the interface has locked up
 - You will need to reset the interface and restart the transmission

Example tx_timeout()

```
static void net_tx_timeout(struct net_device *dev) {
    struct net_local *np = (struct net_local *)dev->priv;
    printk(KERN_WARNING "%s: transmit timed out, %s?\n", dev->name,
           tx_done(dev) ? "IRQ conflict" : "network cable problem");

    /* Try to restart the adaptor. */
    chipset_init(dev, 1);
    np->stats.tx_errors++;
    /* If we have space available to accept new transmit
     * requests, wake up the queueing layer. This would
     * be the case if the chipset_init() call above just
     * flushes out the tx queue and empties it.
     *
     * If instead, the tx queue is retained then the
     * netif_wake_queue() call should be placed in the
     * TX completion interrupt handler of the driver instead
     * of here.
     */
    if (!tx_full(dev))
        netif_wake_queue(dev);
}
```

Key Functions #4

- The `get_stats()` method is invoked when `ifconfig` or `netstat -i` is called
 - You need to return statistics on the interface via the `net_device` structure
- There are a number of other calls that handle custom ioctls, changing the MAC address, MTU, etc. These are not strictly required if your hardware is simple enough

Example get_stats()

```
/*
 * Get the current statistics.
 * This may be called with the card open or closed.
 */
static struct net_device_stats *net_get_stats(struct net_device
    *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    short ioaddr = dev->base_addr;

    /* Update the statistics from the device registers. */
    lp->stats.rx_missed_errors = inw(ioaddr+1);
    return &lp->stats;
}
```

OK, What about Receiving?

- So far, we've only addressed transmitting packets
- The reception of packets is typically connected to the ISR for the device
 - We need to check the interrupt status mask for the device to see if a packet has arrived

Example ISR

```
static void net_interrupt(int irq, void *dev_id, struct pt_regs * regs) {
    struct net_device *dev = dev_id;
    struct net_local *np;
    int ioaddr, status;

    ioaddr = dev->base_addr;
    np = (struct net_local *)dev->priv;

    status = inw(ioaddr + 0);

    if (status & RX_INTR) {
        /* Got a packet(s). */
        net_rx(dev);
    }

    if (status & COUNTERS_INTR) {
        /* Increment the appropriate 'localstats' field. */
        np->stats.tx_window_errors++;
    }
}
```

Inserting into the Stack

- Once we have received the packet (potentially using a tasklet to handle the data movement) we need to update the statistics for the interface and insert the sk_buff into the stack
- This is accomplished using:
`void netif_rx(skb);`



Example Receive #1 of 2

```
static void net_rx(struct net_device *dev) {
    struct net_local *lp = (struct net_local *)dev->priv;
    int ioaddr = dev->base_addr;
    int boguscount = 10;

    do {
        int status = inw(ioaddr);
        int pkt_len = inw(ioaddr);
        if (pkt_len == 0)           /* Read all the frames? */
            break;                /* Done for now */
        if (status & 0x40) { /* There was an error. */
            lp->stats.rx_errors++;
            if (status & 0x20) lp->stats.rx_frame_errors++;
            if (status & 0x10) lp->stats.rx_over_errors++;
            if (status & 0x08) lp->stats.rx_crc_errors++;
            if (status & 0x04) lp->stats.rx_fifo_errors++;
        } else {
            /* Malloc up new buffer. */
            struct sk_buff *skb;
            lp->stats.rx_bytes+=pkt_len;
            skb = dev_alloc_skb(pkt_len);
```

