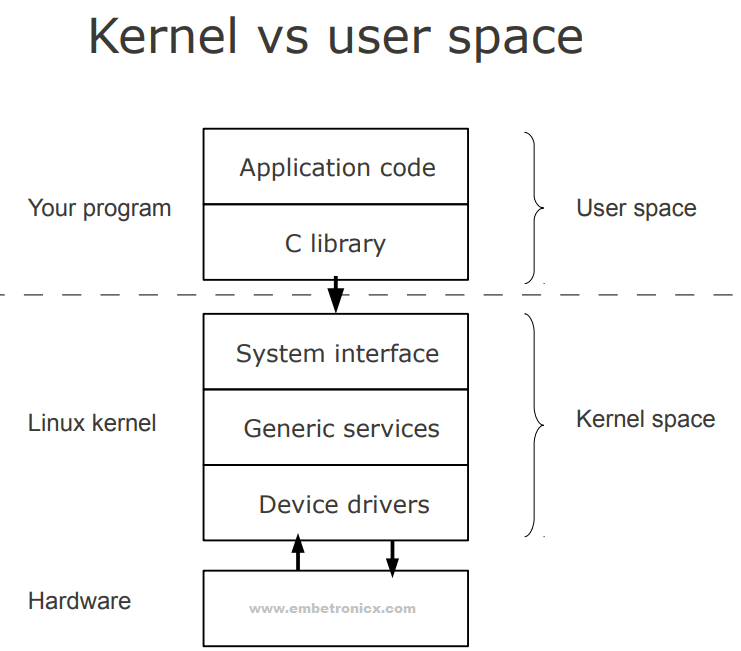
Day 1 :

### INTRODUCTION:

Linux is primarily divided into **User Space** & **Kernel Space**. These two components interact through a System Call Interface – which are predefined and matured interface to Linux Kernel for Userspace applications. The below image will give you a basic understanding.



All user space header filea : /usr/include.

Kernel header -> /lib/modules/5.4.0-42-generic/build/include/linux/

In make file we will define as below to include header file

KDIR = /lib/modules/$(shell uname -r)/build

## Linux Kernel Modules

1. Static – base kernel
2. Dynamic - loadable kernel module(LKM)

* Advantages of LKM: LKMs are very flexible, saves time and also helps in keeping our base kernel error-free.

## Differences Between Kernel Modules and User Programs

* **Kernel modules have separate address space.**
* **Kernel modules have higher execution privileges.**
* **Kernel modules do not execute sequentially.**
* **Kernel modules use different header files.**

## Types

In the traditional classification, there are three kinds of the device:

* Character device
* Block device
* Network device

## Module Information:

## License: GPL, or the GNU General Public License, is an open-source license meant for software. If your software is licensed under the terms of the GPL, it is free

MODULE\_LICENSE**(**"GPL"**)**;

MODULE\_LICENSE**(**"GPL v2"**)**;

MODULE\_LICENSE**(**"Dual BSD/GPL"**)**;

you need to include the Linux/module.h header file.

## Author: Macro we can mention that who is writing this driver or module

MODULE\_AUTHOR**(**"Author"**)**;

## Module Description: Macro we can give a description of the module or driver

MODULE\_DESCRIPTION**(**"A sample driver"**)**;

## Module Version: Macro we can give the version of the module or driver

MODULE\_VERSION("2:1.0");

## Simple Kernel Driver:

Similar to c main function , In linux we are using two separate functions for starting and ending.

1. Init function->This is the function that will execute first when the driver is loaded into the kernel.
2. Exit function->execute last when the driver is unloaded from the kernel

The \_\_init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers(Static module),but not loadable modules

 \_\_initdata which works similarly to \_\_init.

The \_\_exit macro causes the omission of the function when the module is built into the kernel, and like \_\_exit, has no effect for loadable module.

## Printk(): Print messages in the console, lets you classify messages according to their severity by associating different log levels, or priorities, with the messages.

**KERN\_EMERG:**Used for emergency messages, usually those that precede a crash.

**KERN\_ALERT:**A situation requiring immediate action.

**KERN\_CRIT:**Critical conditions, often related to serious hardware or software failures.

**KERN\_ERR:**Used to report error conditions; device drivers often use KERN\_ERR to report hardware difficulties.

**KERN\_WARNING:**Warnings about problematic situations that do not, in themselves, create serious problems with the system.

**KERN\_NOTICE:**Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

**KERN\_INFO:**Informational messages. Many drivers print information about the hardware they find at startup time at this level.

**KERN\_DEBUG:**Used for debugging messages.

printk**(**KERN\_INFO "Welcome To My driver"**)**;

## Difference between printf and printk

* **Printk()**is a kernel-level function, which has the ability to print out to different log levels as defined in. We can see the prints using **dmesg**command.
* **printf()** will always print to a file descriptor – STD\_OUT. We can see the prints in the STD\_OUT console.

Example : First\_device\_driver

**modinfo First\_device\_driver.ko ->** Modinfo will print all the information about our driver

filename: /mnt/hgfs/johny/kishore\_material/Driver\_code/First\_device\_driver/First\_device\_driver.ko

version: 1:0.0

description: A sample driver

author: johny

license: Dual BSD/GPL

license: GPL v2

license: GPL

srcversion: 669ECA0A70C09F730E6E31C

depends:

retpoline: Y

name: First\_device\_driver

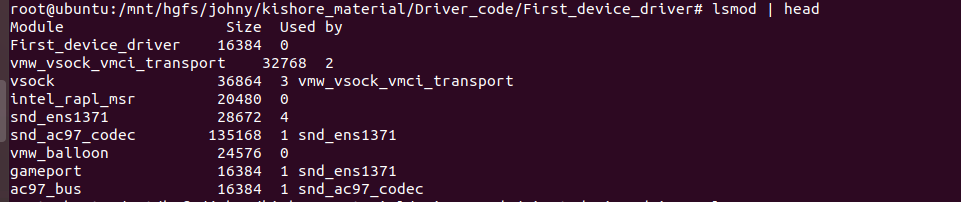
vermagic: 5.4.0-42-generic SMP mod\_unload

# Loading and Unloading the Device driver

**Insmod :**To load a Kernel Module, use the **insmod**command with root privileges.



lsmod used to see the modules were inserted



Or



## Unloading

use the **rmmod**command with root privileges.



## Passing Arguments to Device Driver:

Header file : **linux/moduleparam.h.**

## Module Parameters Macros

* **module\_param()**
* **module\_param\_array()**
* **module\_param\_cb()**

There are several types of permissions:

* S\_IWUSR
* S\_IRUSR
* S\_IXUSR
* S\_IRGRP
* S\_IWGRP
* S\_IXGRP

In this S\_I is a common header.  
R = read ,W =write ,X= Execute.  
USR =user ,GRP =Group  
Using OR ‘|’ (or operation) we can set multiple permissions at a time.

**module\_param(name, type, perm);**

module\_param() macro creates the sub-directory under **/sys/module**

This will create a sysfs entry. (**/sys/module/hello\_world\_module/parameters/valueETX**)

* bool
* invbool->A boolean (true or false)
* charp->A char pointer value.
* int
* long
* short
* uint
* ulong
* ushort--🡪 integer of various length.

**module\_param\_array()->**This macro is used to send the array as an argument

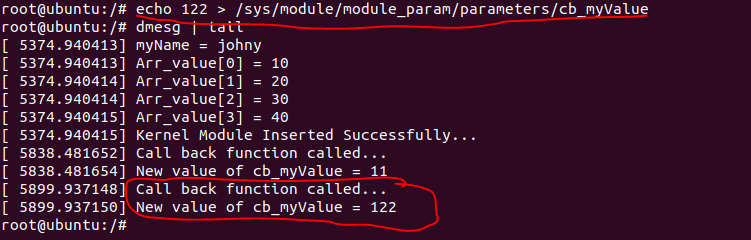
**module\_param\_array(name,type,num,perm);**

## module\_param\_cb()-> used to register the callback

Example : module\_param.c



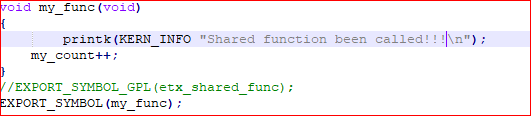
to check **module\_param\_cb()** is whether calling that handler function or not.



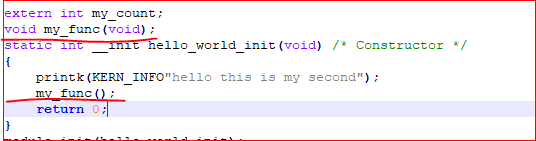
## EXPORT\_SYMBOL()

EXPORT\_SYMBOL() is a macro the Linux kernel headers define. It has not much in common with extern. It tells the kbuild mechanism that the symbol referred to should be part of the global list of kernel symbols. That, in turn allows kernel modules to access them. Code that is built into the kernel itself (as opposed to a module) can, of course, access any non-static symbol via an extern declaration, in accordance with regular C. The EXPORT\_SYMBOL() mechanism allows us to export a symbol for use by loadable modules as well. An interesting thing is that a symbol thus exported by one module becomes accessible to another module that may depend on it!

Step 1: in kernel Module Use EXPORT\_SYMBOL macro to extern the function to other module



Step 2: In another module call the export function.



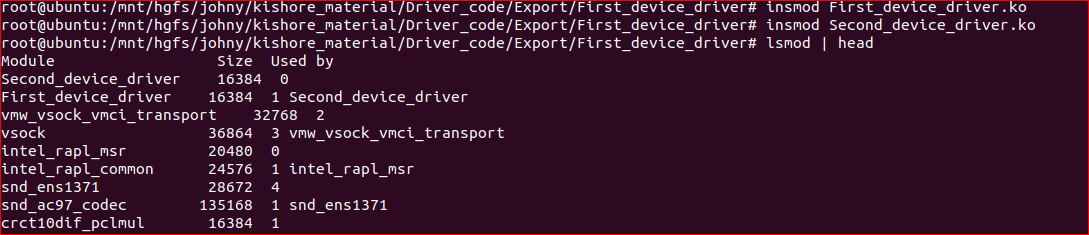
Step 3 : Include the module in make file and compile



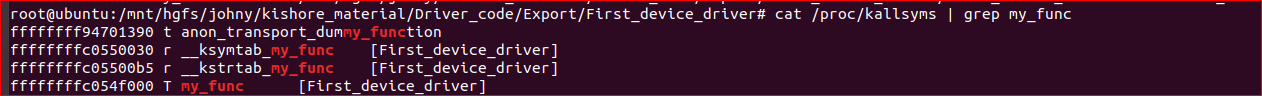
Step4: After compile to view global function and variable the use below command



Step5 : insert the module, once insert both modules .(insert mode should be in order )

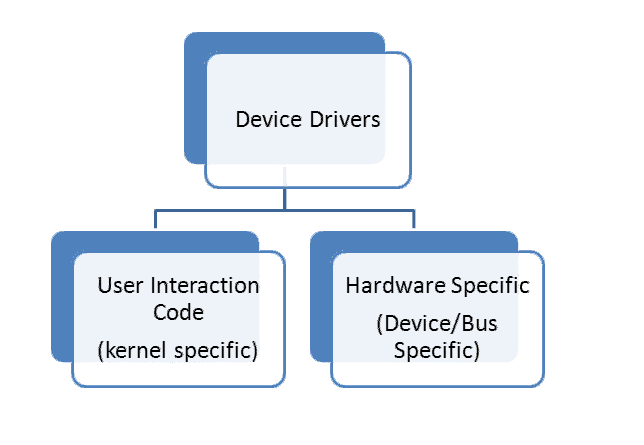


Step 6:To check  shared function and variable become the part of kernel’s symbol table or not.



## Day-2

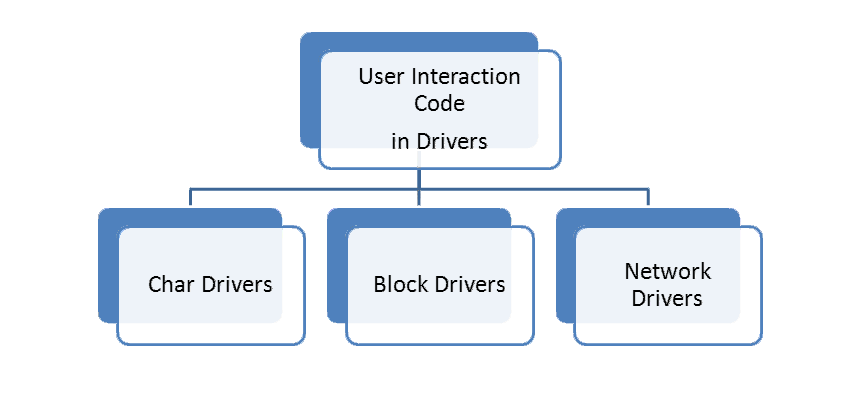
## Device Driver:



* Driver is service, it execute a context of application. , it is kernel specific.
* Hardware access path , we will implement instruction to hardware , developer should now hoe the hardware is connected , what protocol is used etc.

In DD, they are 3 diff approaches of user interaction.

1. Char
2. Block->storages drivers,
3. Network -> wifi , communication drivers etc.



## Character model:

1. Data transfer is sync.
2. Drivers are register to VFS

Steps:

1. Create a device file

mknode command is to create device file , device files are placed at /dev folder .

It’s not mandatory to use /dev folder to create device file.

Device files are category are two type is

* + Char - c
  + Block – b

Each device file is identify with unique id

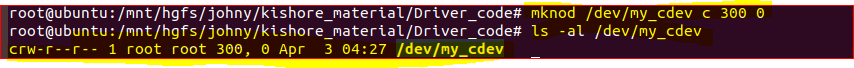
Cat /proc/devices -> shows current list of devices in use , we need to pick an id and assigned to device file which is not in use(unique id) .

1. Implement driver operation (read ,write ) , register that operation with VFS layer .
2. Insert the driver with kernel
3. Write an application and test the driver routine

Step1: mknode /dev/my\_cdev c 300 0

C= Char / b = block

Device file major no. minor no.



Step2:

## Implement the driver:

We need to register set of routines with a valid operation.

static struct file\_operations char\_dev\_fops = {

.owner = THIS\_MODULE, // the current module is owner of the module

.write = char\_dev\_write,

.read = char\_dev\_read,

.open = char\_dev\_open,

.release = char\_dev\_release

};

char\_dev\_write, char\_dev\_read, char\_dev\_open, char\_dev\_release-> we need to register those function with vfs layer .

Since VFS is responsible for switching application request in to driver , so we need to register these address in to VFS , VFS provide function pointer with file operations structure.

/usr/src/linux-headers-5.4.0-42-generic/include/linux/fs.h

#include <linux/fs.h>

The “filesystem” header is the header required for writing device drivers. Many important functions and data structures are declared in here.

Driver should there function with VFS as instance of file operation.We need to register our driver with VFS .

* dev\_t mydev; we need to create an instance of mydev

mydev is 32 bit value (20 bits is major , 12 bits is minor)

dev\_t MKDEV(unsigned int major, unsigned int minor);-> Macro that builds a dev\_t data item from the major and minor numbers.

mydev = MKDEV(MAJORNO,MINORNO);

* register driver :

register\_chrdev\_region(mydev,count,CHAR\_DEV\_NAME);

register\_chrdev\_region should be used when the desired major number is known in advance;

mydev = This major and minor no. is reserve for this driver.

name = name of the driver to reserve major and minor no.

int unregister\_chrdev(unsigned int major, const char \*name); Function that undoes a registration made with register\_chrdev. Both major and the name string must contain the same values that were used to register the driver.

#include <linux/cdev.h>

struct cdev \*cdev\_alloc(void);

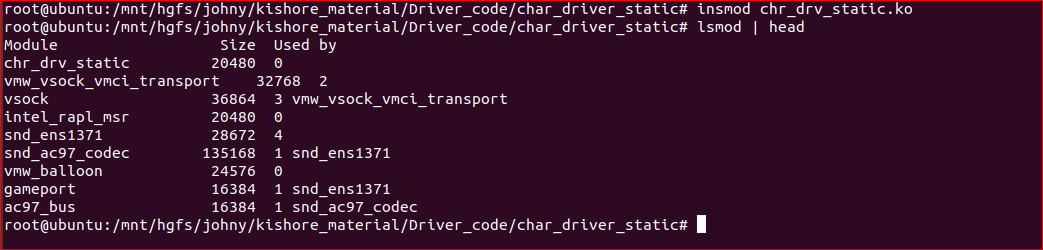
void cdev\_init(struct cdev \*dev, struct file\_operations \*fops);

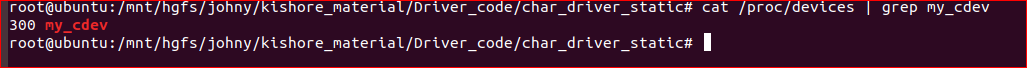
int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count);

void cdev\_del(struct cdev \*dev);

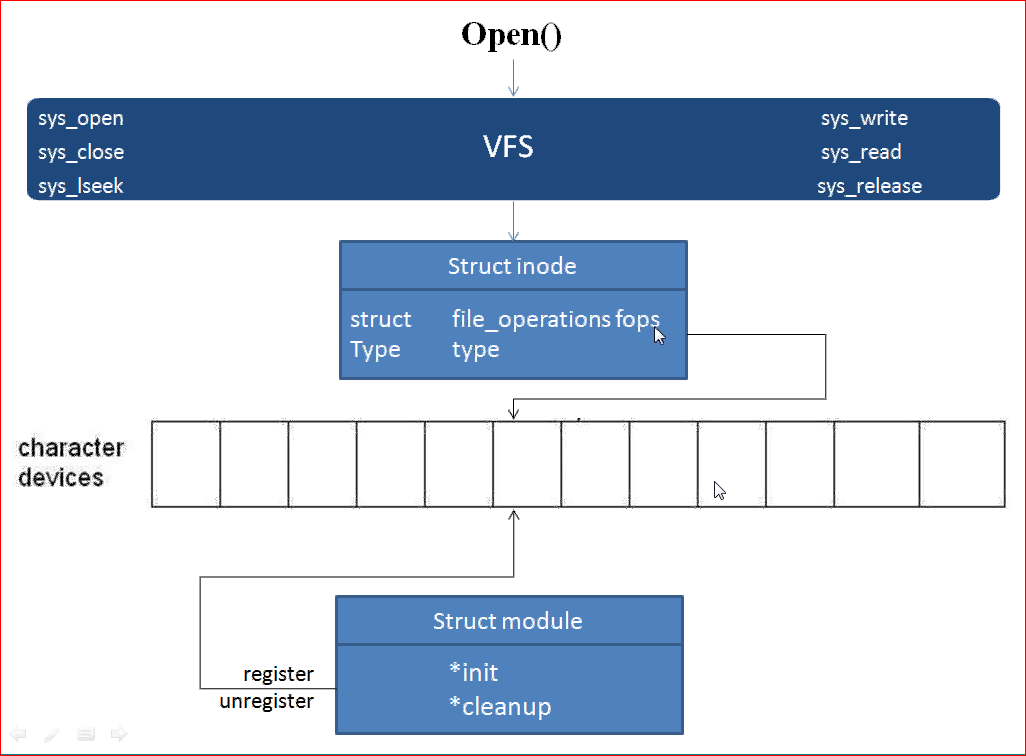
Functions for the management of cdev structures, which represent char devices within the kernel.

Step3 : insert the module :





Step 4 : User space application



* Vfs maintain 2 data structure char device list and block device list.
* It store all cdev objects in device list , to hold drivers objects .
* Each drivers object to specific offset, that offset is major no.
* Major no is index of char device list where driver operation are copied.
* /proc/device -> show the list status, which ever offset/node is not use that will not show in this list.
* Step 1: create a inode , at kernel level which was pointing to device list , in device list we will mention which location to point by using major number .
* Step 2: Inserted the driver, register its operation instance in specific major no device list is pointing.

With this approaches had some problems like if we port driver to other machine , current device 300 offset is free, it may be in use in new device, to avoid this it will always better to acquire major no. dynamically will be approach .

## Dynamic Approach:

Step1: write a driver in initialization routine probe the particular list , which ever object is free , dynamically acquire the offset and push the device operation there .

Step2: find which slot is free then create a file .

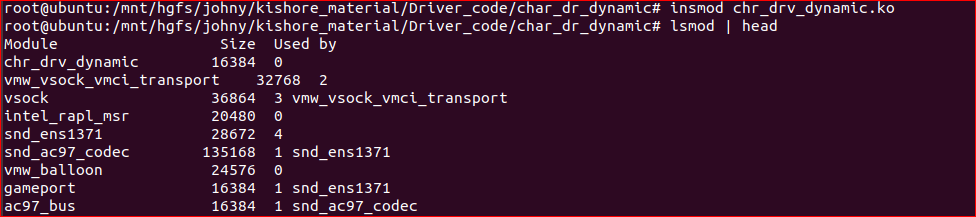
Reverse operation of above diagram.

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name)

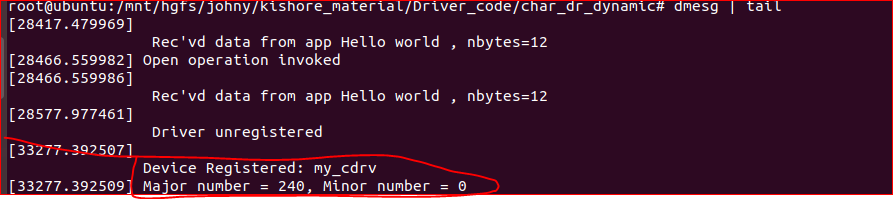
* This Api will find the free device list and return the major no. to that driver.
* Kernel allows n no. of device nodes share a common major number.
* Minor no of identification of inode / files, pointing to that particular device offset of device table. It is 12 bit in size.
* Count = how many device node that can share this particular driver.
* Name = name of the driver.
* printk(KERN\_INFO"Major =%d Minor = %d ",MAJOR(dev),MINOR(dev));

Example : char\_dr\_dynamic

Insert :



Dmesg:



Creating device node :



## Automatically Creating/Deleting Device File:

Automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically.

1. Include the header file **linux/device.h** and **linux/kdev\_t.h**
2. Create the struct Class
3. Create Device with the class which is created by the above step

## Create the class

This will create the struct class for our device driver. It will create a structure under**/sys/class/**.

**struct class \* class\_create (struct module \*owner, const char \*name);**

owner – pointer to the module that is to “own” this struct class

name – pointer to a string for the name of this class

This is used to create a struct class pointer that can then be used in calls to **class\_device\_create**.

**Note,** the pointer created here is to be destroyed when finished by making a call to **class\_destroy**.

**void class\_destroy (struct class \* cls);**

## ProcFs

## Day-3

## Process

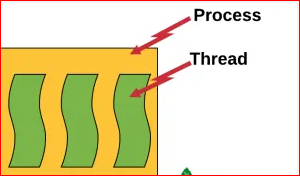
An executing instance of a program is called a process. Some operating systems use the term ‘task‘ to refer to a program that is being executed.  **Process**is a heavyweight process.

## Threads

A *thread* is an independent flow of control that operates within the same address space.

Threads, also known as lightweight processes.

Some of the advantages of the thread, is that since all the threads within the processes share the same address space, the communication between the threads is far easier and less time consuming as compared to processes.



## Thread Management

* A thread is a sequence of instructions.
* CPU can handle one instruction at a time.
* To switch between instructions on parallel threads, the execution state needs to be saved.
* Execution state in its simplest form is a program counter and CPU registers.
* The program counter tells us what instruction to execute next.
* CPU registers hold execution arguments, for example, addition operands.
* This alternation between threads requires management.
* Management includes saving state, restoring state, deciding what thread to pick next.

## Types of Thread

There are two types of threads.

1. User Level Thread
2. Kernel Level Thread

## User Level Thread

Everything is maintained by the user thread library. That thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.

## Kernel Level Thread

Kernel level threads are managed by the OS, thread operations are implemented in the kernel code.

## Kernel Thread Management Functions

* Create Kernel Thread
* Start Kernel Thread
* Stop Kernel Thread
* Other functions in Kernel Thread

**Create Kernel Thread:**

**struct task\_struct \* kthread\_create (int (\* threadfn(void \*data),**

                     void \*data, const char namefmt[], ...);

Where,

threadfn – the function to run until signal pending(current).

data – data ptr for threadfn.

namefmt[] – printf-style name for the thread.

**Example: char\_thread = kthread\_create(my\_thread\_fun,NULL,"my char thread");**

* **Start Kernel Thread**:

This is used to Wake up a specific process.

**int wake\_up\_process (struct task\_struct \* p);**

Where,

**p** – The process to be woken up.

Attempt to wake up the nominated process and move it to the set of runnable processes.

It **returns 1** if the process was woken up,**0** if it was already running.

**Example : wake\_up\_process(char\_thread);**

* **Stop Kernel Thread**

**int kthread\_stop ( struct task\_struct \*k);**

**k** – thread created by kthread\_create.

**Exapmple: kthread\_stop(char\_thread);**

* **kthread\_bind**: This is used to bind a just-created kthread to a cpu

**void kthread\_bind (struct task\_struct \*k, unsigned int cpu);**

Where,

k – thread created by kthread\_create.

cpu – cpu (might not be online, must be possible) for k to run on.

## kthread\_run

This is used to create and wake a thread.

**kthread\_run (threadfn, data, namefmt, ...);**

Where,

threadfn – the function to run until signal\_pending(current).

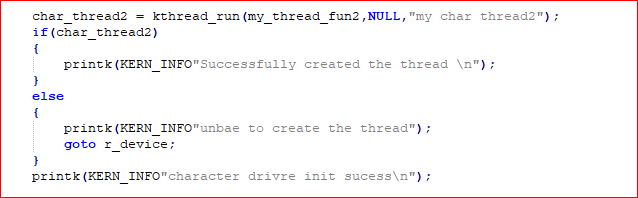
data – data ptr for threadfn.

namefmt – printf-style name for the thread.

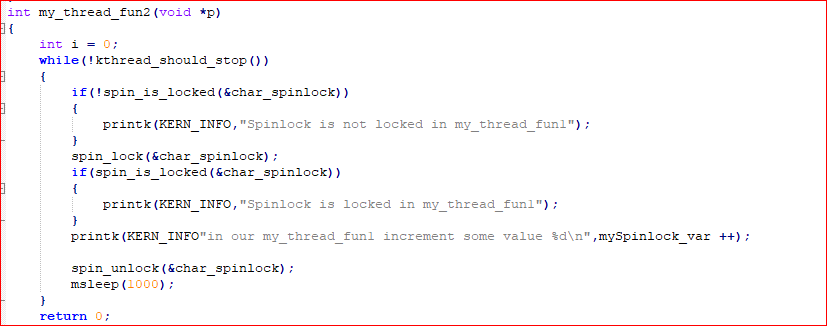
It **returns** the **kthread** or ERR\_PTR(-ENOMEM).

## Implementation

* 1. Create thread :



* 1. Thread handle:



* 1. Stop:



Example: kthread folder .

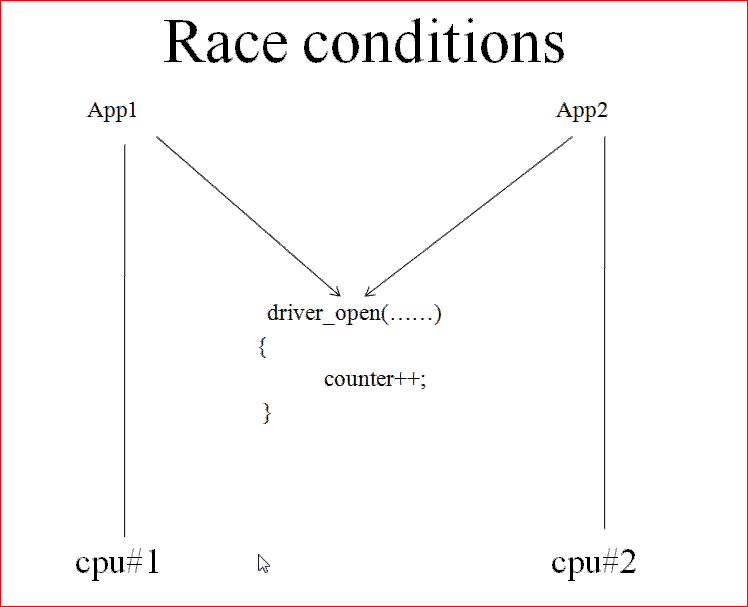
## Kernel\_synchronization

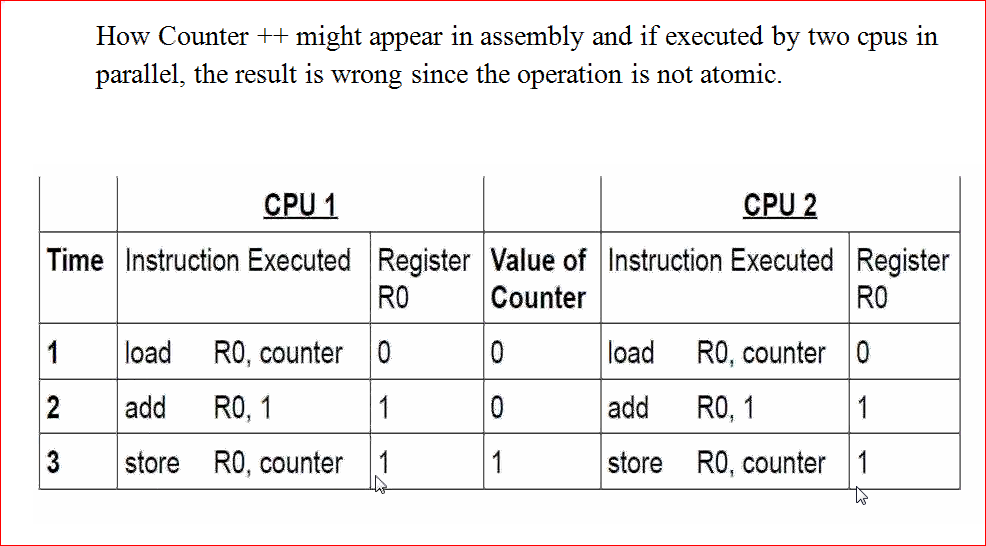
Concurrency = things happening at the same time.

Source of concurrency in linux

* 1. Miltiple processors
  2. Hardware interrupts
  3. Software interrupts
  4. Kernel timers
  5. Work queues
  6. Kernel preemption
* Failure to manage concurrency leads to bug
* Difficult to track down
* Devastating when they happen

Race conditions:

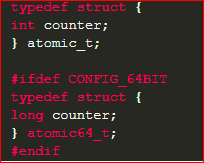




## atomic variables

The read, write and arithmetic operations on the atomic variables will be done in one instruction without interrupt.

Atomic variable are :



## Types of atomic variables

Two different atomic variables are there.

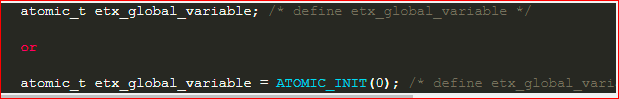
* Atomic variables who operates on Integers
* Atomic variables who operates on Individual Bits

## Atomic Integer Operations

## Header file :

**#include <asm/atomic.h>**.

## Creating atomic variables



## 

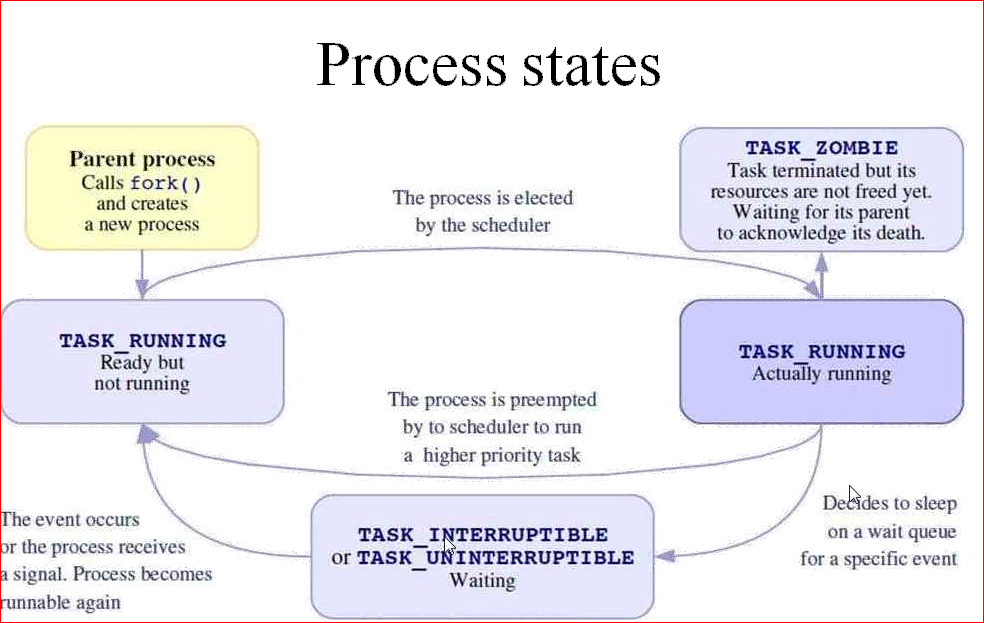
## Locking technique in kernel :

## 

* In wait lock we can use sleep call but in polling lock not wait is involved.

## Linux process has state;

* Create state
* Ready state-> app is waiting in run queue.
* Waiting state -🡪a. Task interruptible wait state -> app will receive a signal and respond
* b. Task uninterruptable wait state.-> app. will not respond to signal during its wait time
* Running state
* Zombie state /exit state -> – here, a process is dead, it has been halted but it’s still has an entry in the process table.



## Wait lock :

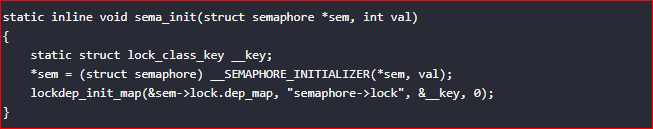
* If lock is succeed, the caller enter the critical section , the that particular cpu will suspend the cpu scheduling i.e., disable preemption, until lock is release .
* If lock acquired is failed, then the locking functions push context in to wait state.

## Semaphores:

Header file :

 include *<asm/semaphore.h>*.

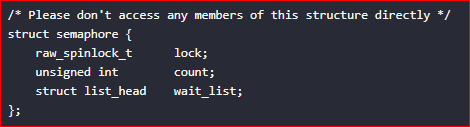
## Initialize Semaphore:

* Static application:
  + 
* Dynamic application:
  + 

## Difference b/w Static and Dynamic

* Is that the value of val is different.
* The statically applied semaphore val=1, that is, the semaphore can only be used by one thread, and the waiting thread task enters the sleep state.
* The dynamically applied semaphore, val can be set by the user. If val> 1, multiple threads can use this semaphore at the same time until val <= 0. Other thread tasks using this semaphore enter the sleep waiting state.