Example Receive #2 of 2

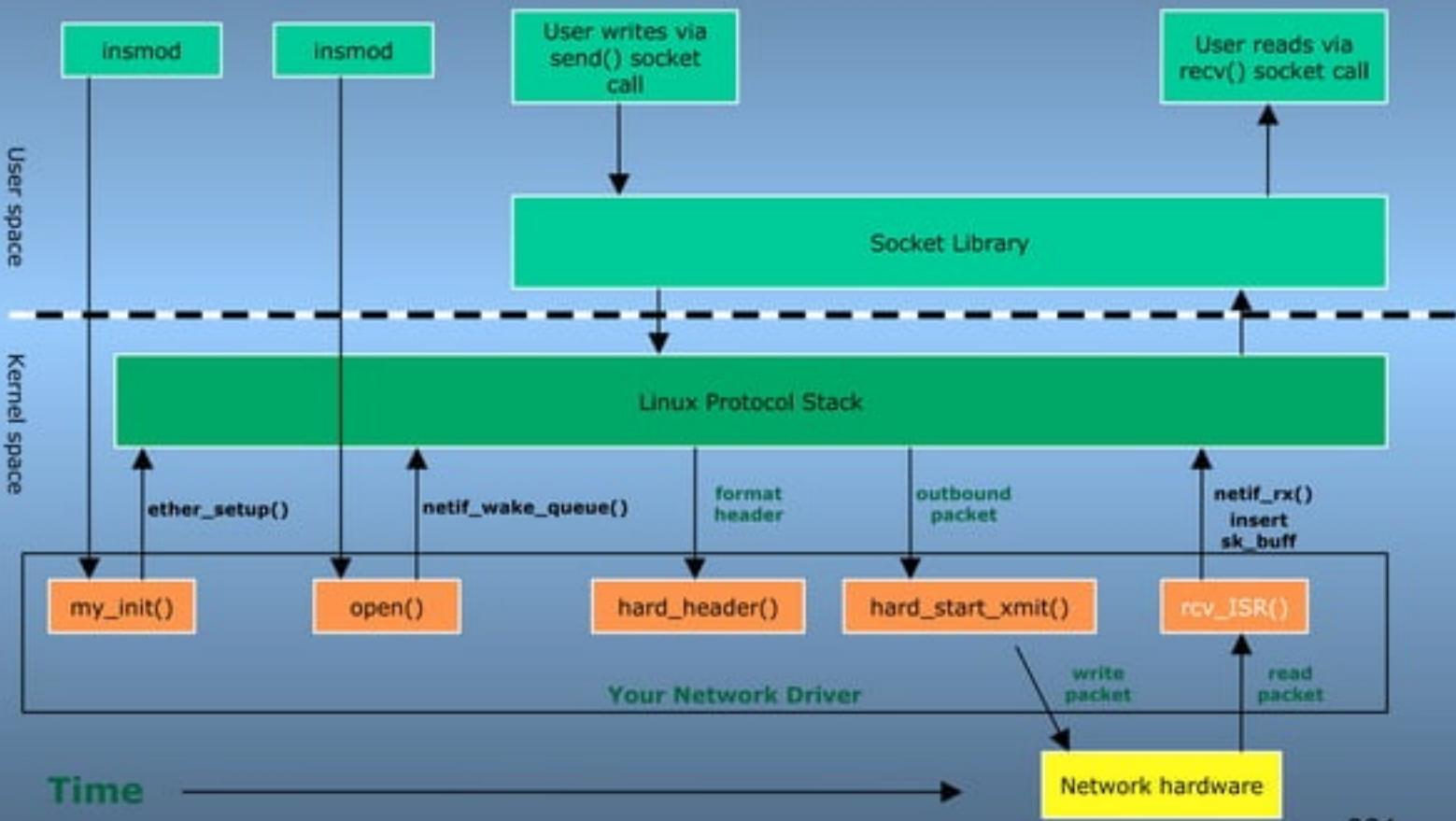
```
    if (skb == NULL) {
        printk(KERN_NOTICE "%s: Memory squeeze, dropping
packet.\n",
                           dev->name);
        lp->stats.rx_dropped++;
        break;
    }
    skb->dev = dev;

    /* 'skb->data' points to the start of sk_buff data area.
 */
    memcpy(skb_put(skb,pkt_len), (void*)dev->rmem_start,
pkt_len);
        /* or */
    insw(ioaddr, skb->data, (pkt_len + 1) >> 1);

    netif_rx(skb);
    dev->last_rx = jiffies;
    lp->stats.rx_packets++;
    lp->stats.rx_bytes += pkt_len;
}
} while (--boguscount);

return;
```

Network Data Flow



Building Network Drivers

- Start from <linux>/drivers/net/isa-skeleton.c or pci-skeleton.c
- Create `init()` first, then add support for formatting the MAC header
 - Add `open()` and `stop()` facilities
- Add support for `start_hard_xmit()` and `tx_timeout()`
- Next create a `get_stats()` and a receive ISR
 - You now have a minimal network driver
 - Enhance it with DMAs, ring buffers, etc.

Network sub-system components:

- Skb_buff
- Net_device
- NAPI
- Interrupt
- Routing sub-system
- Routing table
- Routing cache
- Policy routing
- Routing table look-up algorithm
- Receiving packet
- Forwarding
- Sending a packet
- Netfilter
- Neighbouring subsystem
- IpSec
- Etc..

Transmission Path:

- L5 (sockets and files)
 - Write
 - Sendto
 - Sendmsg
 - `__sock_sendmsg`

- L4 (transport layer)
 - Tcp_sendmsg
 - Allocate skb buffer
 - Copy data from user space to skb
 - Tcp queue is activated
 - Tcp_transmit_skb – builds tcp header

- L3 (Network layer)
 - `Ip_queue_xmit` (does routing , creates IP header)
 - Routing decision results in `dst_entry` object
 - `Skb_buff` to `ip_ouput`
 - `Ip_output` does post-routing filtering

- L2 (link layer)
 - Scheduling the packet to be sent out
 - Qdisc (queueing discipline)
 - `Dev_queue_xmit` to put skb on device queue
 - `Dev_hard_xmit`
 - `Netif_schedule` (which causes TX bottom half to be called)
 - Finally `hard-start_xmit` function of the device is called

Reception

- Network stack future work direction:

Some domain Linux community works

- Real-time performance
- Memory optimization like SLUB allocator
- Process scheduling
- Power management
- Linux test suite
- Different new platform porting
- New device support
- Bugs fixation
- User-level driver support
- Cross-compiler
- Other improvements based on product e.g support of multiple transmit buffer in network device
- Real-time/native network stack
- Etc.....

References

- <http://kernelnewbies.org/>
- <http://lwn.net/>
- http://elinux.org/Main_Page
- <http://linux.derkeiler.com/>
- <http://groups.google.com/group/linux.kernel>
- <http://www.arm.linux.org.uk/>
- <http://www.linux-foundation.org/>
- <http://celinuxforum.org/about>
- <http://www.linuxhq.com/>
- <http://www.kernel.org>
- <http://www.linux.org>
- <http://www.gelato.unsw.edu.au/lxr/source/?a=arm>
- **<http://www.linux-foundation.org/en/Net>**
- <http://www.google.com/>

Books

- Understanding the Linux Kernel, Third Edition , By **Daniel P. Bovet, Marco Cesati**
- Linux kernel Development , By Robert Love
- Linux Device Driver , Third Edition , By Rubini
 - <http://lwn.net/Kernel/LDD3/>
- Building Embedded Linux Systems , By Yaghmour
- Linux Network Internals , By Benvenuti
- Linux TCP/IP Stack , By Thomas Herbert

THANK YOU!!!!!!

MUKUL BHARDWAJ