## How to obtain semaphore



void down(struct semaphore \*sem) :down()---->\_\_down()---->\_\_down\_common()

* The simplest kind of down, if the task gets the semaphore, put the task into the sleep state until the semaphore is released, in fact, set the task scheduling state to: TASK\_UNINTERRUPTIBLE
* the sem->count variable will be checked first, if >0 means the lock is obtained, and directly return, the program will continue to run down and enter the critical section;
* if <=0, it means that the Get the lock, call \_\_down() to queue and sleep;

int down\_interruptible(struct semaphore \*sem):down\_interruptible()---->\_\_down()---->\_\_down\_common()

* down\_interruptible is a variant of down. The difference is that the task that fails to get the semaphore can enter the interruptible sleep state. Set the scheduling state of this task to: TASK\_INTERRUPTIBLE

int down\_killable(struct semaphore \*sem) :down\_killable()---->\_\_down\_killable()---->\_\_down\_common()

* down\_killable is also a variant of down.If the task that does not get the semaphore enters the dormant state and sets the process state to: TASK\_KILLABLE, it can be interrupted by a fatal signal.

int down\_trylock(struct semaphore \*sem):down\_trylock()---->\_\_down\_trylock()---->\_\_down\_common()

* down\_trylock is actually to directly determine whether the value of the semaphore member variable count of the structure is <0.

int down\_timeout(struct semaphore \*sem, long timeout) down\_timeout()---->\_\_down\_timeout()---->\_\_down\_common()

* Down\_timeout is another variant of down. It is a task that fails to obtain the semaphore, enters the sleep time, and sets the process status to TASK\_UNINTERRUPTIBLE.After the timeout, the semaphore is not obtained, and an error is reported.
* Up() function , first check sem->wait\_list, it means that there is no task that needs to be awakened, and return directly;
* otherwise, call \_\_up(), delete this task from sem->wait\_list, and then call wake\_up\_process(waiter->task ), wake up this task;

## Example :

Semaphore folder

## MUTEX

* A mutex is a mutual exclusion lock. Only one thread can hold the lock.
* A mutex can be used to prevent the simultaneous execution of a block of code by multiple threads that are running in a single or multiple processes.
* Mutex is used as a synchronization primitive in situations where a resource has to be shared by multiple threads simultaneously.
* A mutex has ownership. The thread which locks a Mutex must also unlock it.

## Mutex in Linux Kernel

Header file :

**#Include <linux/mutex.h>**

Initializing a mutex statically:

DEFINE\_MUTEX(name)

Initializing a mutex dynamically:

Void mutex\_init(struct mutex \*lock)

API’s:

**void mutex\_lock(struct mutex \*lock);**

* Tries to lock the mutex , sleep otherwise

**Int mutex\_lock\_killable(struct mutex \*lock);**

* Same, but can be interruptable by fatel(SIGKILL) signal.If interrupted , return a non zero value and doesn’t hold the lock .

**int mutex\_lock\_interruptible(struct mutex \*lock);**

* Same , but interruptable with any signal.

**int mutex\_trylock(struct mutex \*lock);**

* Never waits, returns a non zero value if the mutex is not available.

**int mutex\_is\_locked(struct mutex \*lock);**

* just tell whether the mutex is lock or not

**void mutex\_unlock(struct mutex \*lock);**

* Release the lock .

## Example : Mutex folder .

**POLL Based Locking:**

**SpinLock:**

* If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock.

**Header file :**

#include <linux/spinlock.h>

**Initilization :**

* DEFINE\_SPINLOCK(char\_spinlock); //static method
* spinlock\_t char\_spinlock; // Dynamic method
* spin\_lock\_init(&char\_spinlock);//Spinlock init

**Locking between User context:**

**spin\_lock(spinlock\_t \*lock):**

If you share data with user context (between Kernel Threads)

**spin\_trylock(spinlock\_t \*lock):**

Locks the spinlock if it is not already locked. If unable to obtain the lock it exits with an error and do not spin. It returns non-zero if it obtains the lock otherwise returns zero

**spin\_is\_locked(spinlock\_t \*lock)**

This is used to check whether the lock is available or not. It **returns** non-zero if the lock is currently acquired. otherwise returns zero.

**spin\_unlock(spinlock\_t \*lock)**

It does the reverse of the lock. It will unlock which is locked by the above call.



**Locking between User context and Bottom Halves):**

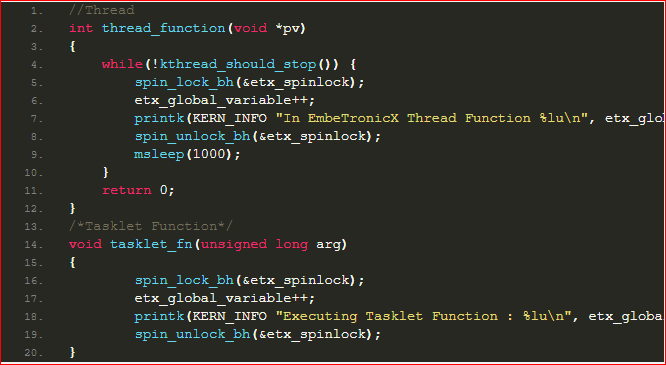
If you share data with a [bottom half](https://embetronicx.com/tag/bottom-half/) and user context (like Kernel Thread), then this approach will be useful.

**spin\_lock\_bh(spinlock\_t \*lock)**

It disables soft interrupts on that CPU, then grabs the lock. Here the suffix ‘\_bh‘ refers to “**Bottom Halves**“.

**spin\_unlock\_bh(spinlock\_t \*lock)**

It will release the lock and re-enables the soft interrupts which are disabled by the above call.



**Locking between Hard IRQ and Bottom Halves:**

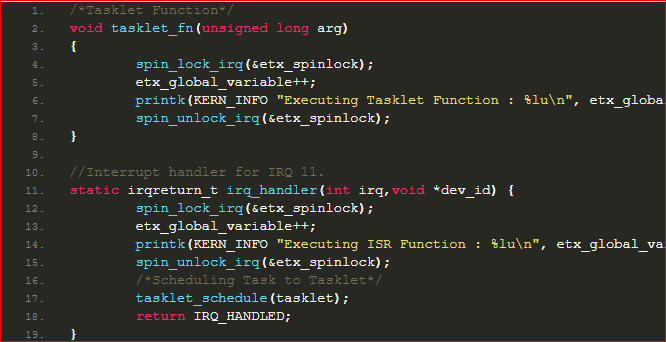
If you share data between Hardware ISR and Bottom halves then you have to disable the IRQ before lock. Because the bottom halves processing can be interrupted by a hardware interrupt.

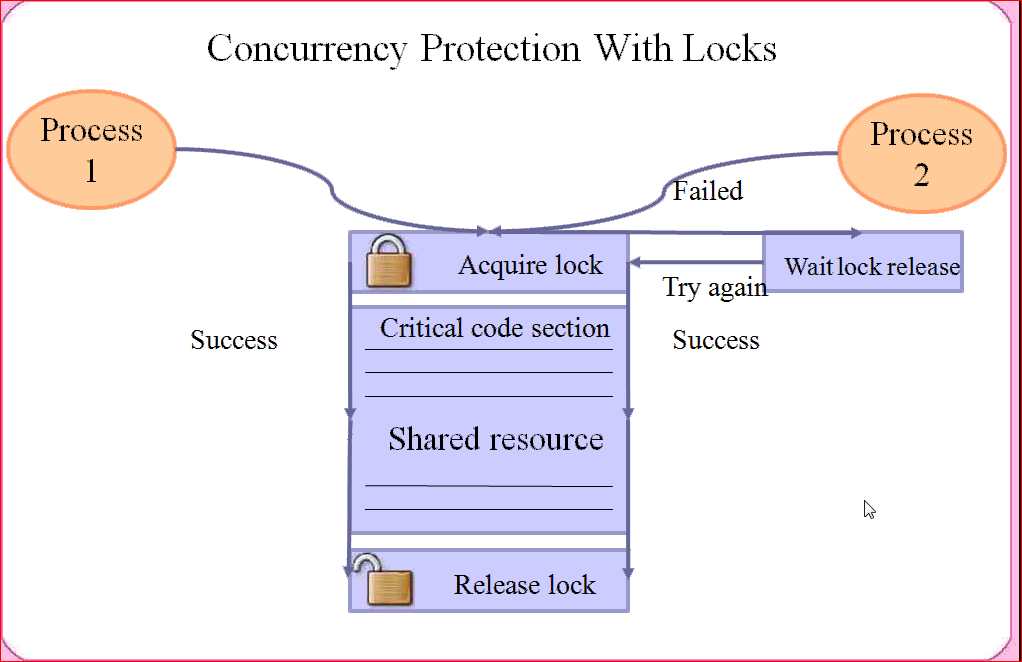
**spin\_lock\_irq(spinlock\_t \*lock)**

This will disable interrupts on that cpu, then grab the lock.

**spin\_unlock\_irq(spinlock\_t \*lock)**

It will release the lock and re-enables the interrupts which are disabled by the above call.



1. 

## IOCTL:

IOCTL is referred to as Input and Output Control, which is used to talking to device drivers.

It is used for special operation in device file IOCTL. The special operation are vendor /device specific .

The ioctl will allow 256 operations/functionality in single IOCTL.

Header file:

Kernel :

**linux/ioctl.h**

User Space:

**<sys/ioctl.h>**

**long ioctl( "file descriptor","ioctl command","Arguments");**

<**file descriptor**>: This the open file on which the ioctl command needs to be executed, which would generally be device files.  
<**ioctl command**>: ioctl command which is implemented to achieve the desired functionality  
<**arguments**>: The arguments need to be passed to the ioctl command.

Steps:

* 1. We need to identify target device and find our total no of special operation need to support .
  2. We need to create one request command, the command should be unique across all drivers
  3. We need to implement IOCTl function in drivers source and implement Switch case to handle request .

**Define the ioctl code:**

**Create IOCTL Command in the Driver:**

**#define "ioctl name" \_\_IOX("magic number","command number","argument type")**

where IOX can be :  
“**IO**“: an ioctl with no parameters  
“**IOW**“: an ioctl with write parameters (copy\_from\_user)  
“**IOR**“: an ioctl with read parameters (copy\_to\_user)  
“**IOWR**“: an ioctl with both write and read parameters

* The Magic Number is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. some times the major number for the device is used here.
* Command Number is the number that is assigned to the ioctl. This is used to differentiate the commands from one another.
* The last is the type of data.

Example:

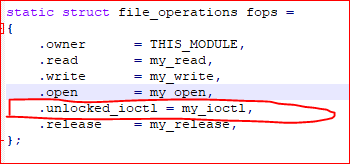
Read and write are special operation

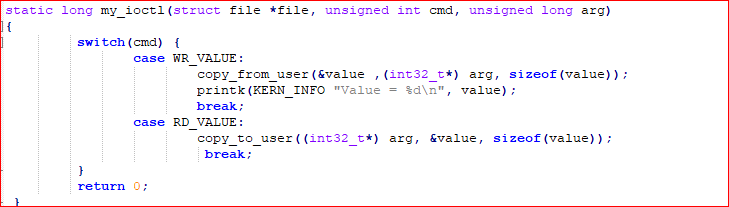
#define WR\_VALUE \_IOW('a','a',int32\_t\*) //(copy\_from\_user)

#define RD\_VALUE \_IOR('a','b',int32\_t\*) //(copy\_to\_user)

**Write IOCTL function in the driver:**

**int  ioctl(struct inode \*inode,struct file \*file,unsigned int cmd,unsigned long arg)**

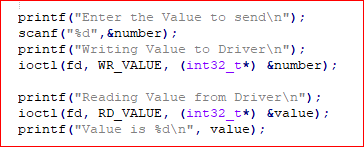




**Creating Ioctl in user space:**



**Writing IOCTL in user:**



Call flow:

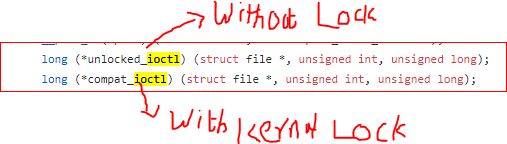
* 1. All file operation (fops)are execute in process context

ioctl----> sys\_ioctl----> big\_kernel\_lock()-🡪 acquire kernel lock

do\_ioctl------------------->fops--->compat\_ioctl==> my\_ioctl

unlock()

* 1. Ioctl fun now end up single thread call , this will not allow multi app / it is not a concurrent function., it is by default single thread function.
  2. If we register IOCTl call in fops with “.unlocked\_ioctl = my\_ioctl,” then it will work as concurrent function/without kernel lock.

one

## ProcFS:

The /dev tree contains [device nodes](http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/dev.html), which gives [user space](http://en.wikipedia.org/wiki/User_space) access to the [device drivers](http://en.wikipedia.org/wiki/Device_driver) in your OS's running kernel.

The /proc tree originated in System V Unix, where it only gave information about each running process, using a /proc/$PID/stuff scheme

In addition to these read-only information files, Linux's /proc also has writable virtual files that can change the state of the running kernel.

This folder is a mount point for the procfs (Process File system) which is a file system in memory. Many processes store information about themselves on this virtual file system. ProcFS also stores other system information.

It can act as a bridge connecting the user space and the kernel space. Userspace program can use proc files to read the information exported by kernel.

Every entry in the proc file system provides some information from the kernel.

* + /proc/meminfo-> The entry “meminfo”  gives the details of the memory being used in the system.
  + /proc/modules-> It gives similar information as lsmod.
  + /proc/devices — registered character and block major numbers
  + /proc/iomem — on-system physical RAM and bus device addresses
  + /proc/ioports — on-system I/O port addresses (especially for x86 systems)
  + /proc/interrupts — registered interrupt request numbers
  + /proc/softirqs — registered soft IRQs
  + /proc/swaps — currently active swaps
  + /proc/kallsyms — running kernel symbols, including from loaded modules
  + /proc/partitions — currently connected block devices and their partitions
  + /proc/filesystems — currently active filesystem drivers
  + /proc/cpuinfo — information about the CPU(s) on the system

proc files can also be used to control and modify kernel behavior on the fly. The proc files need to be writable in this case.

The proc file system is also very useful when we want to debug a kernel module. While debugging we might want to know the values of various variables in the module or maybe the data that the module is handling.

The proc entry can also be used to pass data to the kernel by writing into the kernel, so there can be two kinds of proc entries.

* An entry that only reads data from the kernel space.
* An entry that reads as well as writes data into and from kernel space.

## Header file :

#include<linux/proc\_fs.h>

**Creating procfs directory:**

**struct proc\_dir\_entry \*proc\_mkdir(const char \*name, struct proc\_dir\_entry \*parent)**

name: The name of the directory that will be created under /proc.  
parent: In case the folder needs to be created in a subfolder under /proc a pointer to the same is passed else it can be left as NULL

Example: proc\_dir = proc\_mkdir("my\_proc\_dir",NULL);

**Creating Procfs Entry**

**static inline struct proc\_dir\_entry \*proc\_create ( const char \*name, umode\_t mode, struct proc\_dir\_entry \*parent, const struct file\_operations \*proc\_fops )**

<name>: The name of the proc entry  
<mode>: The access mode for proc entry  
<parent>: The name of the parent directory under /proc. If NULL is passed as a parent, the /proc directory will be set as a parent.  
<proc\_fops>: The structure in which the file operations for the proc entry will be created.

Example :

/\*Creating Proc entry\*/

proc\_file = proc\_create("my\_proc",0666,proc\_dir ,&proc\_fops);

if(proc\_file == NULL)

{

printk(KERN\_ALERT "Error could not open %s\n","proc\_file ");

}

## Procfs File System

Now we need to create file\_operations structure proc\_fops in which we can map the read and write functions for the proc entry.

static struct file\_operations proc\_fops = {

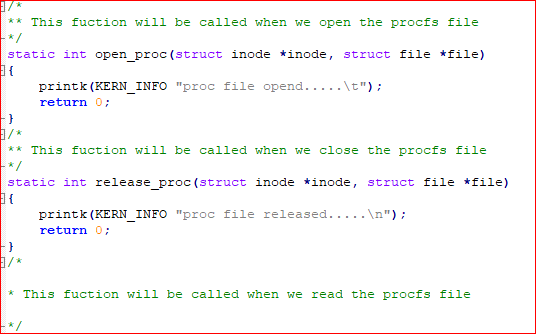
.open = open\_proc,

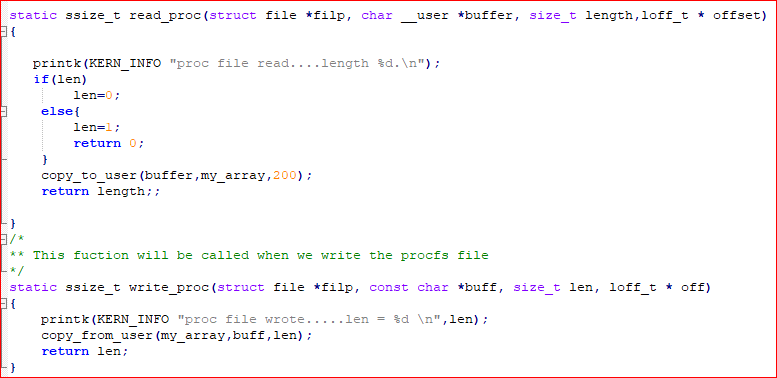
.read = read\_proc,

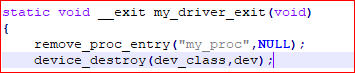
.write = write\_proc,

.release = release\_proc

};



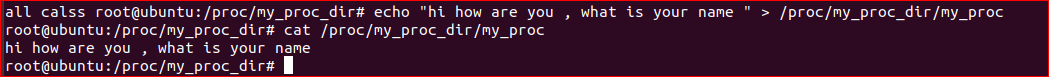


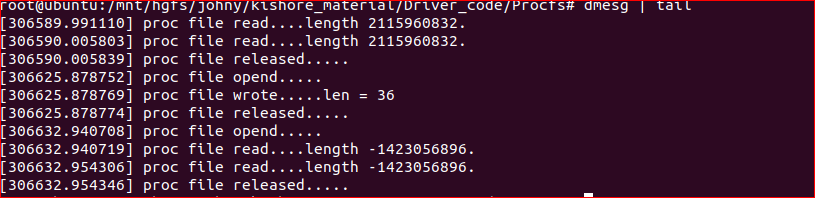


Example:

## Procfs folder .

How to read and write data in /proc file





Advantage :

/proc as a pipe to kernel. You can modify kernel parameters and see what kernel is doing at a moment in time.

* PROC file system is completely managed by the kernel, and is not stored on disk like other file system.
* Its stored in RAM (memory)
* Most of the files in /proc is of 0 bytes in size. (this is quite interesting)
* proc is a special file system and is not associated with any hard drive device.
* Files inside /proc are not real files, they act as an interface to kernel data structures and process information. As they are not real files, properties like file size is not applicable to them (hence shown as zero bytes)
* Contents inside /proc files are populated dynamically when requested. Due to this the data fetched from /proc is the most recent data provided by the kernel
* Certian files inside /proc can be modified to change the behaviour of a running kernel. For example, /proc/sys/ files.
* Most of the system monitoring commands like ps, top, free, etc use process files inside /proc/ to fetch information.
* Complete detail of a running process can be fetched from the /proc/<pid> directory.
* A special directory called /proc/self can be used by a program to find details about its own process.

## Block IO:

While implement Kernel service or routinizing, sometimes we need to check the available resources , if available resource are busy we need to wait/sleep till those are available .

Methods to wait in kernel:

## Semaphore :

## Completion

## Completion:

Its,the name itself says. When we want to notify or wakeup some thread or something when we finished some work, then we can use completion.

Example: We want to wait for one thread for something to run. Until that time that thread has to sleep. Once that process finished then we need to wake up that thread which is sleeping. We can do this by using completion without race conditions.

## Header file:

**#include <linux/completion.h>**

There are 5 important steps in Completions.

1. Initializing Completion
2. Re-Initializing Completion
3. Waiting for completion (The code is waiting and sleeping for something to finish)
4. Waking Up Task (Sending signal to sleeping part)
5. Check the status

## Initialize Completion:

We need to create a variable of type struct completion**,**which has only two fields:

struct completion {

unsigned int done;

wait\_queue\_head\_t wait;

};

Static Method :

**DECLARE\_COMPLETION(my\_wait);**

Dynamic method:

**init\_completion (struct completion \* my\_wait);**

My\_wait -> is the name of the struct which is going to create statically

In this **init\_completion** call we initialize the waitqueue and set doneto 0, i.e. “not completed” or “not done”.

**Re-Initializing Completion:**

**reinit\_completion (struct completion \* x);**

Where, x – completion structure that is to be reinitialized

**Example:reinit\_completion (&my\_wait);**

This function should be used to reinitialize a completion structure so it can be reused. This is especially important after complete\_all is used. This simply resets the ->done field to 0 (“not done”), without touching the waitqueue.

**Waiting for completion:**

This is used to make the function waits for the completion of a task.

**void wait\_for\_completion (struct completion \* x);**

x – holds the state of this particular completion

**wait\_for\_completion\_timeout**

This is used to make the function waits for completion of a task with a timeout.

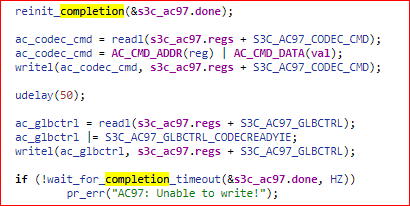
**unsigned long wait\_for\_completion\_timeout (struct completion \* x, unsigned long timeout);**

x – holds the state of this particular completion

timeout – timeout value in jiffies

return 0 , It **returns** **0**if timed out, and **positive**

This waits for either completion of a specific task to be signaled or for a specified timeout to expire. The timeout is in jiffies. It is not interruptible.



**wait\_for\_completion\_interruptible:**

This waits for completion of a specific task to be signaled. It is interruptible.

**int wait\_for\_completion\_interruptible (struct completion \* x);**

where, x – holds the state of this particular completion

It return -ERESTARTSYS if interrupted, 0 if completed

**wait\_for\_completion\_interruptible\_timeout**

This waits for either completion of a specific task to be signaled or for a specified timeout to expire

**Waking Up Task**

**complete**

This will wake up a single thread waiting on this completion. Threads will be awakened in the same order in which they were queued.

**void complete (struct completion \* x);**

where, x – holds the state of this particular completion

**complete\_all**

This will wake up all threads waiting on this particular completion event.

**void complete\_all (struct completion \* x);**

where, x – holds the state of this particular completion

**Check the status**

**completion\_done**

This is the test to see if a completion has any waiters.

**bool completion\_done (struct completion \* x);**

where, x – holds the state of this particular completion

It returns 0 if there are waiters (wait\_for\_completion in progress) **1** if there are no waiters.

This completion\_done() is safe to be called in IRQ or atomic context.

Example : Completation

## Waitqueue in Linux:

There are 3 important steps in Waitqueue.

1. [Initializing Waitqueue](https://embetronicx.com/tutorials/linux/device-drivers/waitqueue-in-linux-device-driver-tutorial/#Initializing_Waitqueue)
2. [Queuing](https://embetronicx.com/tutorials/linux/device-drivers/waitqueue-in-linux-device-driver-tutorial/#Queuing) (Put the Task to sleep until the event comes)
3. [Waking Up Queued Task](https://embetronicx.com/tutorials/linux/device-drivers/waitqueue-in-linux-device-driver-tutorial/#Waking_Up_Queued_Task)

**Static Method**

**DECLARE\_WAIT\_QUEUE\_HEAD(wq);**

Where the “wq” is the name of the queue on which task will be put to sleep.

**Dynamic Method**

**wait\_queue\_head\_t wq;**

**init\_waitqueue\_head (&wq);**

**wait\_event:**

sleep until a condition gets true.

**wait\_event(wq, condition);**

wq – the waitqueue to wait on

condition – a C expression for the event to wait for

The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the condition evaluates to true. The condition is checked each time the waitqueue wq is woken up.

**wait\_event\_timeout**

sleep until a condition gets true or a timeout elapses

wait\_event\_timeout(wq, condition, timeout);

wq –  the waitqueue to wait on

condtion – a C expression for the event to wait for

timeout –  timeout, in jiffies

The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the condition evaluates to true or timeout elapses. The condition is checked each time the waitqueue wq is woken up.

**wait\_event\_cmd:**

sleep until a condition gets true

wait\_event\_cmd(wq, condition, cmd1, cmd2);

wq –  the waitqueue to wait on

condtion – a C expression for the event to wait for

cmd1 – the command will be executed before sleep

cmd2 – the command will be executed after sleep

The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the condition evaluates to true. The condition is checked each time the waitqueue wq is woken up.

**wait\_event\_interruptible:**

sleep until a condition gets true

wait\_event\_interruptible(wq, condition);

wq –  the waitqueue to wait on

condtion – a C expression for the event to wait for

The process is put to sleep (TASK\_INTERRUPTIBLE) until the condition evaluates to true or a signal is received. The condition is checked each time the waitqueue wq is woken up.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if condition evaluated to true.

**wait\_event\_interruptible\_timeout:**

sleep until a condition gets true or a timeout elapses

wait\_event\_interruptible\_timeout(wq, condition, timeout);

wq –  the waitqueue to wait on

condtion – a C expression for the event to wait for

timeout –  timeout, in jiffies

The process is put to sleep (TASK\_INTERRUPTIBLE) until the condition evaluates to true or a signal is received or timeout elapes. The condition is checked each time the waitqueue wq is woken up.

It returns, 0 if the condition evaluated to false after the timeout elapsed, 1 if the condition evaluated to true after the timeout elapsed, the remaining jiffies (at least 1) if the condition evaluated to true before the timeout elapsed, or -ERESTARTSYS if it was interrupted by a signal.

**wait\_event\_killable:**

sleep until a condition gets true

wait\_event\_killable(wq, condition);

wq –  the waitqueue to wait on

condtion – a C expression for the event to wait for

The process is put to sleep (TASK\_KILLABLE) until the condition evaluates to true or a signal is received. The condition is checked each time the waitqueue wq is woken up.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if condition evaluated to true.

**Waking Up Queued Task:**

**wake\_up:**

wakes up only one process from the wait queue which is in non-interruptible sleep.

wake\_up(&wq);

wq – the waitqueue to wake up

**wake\_up\_all:**

wakes up all the processes on the wait queue

wake\_up\_all(&wq);

wq – the waitqueue to wake up

**wake\_up\_interruptible:**

wakes up only one process from the wait queue that is in interruptible sleep

wake\_up\_interruptible(&wq);

wq – the waitqueue to wake up

## whats the difference between wait\_event\_interruptible and wait\_for\_completion\_interruptible ?

wait\_event sleeps until a wait\_queue is awoken (by wake\_up) and then  
checks a condition -- which can be arbitrary -- and goes back to sleep  
if it's false. wake\_up wakes up \_all\_ processes on that wait\_queue.  
The condition is checked so that you can not miss the wake up (assured  
by careful ordering).  
  
wait\_for\_completion waits on completion, which is basically a semaphore.  
One complete will wake up one wait\_for\_completion. Wait for completion  
does not check any additional conditions.

sleep until a condition gets true

**wait\_event\_interruptible(wq, condition);**

**wq** –  the waitqueue to wait on

**condtion** – a C expression for the event to wait for

The process is put to sleep (TASK\_INTERRUPTIBLE) until the ***condition*** evaluates to true or a signal is received. The ***condition*** is checked each time the waitqueue ***wq*** is woken up.

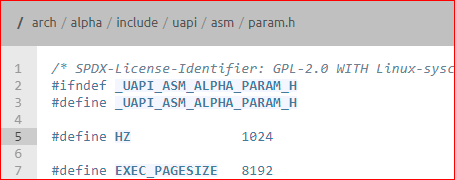
The function will **return** **-ERESTARTSYS** if it was interrupted by a signal and 0 if ***condition*** evaluated to true.

## Async Notification:

## Timers :

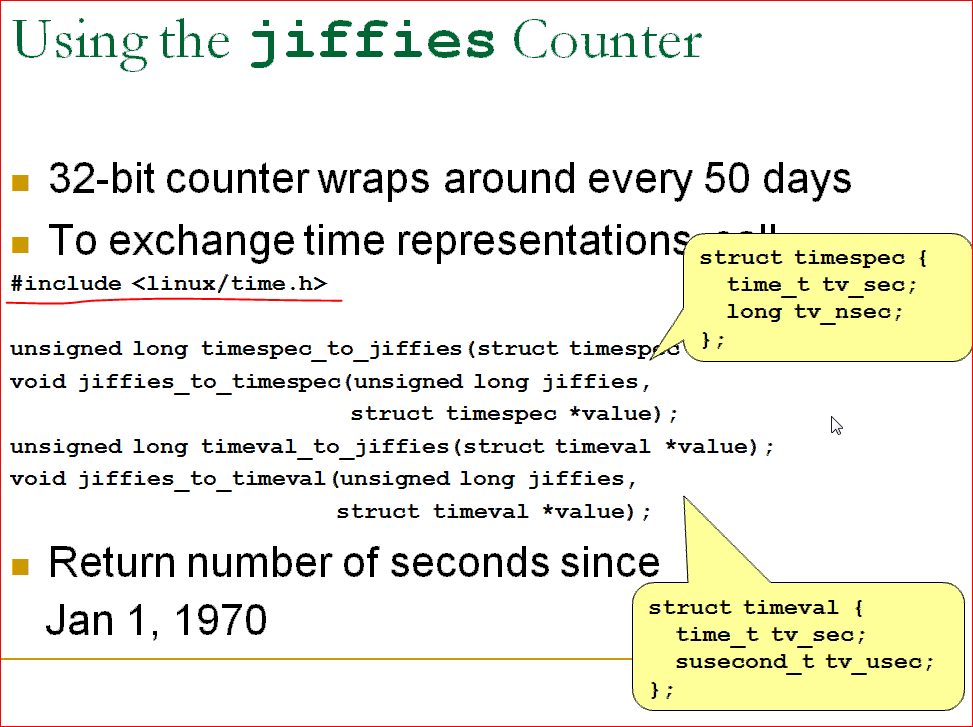
Kernel keep track of time via timer interrupts.

1. Generated by the timing hardware
2. Programmed at boot time according to HZ (default value is 1024)
3. At time the timer interrupt was configure as 1024 interrupts per second.
4. We have provision to modify HZ value



1. Every time a timer interrupt occur, kernel counter called jiffies will increment .

Init to zero at boot time .

1. This timer interrupts are generated at regular timer intervals by using system’s timing hardware.
2. Every time a timer interrupt occurs, the value of an internal kernel counter is incremented.
3. The counter is initialized to 0 at system boot, so it represents the number of clock ticks since last boot.
4. jiffies ==> On kernel boot-up, jiffies is initialized to a special initial value, and it is incremented by one for each timer interrupt.
5. Scheduler will use jiffies time to schedule process and cpu time .
6. jiffies = no. of ticks
7. seconds to unit jiffies: seconds \* HZ
8. jiffies to seconds:jiffies / HZ
9. unsigned long now\_tick = jiffies ; // now
10. unsigned long next\_tick = jiffies + 1 ; // one tick from now
11. unsigned long timer\_later = jiffies + (10\*HZ) ; // 10s from now
12. unsigned long time\_fraction = jiffies + (HZ/10) ; // 0.1s from now
13. 

## Uses of Kernel Timers

1. Polling a device by checking its state at regular intervals when the hardware can’t fire interrupts.
2. The user wants to send some messages to another device at regular intervals.
3. Send error when some action didn’t happen in a particular time period.

**Header file :**

**#include <linux/timer.h>**

**#include <linux/jiffies.h>**

**Structure:**

**struct timer\_list {**

**unsigned long expires;**

**void (\*function)(unsigned long);**

**unsigned long data;**

**};**

Where as

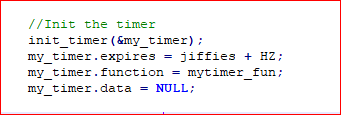
 Expires**->**field contains the expiration time of the timer (in jiffies).  
On expiration, function() will be called with the given datavalue.

## init\_timer

To initialize the timer, then you need to set the callback functionand data of the timer\_liststructure manually.

**void fastcall init\_timer ( struct timer\_list \* timer);**





OR

## setup\_timer

Instead of initializing timer manually by calling init\_timer, you can use this function to set dataand functionof timer\_liststructure and initialize the timer

### void timer\_setup (timer, function, data);

**Example:**

**/\* setup your timer to call my\_timer\_callback \*/**

**timer\_setup (&my\_timer, mytimer\_fun, 0);**

**//Timer Callback function. This will be called when timer expires**

**void mytimer\_fun (unsigned long data){**

**}**

**OR**

## DEFINE\_TIMER

**DEFINE\_TIMER(\_name, \_function, \_expires, \_data)**

If we are using this method, then no need to create the timer\_liststructure on our side. The kernel will create the structure in the name of \_nameand initialize it.

\_name – name of the timer\_list structure to be created

\_function – Callback function to be called when the timer expires

\_expires – the expiration time of the timer (in jiffies)

\_data – data has to be given to the callback function

## Start a Kernel Timer

## add\_timer

This will start a timer.

**void add\_timer(struct timer\_list \*timer);**

## 

## Modifying Kernel Timer’s timeout

## mod\_timer

**int mod\_timer (struct timer\_list \* timer, unsigned long expires);**

This function is used to modify a timer’s timeout. This is a more efficient way to update the expire field of an active timer (if the timer is inactive it will be activated).

**mod\_timer(timer, expires)** is equivalent to:

**del\_timer(timer); timer->expires = expires; add\_timer(timer);**

## Stop a Kernel Timer

## del\_timer

**int del\_timer (struct timer\_list \* timer);**

This will deactivate a timer. This works on both active and inactive timers.

**Argument:**

timer– the timer needs to be deactivated

**Return:**

The function returns whether it has deactivated a pending timer or not.

0 – del\_timerof an inactive timer

1 – del\_timerof an active timer

## del\_timer\_sync

**int del\_timer\_sync (struct timer\_list \* timer);**

This will deactivate a timer and wait for the handler to finish. This works on both active and inactive timers.

**Argument:**

timer– the timer needs to be deactivated

## timer\_pending

**int timer\_pending(const struct timer\_list \* timer);**

This will tell whether a given timer is currently pending, or not.

Example : timer folder

## Delay Execution:

Long (multi-jiffy) delay(mill sec to sec )

Busy waiting (cpu\_relax()

yielding the processor (schedule\_timeout)

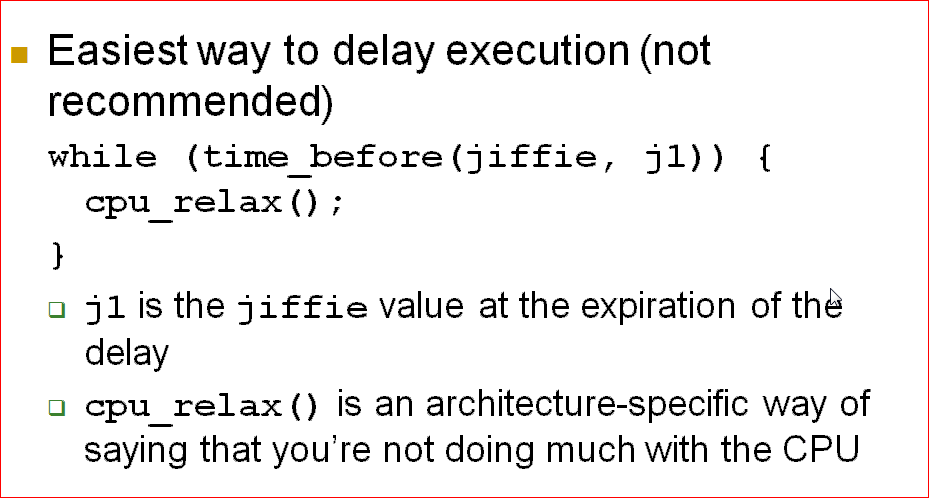
Timeout-(wait-events)

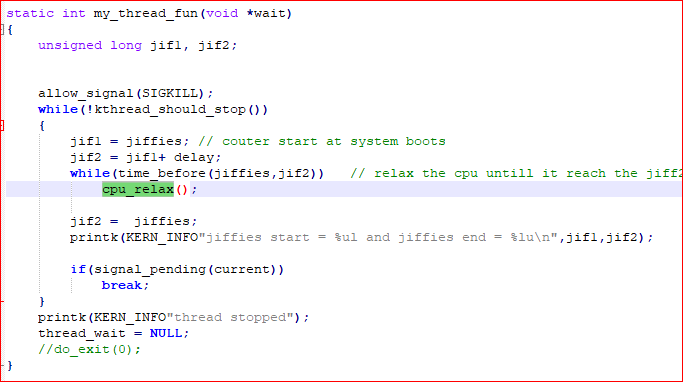
Short delay(micro sec)

**Busy waiting :** Busy-waiting approach just involves executing a while loop which wastes CPU cycles since it is not a productive work.

If the critical section code is of few lines, then its better to do busy-waiting since CPU spends very less waiting

But if the critical section code consists of thousands of lines, its better to use sleep-and-wake approach.





Several degrade system performance

If kernel does not allow preemption

Loop lock the processor for the duration of the delay

Scheduler never preempts kernel processes.

Computer lock dead until time J1 is reached .

If interrupts are disable if processor enter this loop.

Jiffier will not be updated.

Even for preemptive kernel .

Example: Busywait.

**Yielding the Processor:**

Explicitly release the cpu when not in using it

While(time\_before(jiffIe,j1)){

Schedule();

}

Behaviour similar to busy waiting under a preemptive kernel

Still consumes cpu cycles and battery power

No guarantee that the process will get the cpu back soon .

**Timeout:**

Another way to schedule timeout waiting for an event

**Header file:**

#include <linux/sched.h>

Signed long schedule\_timeout(signed long timeout);

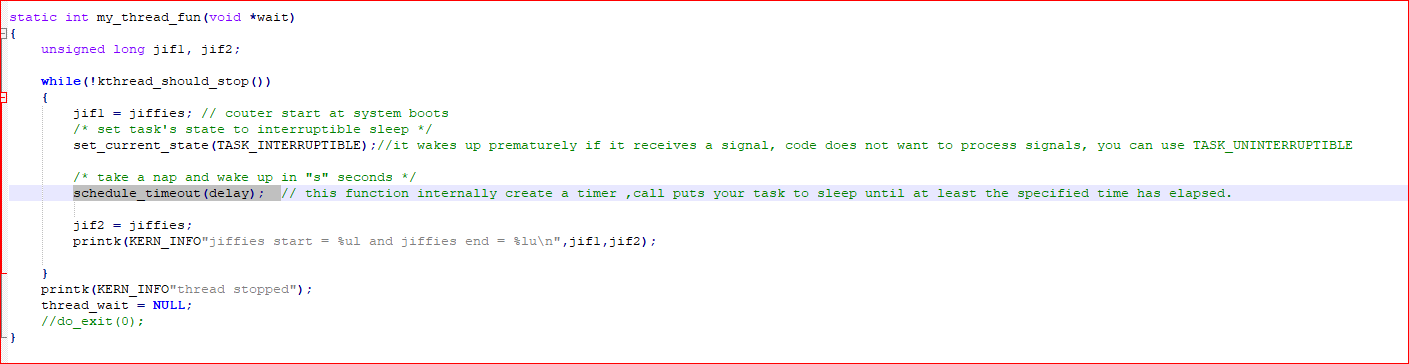
Timeout-> no. of jiffies.

It has 2 states :

* + 1. INTERRUPTIBLE
    2. UNINTERRUPTABLE

A process may not resume immediately when timer expires.

Example : sched\_timer



**Interrupts:**

The hardware interrupts are the signals triggered by hw devices on the processor, The hardware devices are connected to a chip called interrupt controller.

Interrupt controller will forward the signal to processor,

No. of IRQ is specific to architecture .

**What happen when interrupt occur:**

* Upon receiving an interrupt, the interrupt controller sends a signal to the processor.
* The processor detects this signal and interrupts its current execution to handle the interrupt.
* The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.

**Interrupts**

**Maskable** – All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts. A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.

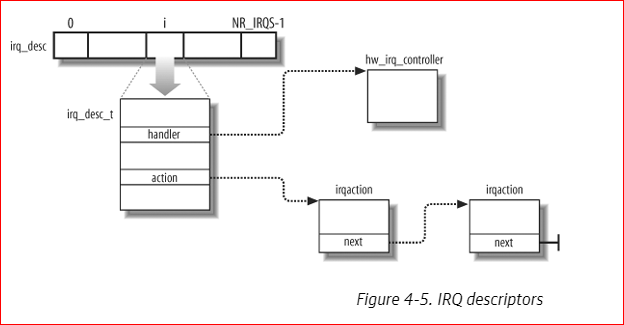
**Non maskable –** Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts. Non maskable interrupts are always recognized by the CPU.

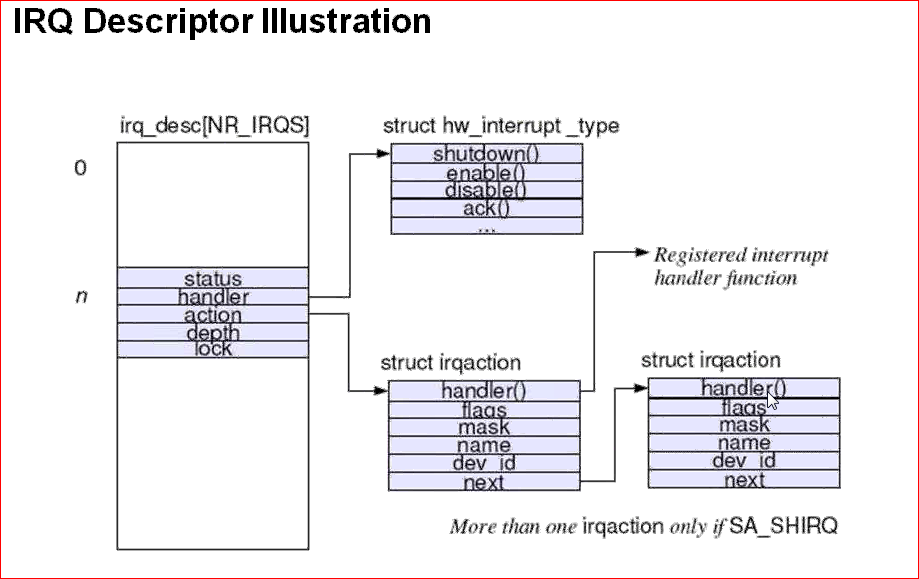
**Interrupt handler**

**How to implement ISR routine:**

* 1. The driver development should know which irq line the device is using to trigger interrupt/ which irq pin is for that device.
  2. Data structure = ITQ discreditable table
  3. Interrupt disreputable table is Linux kernel data structure , which is linked list of interrupt request line disreputable. Each disreputable hear has 2 operation
     + Action
     + Handler -> it is instance of type hardware interrupt type

The data structure that defines a PIC object is called hw\_interrupt\_type (also called hw\_irq\_controller).

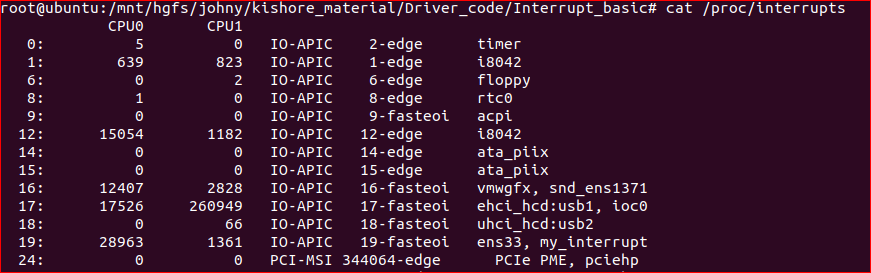




* 1. Linux allows a specific irq line is shared across different devices (like a audio and network card use same irq ). There is possibility of registering multiple devices registering same line , the order is serial not parallel. Ie., if one device complete its action Other device will access.

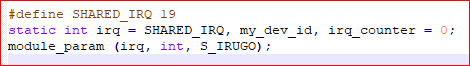
Cat /proc/interrupts 🡪 it dumps interrupt desctiptable table .

Irq no cpu0 cpu1 type name



Steps:

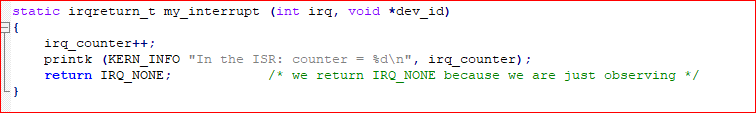
Define the IRQ



Create a key function on irq action



Implement handler :



How to check :



Unregister the irq:



**IRQ flags:**

*\* IRQF\_SHARED - allow sharing the irq among several devices*

*\* IRQF\_PROBE\_SHARED - set by callers when they expect sharing mismatches to occur*

*\* IRQF\_TIMER - Flag to mark this interrupt as timer interrupt*

*\* IRQF\_PERCPU - Interrupt is per cpu*

*\* IRQF\_NOBALANCING - Flag to exclude this interrupt from irq balancing*

*\* IRQF\_IRQPOLL - Interrupt is used for polling (only the interrupt that is*

*\* registered first in a shared interrupt is considered for*

*\* performance reasons)*

*\* IRQF\_ONESHOT - Interrupt is not reenabled after the hardirq handler finished.*

*\* Used by threaded interrupts which need to keep the*

*\* irq line disabled until the threaded handler has been run.*

*\* IRQF\_NO\_SUSPEND - Do not disable this IRQ during suspend. Does not guarantee*

*\* that this interrupt will wake the system from a suspended*

*\* state. See Documentation/power/suspend-and-interrupts.rst*

*\* IRQF\_FORCE\_RESUME - Force enable it on resume even if IRQF\_NO\_SUSPEND is set*

*\* IRQF\_NO\_THREAD - Interrupt cannot be threaded*

*\* IRQF\_EARLY\_RESUME - Resume IRQ early during syscore instead of at device*

*\* resume time.*

*\* IRQF\_COND\_SUSPEND - If the IRQ is shared with a NO\_SUSPEND user, execute this*

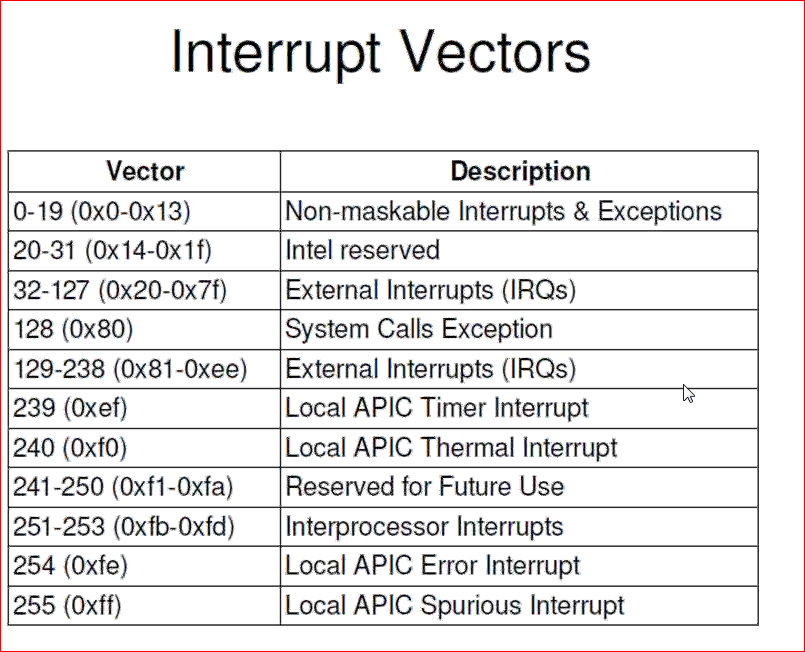
*\* interrupt handler after suspending interrupts. For system*

*\* wakeup devices users need to implement wakeup detection in*

*\* their interrupt handlers.*

*\*/*

How to implement irq and issue :

Base addeess 

* When irq trigger on cpu , the process will jump to vector table whose base address is configured on process boot time
* It contain list of descriptables at cpu level.
* 32 -127 are external interrupts .
* Linux kernel configure do\_irq routine as a default response function for all external interrupts
* Do\_irq is routine of process zero
* Which is responsible for allocation of interrupt stack and invoke appropriate interrupt service function
* The interrupt stack has 2 types
  1. 8k
  2. 4k

8k -> is default , same process stack will used for interrupt

If kernel configure as 4k stack then 4k for process and 4k for interrupt

Steps for do\_irq:

* 1. Find interrupt request line on which interrupt will trigger
  2. Look IDT for address of register interrupt service routine
  3. Invoke register irq
  4. Enable irq line
  5. Execute with priority work
  6. Invoke processor scheduler

When irq handle was executed /running all other process was suspended. interrupt will use process stack . This handle will give highest priority (permission was disable)

Interrupt Latency: The amount of time system spend to respond interrupt.

1. Hardware latency: the amount of time process is taking to ack/signal to jump into vector table

The amount of time cpu spend.

1. Kernel latency : the amount of time kernel is taking to loop/start do\_IRQ(process zero time to start irq.
2. Interrupt handler :irq routine execution time
3. Soft irq latency
4. Scheduler :
   * 1. Check run queue(check for higest priority task)
     2. Signal handler
     3. Taking one task and give cpu sclice.

The amount of time scheduler is taking to select a task and give to cpu.

To Overcome interrupt latency we need to overcome all factors .

Case study:

Network card:

Each packet will received / trigger IRQ.

Steps involve to handle the data:

1. Allocate buffer to hold packet
2. Copy the data/packet to memory
3. Process
4. Deliver to upper layer

Sudo code :

IRQ

{

* 1. Allocate buffer
  2. Copy data from h/w
  3. Process physical header
  4. Queue packet for protocol layer

}

Problem : It will never fix the time

While implement ISR the following issues are to be consider.

Do and Donts:

Donts:

1. Avoid calling memory routine
2. Avoid transferring data b/w two buffers block (sync mode)
3. Avoid containing for access any global data structure.
4. Operation of user space address
5. Call to scheduler
6. Avoid calling non atomic operations

Do:

1. Use pre allocate buffer
2. Consider using DMA when ever data need to device and memory
3. Consider using pre cpu data when ever need
4. .Identify non critical work and use appropriate difference routine to execute them when system is idle or other schedule time.

Critical -> taking to hardware

Non Critical -> Asyns operation , doing later

Sudo do:

ISR

{

IRQ

Schedule\_bh(func)

}

Func()

{

Soft IRQ Bottom Half

Body

}

What is softIRQ?

Soft interrupt means any code that run with interrupt priority by suspending scheduler but IRQ enable.

Run with interrupt priority

Immediately after ISR finish before invoke scheduler .

When scheduler enable CPU will give to all process(workqueue)

Bottom Half :

2 types:

1. Soft interrupt/ tasklets 🡪 interrupt context
2. Work queue 🡪 process context.

Soft irq:

1. Statically Allocate at compiler Time
2. Maximum of 32 softirqs
3. Preemption Disable , Interrupts Enable
4. Activating a softirq

Register Softirq with open\_softirq()

Raise Softirq with raise\_softirq()

1. Executing a Softirq
   1. Return from Hardware interrupt
   2. By the ksoftirq Thread
   3. Code that Explicitly check and Executes softirq

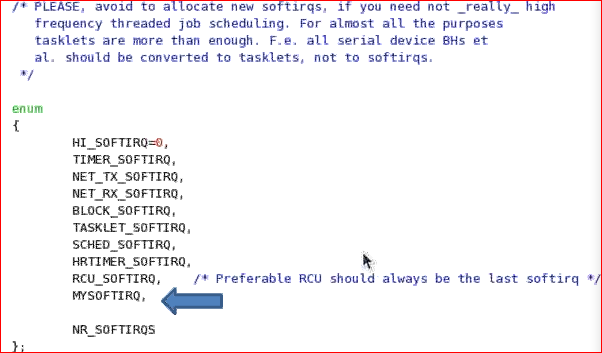
Header file :

Usr/src/linuxXXXX/include/linux/interrupts.h

Contain all interrupt related information.

Step

1. Each entry is a name of soft irq,at time compile softirq\_XXX action will create.



1. In Open \_softirq function we will use this name(MYSOFTIRQ) as a first arg and mention address of handler function.



This function should be in init module.

1. Handler

ISR

{

raise\_softirq(mysoftirq)

}

Func

{

----Non critical code

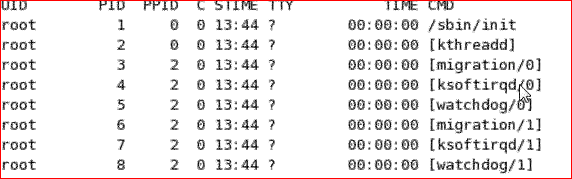
}

Step

1. When raise\_softirq gets called, specifies softirq instance is queued to the list of pending softirq’s(per cpu).its mandatory that top half and bottom halves should run on same cpu to resolve cache line issue.
2. Pending softirq list is clear by do\_irq routine immediately after isr terminate with interrupt line enable.
3. Preemption is disable, It run with highest priority with interrupts disable ,Soft irqs are permitted only with nested IRQ .
4. Softirq is getting block or permitted only with nested interrupt.
5. Softirq may execute in the context of ksoftirqd (percpu kernel thread), if you reschedule too many time it will execute in ksofirqd context.
6. Spinunlock \_bh – process/kernel context (before execution do\_irq will check the lock , after finish of lock it will execute do\_irq action)
7. Kthread – process/kernel context
8. Do\_irq –interrupt context
9. Rescheduling bottom halves to relinquish cpu form with in a bottom halves when a critical resource require for the bottom half execution is not available.
10. Softirq will execute parally , by considering mutual exclusive lock .

Ps –Af

All square brackets are ktherad



Sudo :

Func()

{

/\*write to list\*/

}

Read

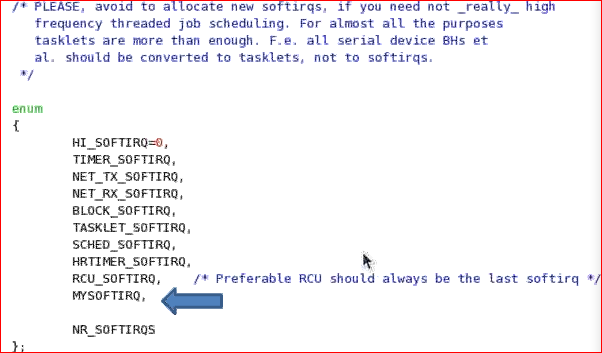
{

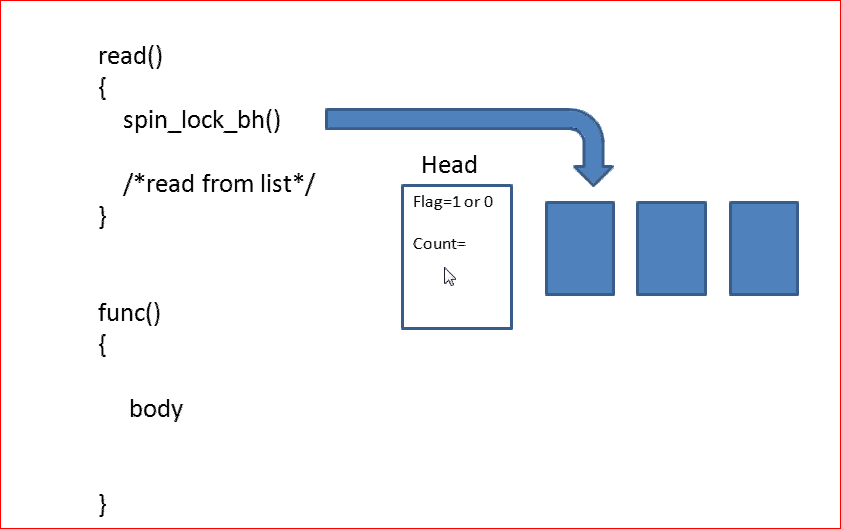
Spin\_lock\_bh()

/\*read from list\*/

Spin\_unlock\_bh()

}





Every softirq , we will maintains a flag that flag will indicate lock is enable or not , and maintains count .

Limitation of softirq:

1. Softirq are concurrent.,ie., same softirq could run on N number CPU parallel .
2. While implement soft irq code using mutual exclusion lock is mandatory where ever needed.
3. Using locks in interrupt code will result in variable time interrupt latency.

Because of this limitation so most of the modules are not used .

Tasklets:

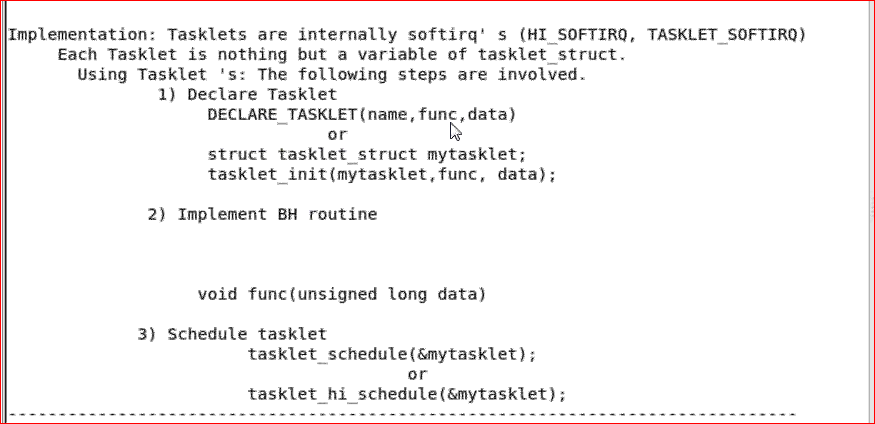
A copy of a soft irq without concurrency is tasklet.

Tasklet will run in serial botom half .

Tasklet are dynamic softirqs that can be used form with in module drivers without concurrency (tasklets always executed serially)

1. Build on top of softirqs
2. Represented by two softirqs
3. Each tasklets has a tasklet\_struct structure
4. Created statically or Dynamically.
5. Less Restrictive synchronization requirements
6. Scheduled with tasklet\_schedule
7. Executed with other softirq

Implementation:



Tasklet\_hi\_schedule -> is high priority tasklet .

Execution possibilities:

1. Do\_irq routine after interrupt
2. Spinunlock \_bh – process/kernel context (before execution do\_irq will check the lock , after finish of loch it will execute do\_irq action)
3. Kthread – process/kernel context
4. Tasklet are executed using the same policy that is applied for softirq.since interrupt subsystem of kernel view a tasklet either as instance of type hi\_softirq or tasklet\_softirq.
5. Interrupt subsystem guaranties the following with regards to execution of tasklets.

Properties:

1. \* If tasklet\_schedule() is called, then tasklet is guaranteed, to be executed on same cpu(99.9 % same cpu if you are not rescheduled repeatedly ) at least once after this.

\* If the tasklet is already scheduled, but its execution is still not started, it will be executed only once.( When tasklet schedule is called with a tasklet instance which is already scheduled but the execution still not staretd but execute it once)

1. \* If this tasklet is already running on another CPU (or schedule is called from tasklet itself), it is rescheduled for later.(if the tasklet is already running on another cpu , it will be reschedule alter)
2. \* Tasklet is strictly serialized wrt itself, but not wrt another tasklets. If client needs some intertask synchronization, he makes it with spinlocks.(tasklet are strictly serialized wrt them self wrt other tasklet, if u r accessing global data structure it is advisable lock,same tasklet will never run parally)

Workqueue:

Work queue is instance of work structure.

Its is process context bh

It is used for long operation

Interrupts ON ARM :

* An **interrupt** is the automatic transfer of software execution in response to a hardware event that is **asynchronous** with the current software execution.
* This hardware event is called a **trigger**.
* When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag.

The software has dynamic control over some aspects of the interrupt request sequence.

1) device arm (each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate.)  
2) NVIC enable,  
3) global enable,  
4) interrupt priority level must be higher than current level executing, and  
5) hardware event trigger.

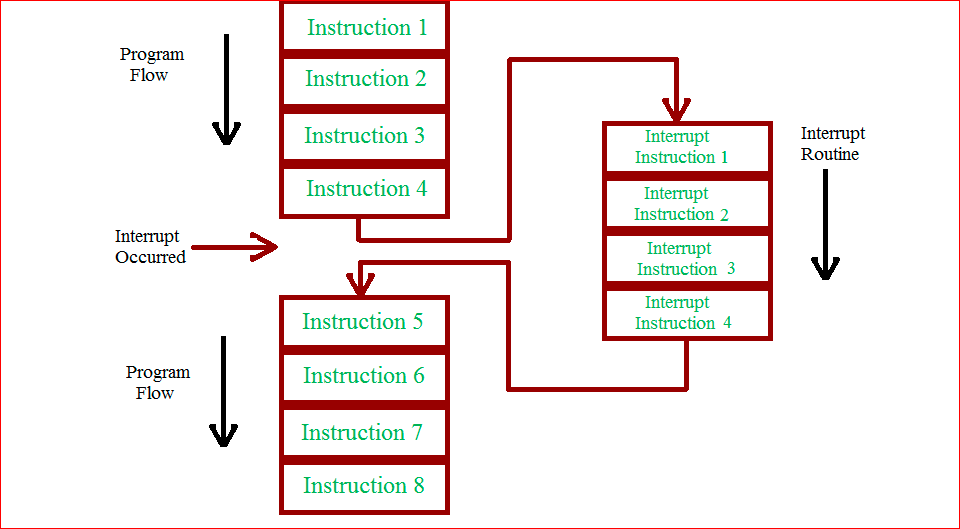
For an interrupt to occur, these five conditions must be simultaneously true but can occur in any order.

An interrupt causes the following sequence of five events

1. Current instruction is finished,
2. The execution of the currently running program is suspended,Eight ((**R0**, **R1**, **R2**, **R3**, **R12**, **LR**, **PC**, and **PSR** with the **R0** on top).)registers are pushed on the stack,
3. LR (is set to a specific value signifying an interrupt service routine (ISR) is being run (bits [31:4] to 0xFFFFFFF, and bits [3:0] specify the type of interrupt return to perform)is set to 0xFFFFFFF9,
4. IPSR is set to the interrupt number,
5. PC is loaded with the interrupt vector

These five steps, called a **context switch**, occur automatically in hardware as the context is switched from a foreground thread to a background thread(ISR).

For regular function calls we use the registers and stack to pass parameters, but interrupt threads have logically separate registers and stack. More specifically, registers are automatically saved by the processor as it switches from main program (foreground thread) to interrupt service routine (background thread). Exiting an ISR will restore the registers back to their previous values. Thus, all parameter passing must occur through global memory. One cannot pass data from the main program to the interrupt service routine using registers or the stack.



From the above figure, the CPU executes its normal set of codes until an IRQ occurs. When the IRQ signal is received, the CPU stops executing the regular code and starts executing the ISR. Once the execution of the ISR is completed by the CPU, it returns back to execution of the normal code.

Of the three categories, the FIQ requests have the highest priority, Vectored IRQ requests have the medium priority and non – vectored IRQ requests have the least priority.

In case of Vectored IRQ requests, the CPU has a knowledge of the ISR. A special table called Interrupt Vector Table (IVT) contains all the information about the Vectored IRQ.

In case of Non – Vectored IRQ, as the name itself indicates, the CPU isn’t aware of either the source of the Interrupt or the ISR address of the Interrupts. In this case, the CPU must be provided with a default ISR address.

**Note**: All the Non – Vectored IRQ Requests have the same ISR address which is defined in the VICDefVectAddr register.

Exception Handling:

An exception is any condition that needs to halt the normal sequential execution of instructions.

This section covers the following exception handling topics:

■ARM processor mode and exceptions

■Vector table

■Exception priorities

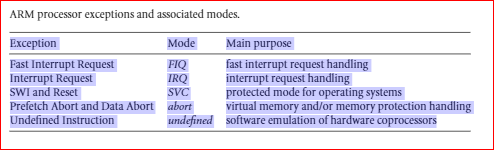
■Link register offset

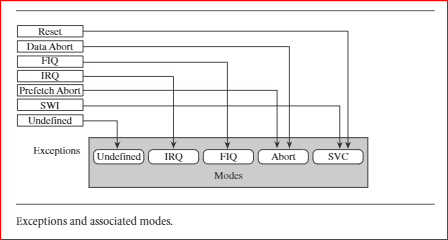
ARM processor exceptions and associated modes:

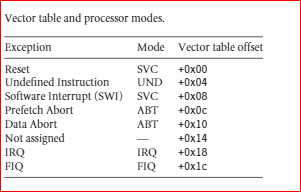
Each exception causes the core to enter a specific mode. In addition, any of the ARM processor modes can be entered manually by changing the cpsr.User and system mode are the only two modes that are not entered by a corresponding exception, in other words, to enter these modes you must modify the cpsr. When an exception causes a mode change, the core automatically

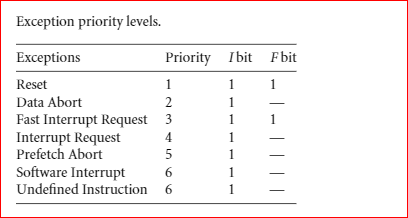
■ saves the cpsr to the spsr of the exception mode

■saves the pc to the lr of the exception mode









IRQ and FIQ Exceptions

An IRQ or FIQ exception causes the processor hardware to go through a standard procedure (provided the interrupts are not masked):

1. The processor changes to a specific interrupt request mode, which reflects the interrupt being raised.

2. The previous mode’s cpsr is saved into the spsr of the new interrupt request mode.

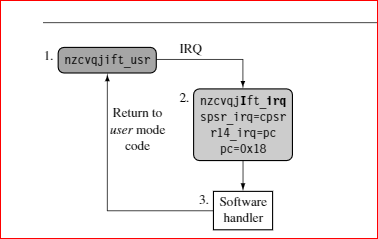
3. The pc is saved in the lr of the new interrupt request mode.

4.Interrupt/s are disabled—either the IRQ or both IRQ and FIQ exceptions are disabled in the cpsr. This immediately stops another interrupt request of the same type being raised.

5. The processor branches to a specific entry in the vector table.

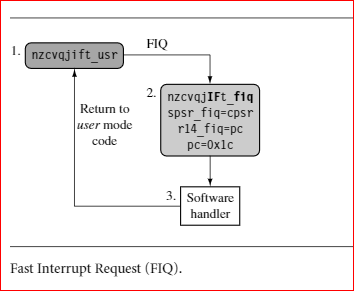
Example:

what happens when an IRQ exception is raised when the processor is in user mode. The processor starts in state 1. In this example both the IRQ and FIQ exception bits in the cpsr are enabled.When an IRQ occurs the processor moves into state 2. This transition automatically sets the IRQ bit to one, disabling any further IRQ exceptions. The FIQ exception, however,remains enabled because FIQ has a higher priority and therefore does not get disabled when a low-priority IRQ exception is raised. The cpsr processor mode changes to IRQ mode. The user mode cpsr is automatically copied into spsr\_irq.



Registerr14\_irq is assigned the value of the pc when the interrupt was raised. The pc is then set to the IRQ entry+0x18 in the vector table. In state 3 the software handler takes over and calls the appropriate interrupt service routine to service the source of the interrupt. Upon completion, the processor mode reverts back to the original usermode code in state 1.

FIQ:example of an FIQ exception. The processor goes through a similar procedure as with the IRQ exception, but instead of just masking further IRQ exceptions from occurring, the processor also masks out further FIQ exceptions. This means that both interrupts are disabled when entering the software handler in state 3.



Changing to FIQ mode means there is no requirement to save registers r8 to r12 since these registers are banked in FIQmode. These registers can be used to hold temporary data, such as buffer pointers or counters. This makes FIQ ideal for servicing a single-source,high-priority, low-latency interrupt.

Enabling and Disabling FIQ and IRQ Exceptions:

The ARM processor core has a simple procedure to manually enable and disable interrupts that involves modifying the cpsr when the processor is in a privileged mode

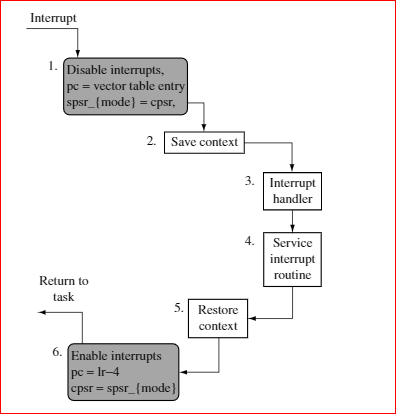
Basic Interrupt Stack Design and Implementation

The IRQ mode stack has to be set up before interrupts are enabled—normally in the initialization code for the system. It is important that the maximum size of the stack is known in a simple embedded system, since the stack size is reserved in the initial stages of boot-up by the firmware.

Interrupt Handling Schemes:

Non nested Interrupt Handler:

The simplest interrupt handler is a handler that is non nested: the interrupts are disabled until control is returned back to the interrupted task or process. Because a non nested interrupt handler can only service a single interrupt at a time,



Disable interrupt/s—When the IRQ exception is raised, the ARM processor will disable further IRQ exceptions from occurring. The processor mode is set to the appropri-ate interrupt request mode, and the previous cpsr is copied into the newly available spsr\_{interrupt request mode}. The processor will then set the pc to point to the correct entry in the vector table and execute the instruction. This instruction will alter the pc to point to the specific interrupt handler.

2.Save context—On entry the handler code saves a subset of the current processor moden on banked registers.

3.Interrupt handler—The handler then identifies the external interrupt source and executes the appropriate interrupt service routine (ISR).

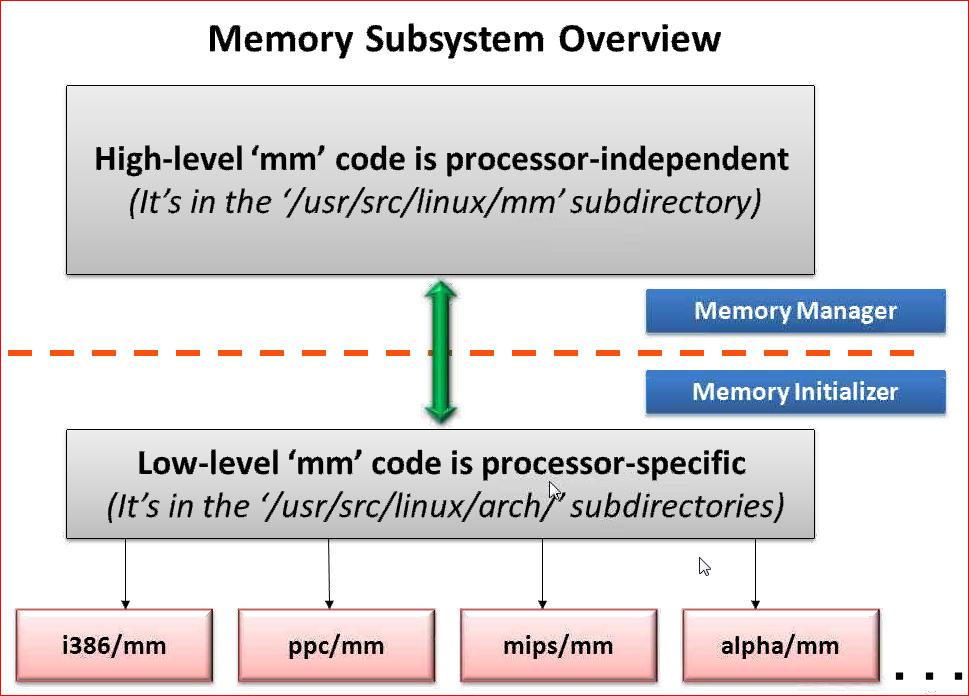
4.Interrupt service routine—The ISR services the external interrupt source and resets the interrupt.

5.Restore context—The ISR returns back to the interrupt handler, which restores the context.

6.Enable interrupts—Finally, to return from the interrupt handler, the spsr\_{interruptrequest mode}is restored back into the cpsr. The pc is then set to the next instruction after the interrupt was raised.

* Simple Nonnested Interrupt Handler
* Handles and services individual interrupts sequentially.
* High interrupt latency; cannot handle further interrupts occurring while an interruptis being serviced.
* Advantages: relatively easy to implement and debug.
* Disadvantage: cannot be used to handle complex embedded systems with multiplepriority interrupts.

MEMORY MANAGEMENT:



It has 2 modules :

* 1. Linux/mm- it is memory manager (paging etc)
  2. Linux/arch – all memory int will happen in this module(ie., arch specific)

This will run in boot time

Its is processor dependent.

Memory view :

Arch-x86:

Real Mode :When process start it will be in real mode .Process will view RAM in single array.

Protected mode:Memory view is set of array/vector. It is fixed size -4k (arch specific)

Switch time : Kernel boot time it switch to real mode to protected mode .



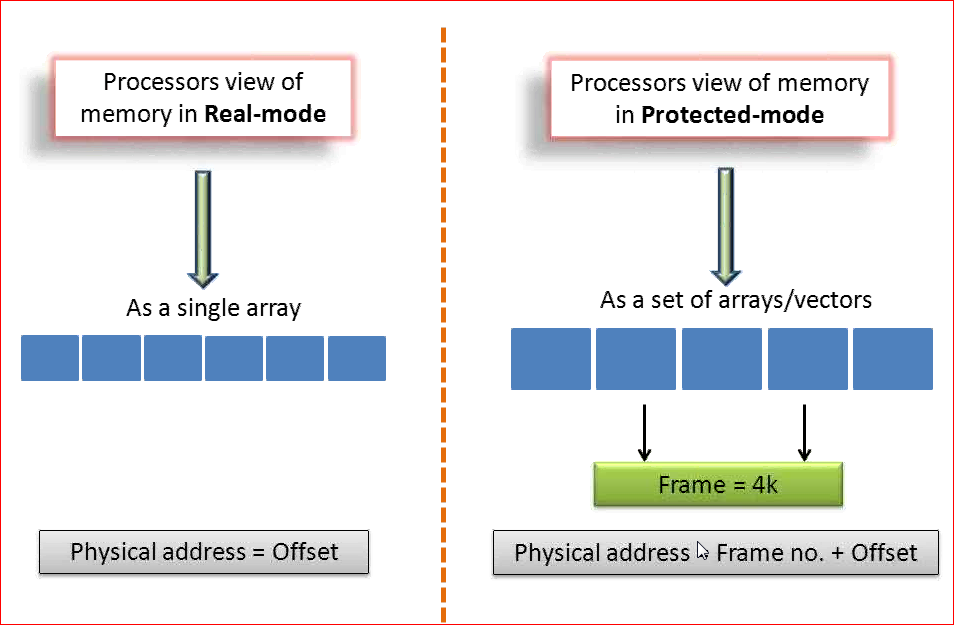
ARM arch :

Real mode : CPU view whole thing as single vector.

Physical address = offset of big array(RAM)

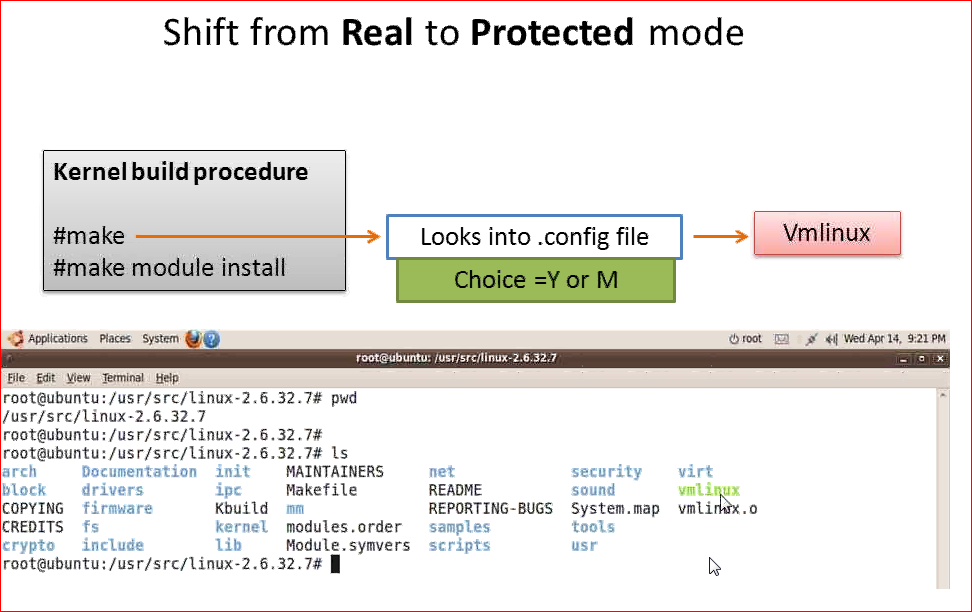
Protected Mode : CPU view changes to one array to multiple arrays.

Physical address = frame no + offset.



Shift form real mode to protected mode :

1. Make -> it check’s .config file, it has 2 entries Y and M, Y -> it look branches of src and compile and link and generate elf file .,ie Vmlinux.



1. Make execute in 2 halfs

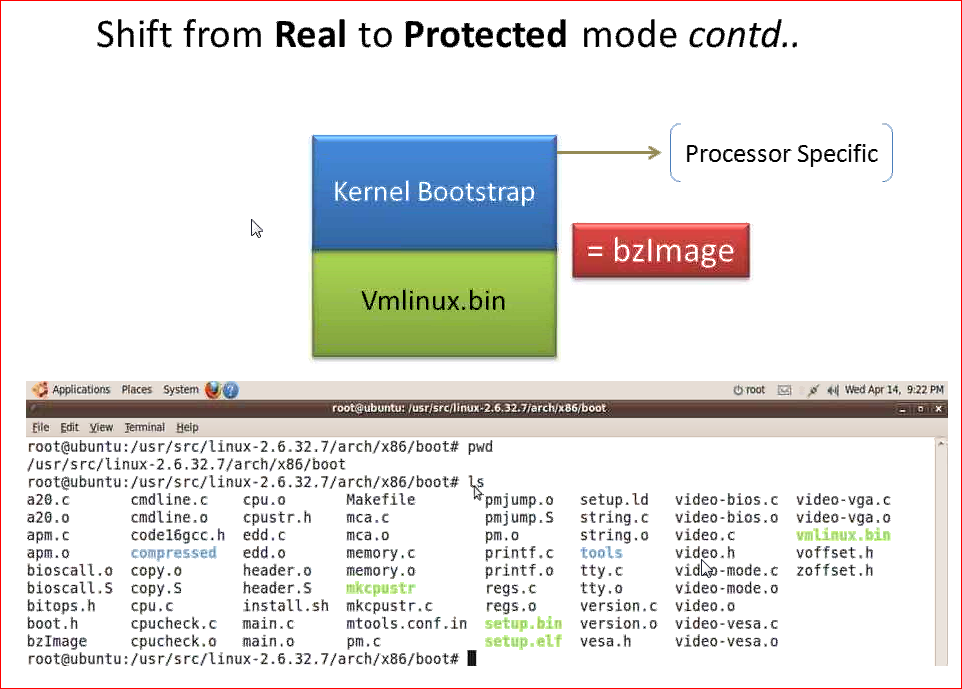
first half = vmlinix

second half is generate bzimage

it check arch folder of make file . it takes compressed vmlinux + Assembly code (kernel boot strap code) and generate bzimage/zimage.

Its is processor specific / header to kernel .

Bzimage = compressed kernel image + boot strap code .

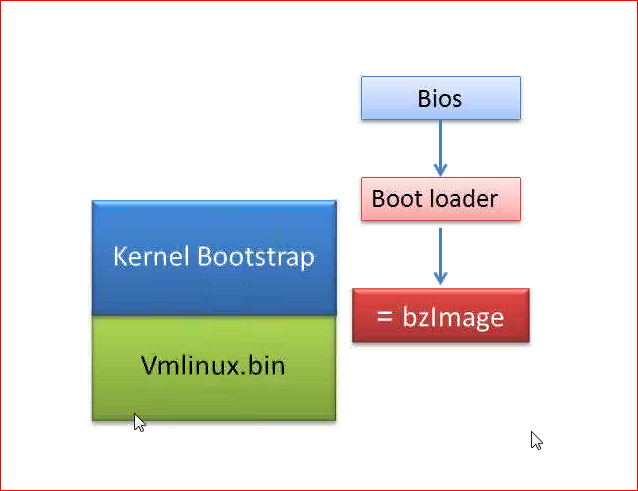


1. Who loads the kernel

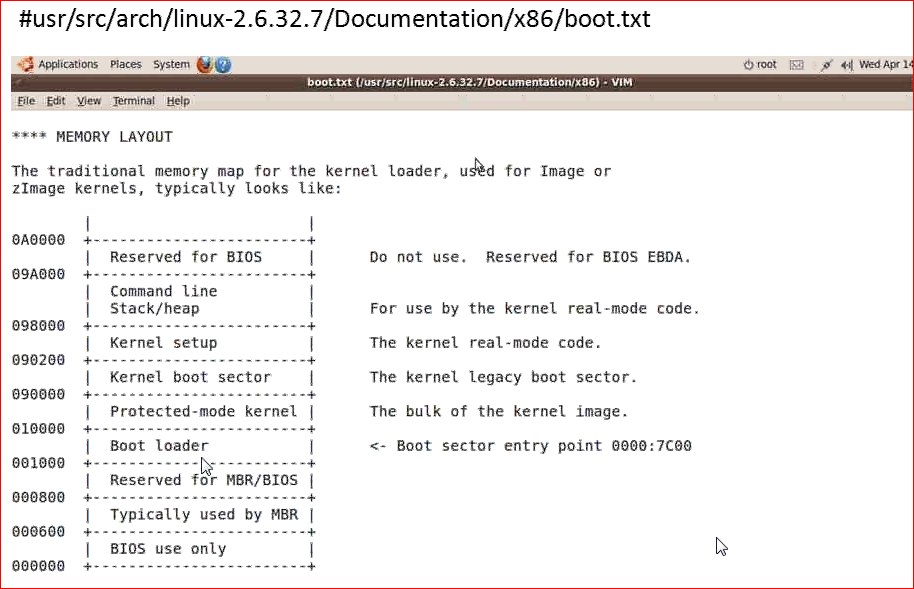
Please refer below diagram .

When bzimage start executing boot strap.

1. Basic init the drivers(i/o, VGA driver)
2. Shiting the CPU form real to protected
3. Uncompressing the image and start kernel.



1. X86 document :



Responsibilities of memory initializer:

1. Allocate a page list with instance of struct page representing each physical frame.

Page is obj represent of frame.

1. Categories pages in to appropriate zone.

3 zone – a. zone DMA (0-16MB)

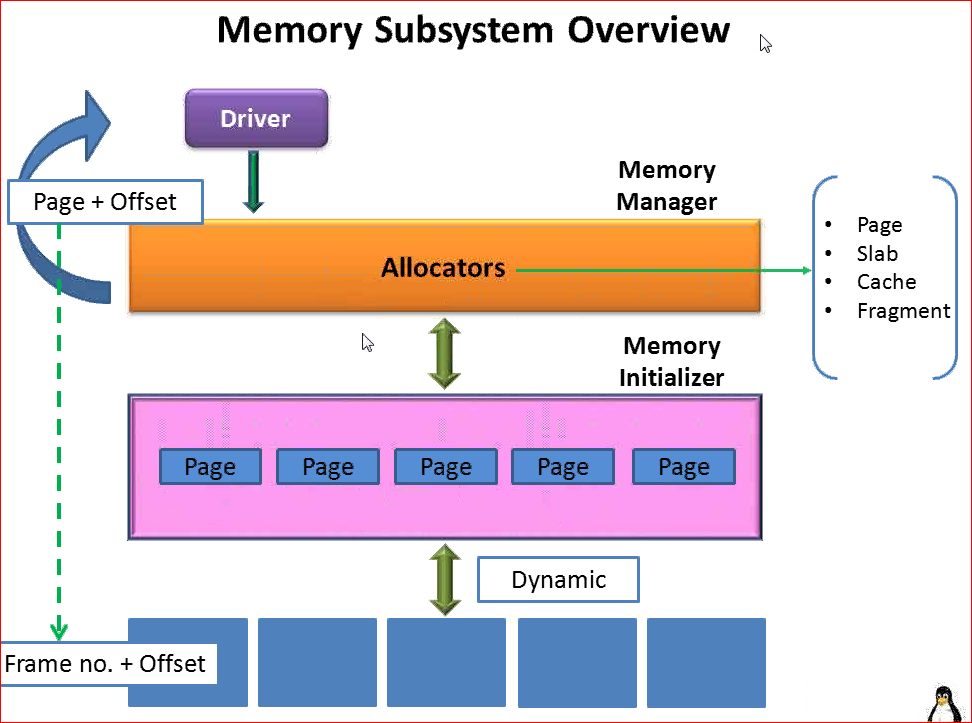
b.Zone Normal (16 to 896MB)

c. Zone High (896MB and Above)

3. Initialize high level MM .(all alg/ management will be done here)

4. 4 types of allocators

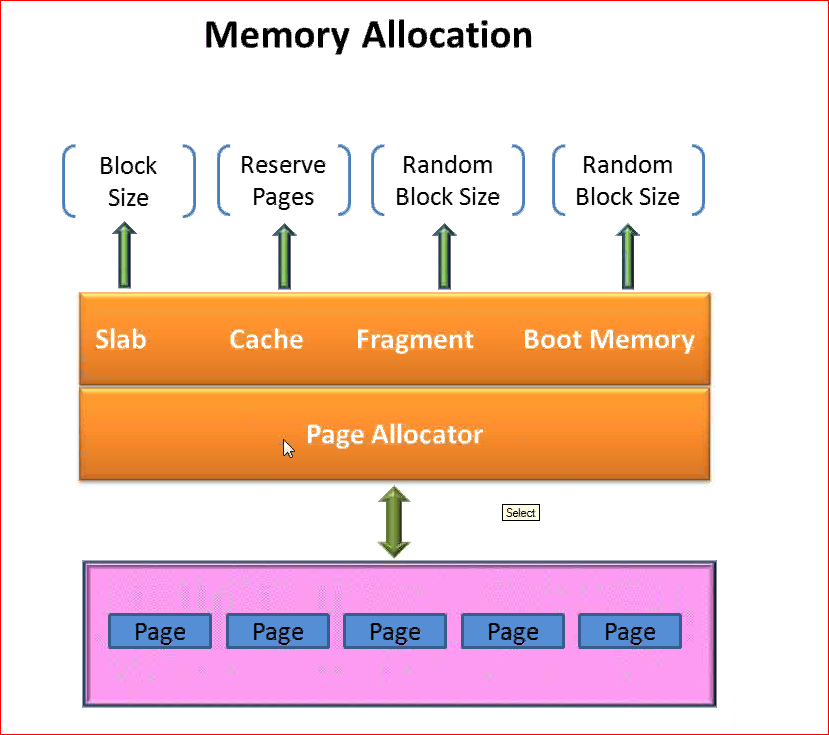
* + - Page-help to take a page /multiple pages
    - Slab- help u to allocate block (1k,50,mb etc)
    - Cache- help to reserve some memory
    - Fragments – connect all fragment memory and connect virtual continual address.



High level memory:

It has 5 allocators.

1. Page allocator
2. Slab
3. Cache
4. Fragments
5. Boot memory
6. Page allocator:
7. It has physical access to page list
8. Slab allocator:
9. Very often used
10. Provide allocation for N number of bytes
11. It will return physical continuous memory
12. Cache memory:
13. Available for kernel space
14. Reserve memory (use those pages as kernel objects)
15. Not available to app
16. Fragment allocator:
17. Add size request(100,200 bytes or kbs)
18. Virtual continuous address
19. Request size is large
20. Boot memory
21. Used for startup drivers



Slab:

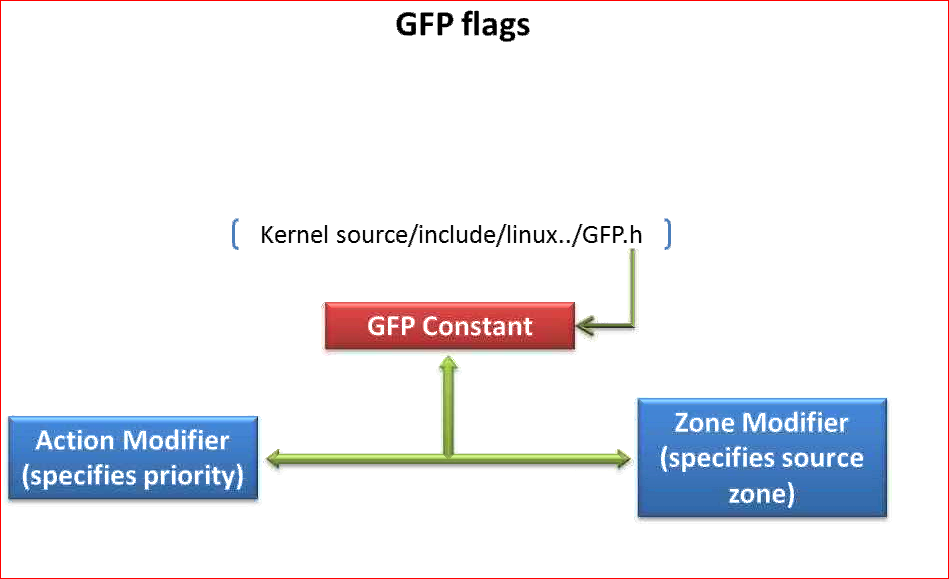
* Basic allocators, kernel equivalents of glibc malloc and free
* Static inline void \*kmalloc(size\_t size, int flags)

Size : Number of bytes to allocates

Flag: priority

* Void free(const void \*objp)

It will execute success , if allocate memory is success.



The kernel memory allocation function kmalloc is a powerful and high-speed (unless blocked) tool.

The allocated memory is continuous in physical memory and keeps the original data (not cleared).

Prototype:

#include <linux/slab.h>

void \* kmalloc (size\_t size, int flags);

      size parameter description: the physical memory of the kernel management system, physical memory can only be allocated by page.

kmalloc and the typical user space malloc are actually very different.

The kernel uses special page-based allocation techniques to utilize system RAM in the best way.

Linux handles memory allocation by creating a series of memory object collections, each of which has a fixed memory block size.

When processing an allocation request, an entire block is passed directly to the requester in a set that contains large enough memory blocks.

It must be noted that the kernel can only allocate some predefined, fixed-size byte arrays. The smallest memory block that kmalloc can handle is 32 or 64 bytes (architecture-dependent), and the upper limit of the memory block size varies with the system and kernel configuration.

Considering the portability, memory larger than 128 KB should not be allocated. If you need more than a few KB of memory blocks, it is best to use other methods.

Kmalloc flags:

**GPL\_KERNEL:** Normal Zone – Normal priority

Standard kernel memory allocation

May block .

GFP\_ATOMIC – Normal Zone,Highest Priority

Allocated Ram from interrupt handlers or code not triggered by user processes.

Never block

**GFP\_USER –** Normal zone –Low priority

Allocate memory for user processes

May block . Lowest priority

GFP\_DMA –Allocate in DMZ zone

GFP\_HIGHMEM – Allocate in high memory

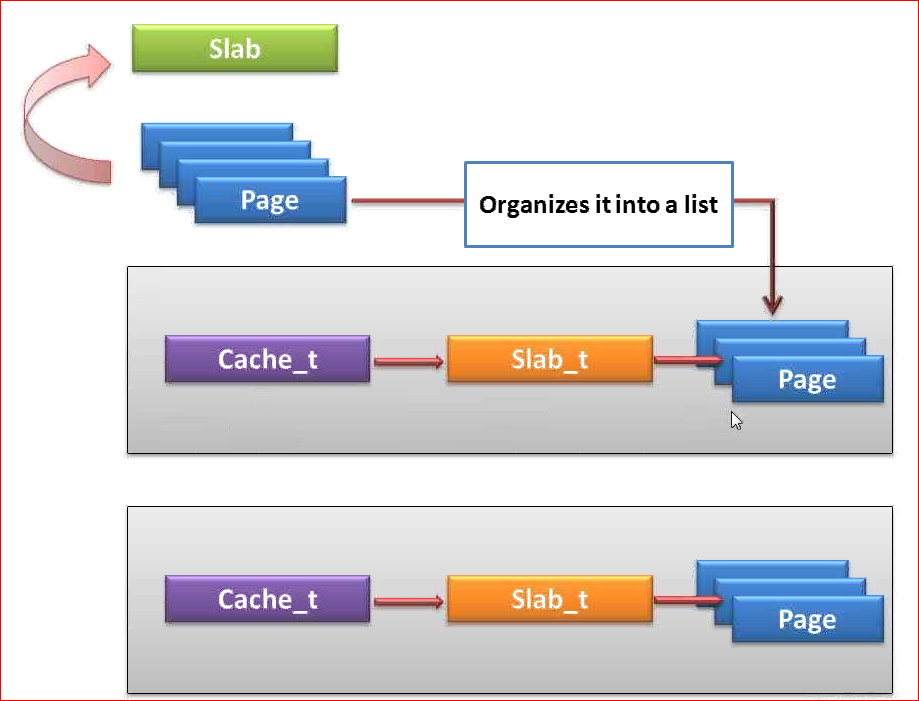
GFP\_REPEAT- Ask to try harder

GFP\_NOFAIL-Must not fail .never given up

GFP\_NORETRY-if allocation fails ,don’t try to get free page

Free :

Void kfree(const void \*obj);



At init time slab allocator acquire set of pages from page allocators ,these pages it orginised into individual data structure into slab list.

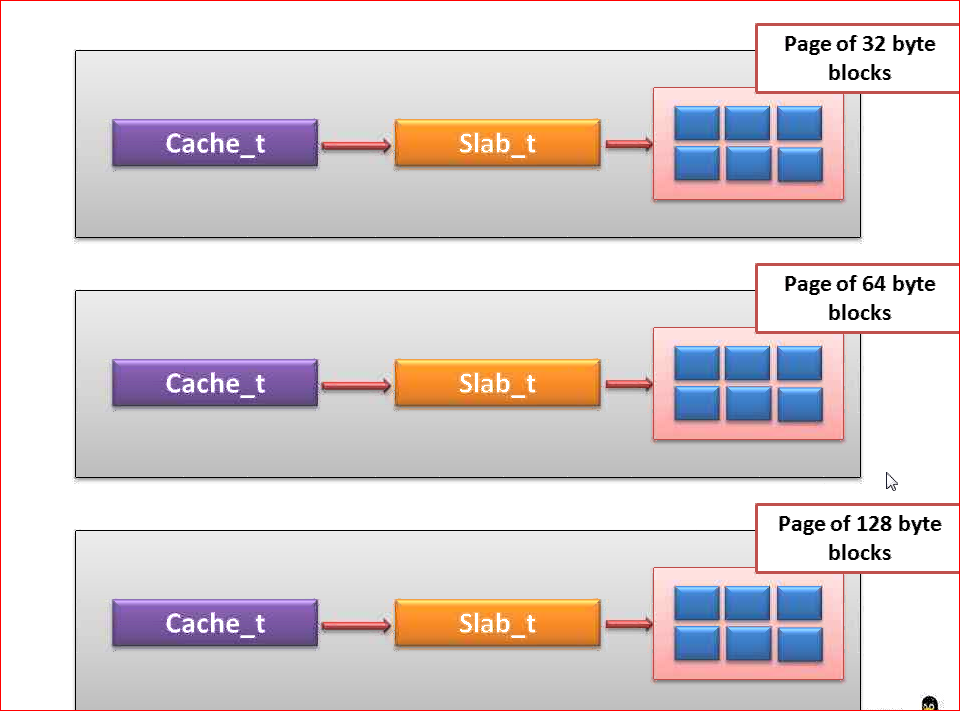
Each of the slab list contain instance of kmem\_cache\_t object.

Cache\_t ---🡪 slab\_t --🡪 pages

Each cache\_t point to a slab instance called slab\_t type, each page refer as N no of pages

Each slab\_t is refer as N number of pages .

Slab allocator organized pages in to list. Each list is assigned with a size the is (32 ,64,128)



Kmalloc:

* When Kmalloc request come then it will jump in to slap allocator and find the best fit and return the address.
* Kmalloc is very quick
* Kmalloc it return address of free block form the slab cache

Cat /proc/slabinfo

Header file:

#include <linux/slab.h>

**kmalloc:** It uses the generic slab caches available to any kernel code. so your module will share slab cache with other components in kernel.

**kmem\_cache\_alloc:** It will allocate objects from a dedicated slab cache created by kmem\_cache\_create. If you specifically want a better slab cache management dedicated to your module only, that too for a specific type of objects, use kmem\_cache\_create followed by kmem\_cache\_alloc. USB/SCSI drivers use this. kmem\_cache\_create takes sizeof your object you want to create slab of, a name which appears in /proc/slabinfo and flags to govern behavior of your slab cache.

Lookaside cache:

The cache manager of the Linux kernel is sometimes referred to as the "slab allocator". The cache implemented by the slab allocator is of

Type kmem\_cache\_t.

The corresponding functions are:  
      kmem\_cache\_t \*kmem\_cache\_create(const char \*name,

size\_t size,

size\_t offset,

unsigned long flags,

void (\* constructor) (void \*, kmem\_cache\_t \* unsigned long flags),

void (\* destructor) (void \*, kmem\_cache\_t \*, unsigned long flags));

/\* Create a cache object of the same size that can accommodate any number of memory areas\*/  
      Allocate objects from the created backup cache by calling kmem\_cache\_alloc:  
void \*kmem\_cache\_alloc(kmem\_cache\_t \*cache, int flags);  
/\* cache parameter is the newly created cache, flags are the same as kmalloc \*/  
      use kmem\_cache\_free to release an object:

void kmem\_cache\_free(kmem\_cache\_t \*cache, const void \*obj);  
      when the driver runs out of this backup cache (usually when the module is unloaded), release the cache:

int kmem\_cache\_destroy(kmem\_cache\_t \*cache);  
/\* only from this cache Only after all the assigned objects have returned has been successful. Therefore, the return value of kmem\_cache\_destroy should be checked: Failure indicates a memory leak in the module \*/

Example of kmem\_cache\_alloc.

Memory Pool :

Header file :

#inlcude <liunux/mempool.h>

Cache will reserve some pages and it will used by driver by using kmalloc.

Cache reserved some pages , we will allocate that memory when kernel service want/need.

Kmem\_cache\_alloc -> actual allocation when required

Kmem\_cache\_create-> reserve memory .

Mempool : memory pool will allocate instances on top of cache and reuse when needed .

We create cache and then allocate instance , when ever we needed we will pull instance and use it .

Example : n/w card , usb drivers etc.

Slab layer allows kernel service to create memory pools that can be used for pre allocation of specific object .

Memory pool: In order to ensure successful memory allocation when memory allocation does not allow failure, the kernel provides an abstraction called memory pool ("mempool"), which is actually some kind of backup cache. It is intended for use in emergency situations. Therefore, you must pay attention to it: mempool will allocate some memory blocks to make it idle and not really used, so it is easy to consume a lot of memory. And don't use mempool to handle failed allocations. Avoid using mempool in the driver code. The corresponding functions are:

**Createing memory pool :**

mempool\_t \*mem;

struct kmem\_cache \*cache\_ptr;

cache\_ptr=kmem\_cache\_create("cache\_mem1",12,0,SLAB\_HWCACHE\_ALIGN,NULL);

mem=mempool\_create(20,mempool\_alloc\_slab,mempool\_free\_slab,cache\_ptr);

1. we need it mention min no. of active object at any given point it should maintain 20 object
2. allocate routine
3. de-allocate routine

Acquiring object form pool :

ptr=(struct abc \*)mempool\_alloc(mem,GFP\_KERNEL);

Free:

mempool\_free(ptr,mem);

mempool\_destroy(mem);

Usage :

Suppose we want to maintain a pool of memory in a device driver or module. How can that pool be created and be available to multiple processes lets say 4 processes, accessing this driver/module.

Assume 1 MB of memory in the pool.

In order to create a memory pool, you need to use the **kernel's slab allocator**, or by maintaining the memory pool by yourself like what you did (kmalloc). By using **kernel's slab allocator**, you can use one of those:

* kmem\_cache\_create()
* mempool\_create()

I think the key problem for you to maintain a pool by yourself is a risk of creating memory fragmentation issue which will quickly run out of your memory or you can't allocate a large memory block even if there are lots of free memory.

Another benefit of using **kernel's slab allocator** is you can easily monitor the memory usage by looking into your /proc/slab entries.

## Platform Driver :

* 1. Platform devices are **inherently not discoverable**, i.e. the hardware cannot say "Hey! I'm present!" to the software. Typical examples are i2c devices, kernel/Documentation/i2c/instantiating-devices states:

Unlike PCI or USB devices, I2C devices are not enumerated at the hardware level (at run time). Instead, the software must know (at compile time) which devices are connected on each I2C bus segment. So USB and PCI are not platform devices

* 1. Platform devices are bound to drivers by matching names,
  2. Platform devices should be registered very early during system boot. Because they are often critical to the rest of the system (platform) and its drivers.

register a platform driver that will manage this device. It should define a unique name,

register your platform device, defining the same name as the driver.

* 1. Platform driver is for those devices that are on chip.

Not true (in theory, but true in practice). i2c devices are not onChip, but are platform devices because they are not discoverable. Also we can think of onChip devices which are normal devices. Example: an integrated PCI GPU chip on a modern x86 processor. It is discoverable, thus not a platform device.

* Adapter driver and bus driver enumerates devices when ecer they appear on system ex USB
* Devices on linux system are not always automatically discoverable eg i2c and spi
* Driver model introduce Platform devices to resolve this
* Platform devices are those who are not inherently at runtime
* Software must know about devices and resources .

## How to write a platform device driver :

* Register platform bus as a basic infrastructure.
* Register the device with unique name in linux kernel with platform bus
* Register a platform driver with platform bus , providing matching name or id\_table.
* Integrate driver in kernel device tree.

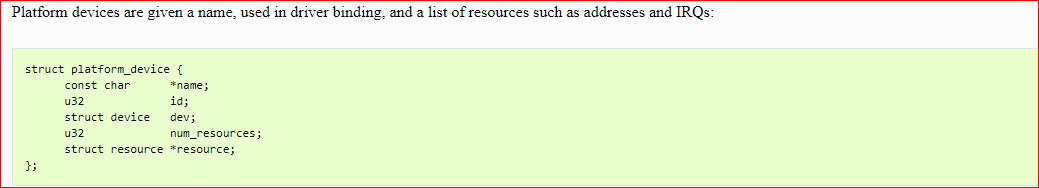
## Platform bus :

* Device driver model – all devices should be connected to a bus .
* Virtual bus

Drivers/base/platform.c

## Platform device :

Platform \_device Structure.



## Registering Platform device:

Two ,methods :

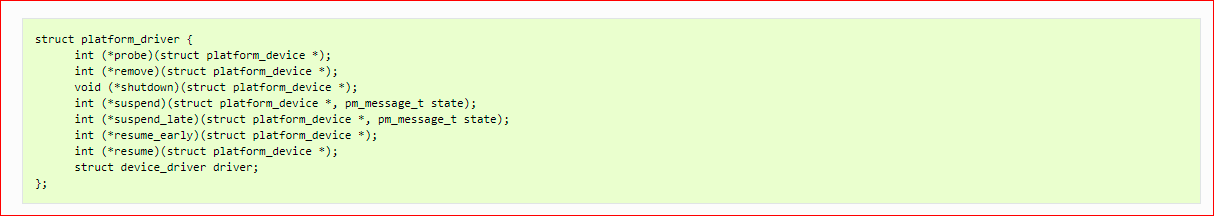
* 1. Using board file
  2. Using device tree
* Registering platform device using Board file:

 /[arch](http://androidxref.com/kernel_3.18/xref/arch/)/[mips](http://androidxref.com/kernel_3.18/xref/arch/mips/)/[mti-sead3](http://androidxref.com/kernel_3.18/xref/arch/mips/mti-sead3/)/[sead3-i2c.c](http://androidxref.com/kernel_3.18/xref/arch/mips/mti-sead3/sead3-i2c.c)

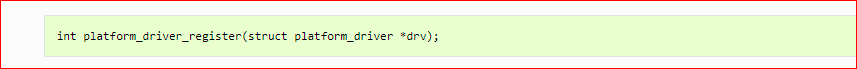


## Registering Platform driver:

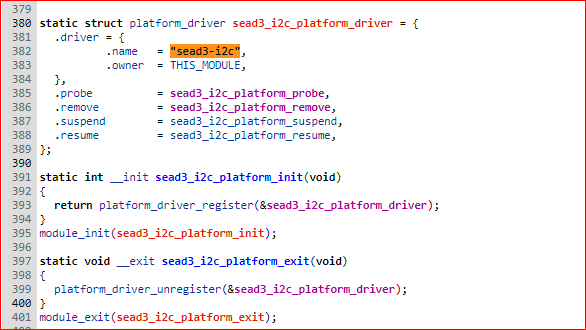
Platform drivers follow the standard driver model convention



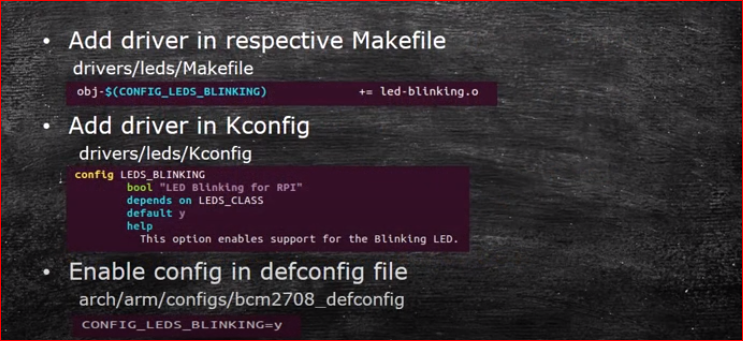
Platform drivers register themselves the normal way:



/[arch](http://androidxref.com/kernel_3.18/xref/arch/)/[mips](http://androidxref.com/kernel_3.18/xref/arch/mips/)/[mti-sead3](http://androidxref.com/kernel_3.18/xref/arch/mips/mti-sead3/)/[sead3-i2c-drv.c](http://androidxref.com/kernel_3.18/xref/arch/mips/mti-sead3/sead3-i2c-drv.c)



## Integrate driver in kernel source tree :



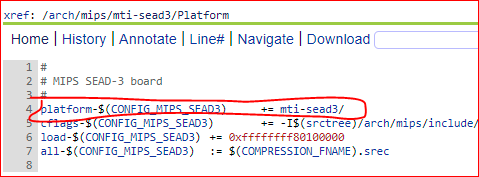
Example:

Adding Driver in respective Makefile

/[arch](http://androidxref.com/kernel_3.18/xref/arch/)/[mips](http://androidxref.com/kernel_3.18/xref/arch/mips/)/[mti-sead3](http://androidxref.com/kernel_3.18/xref/arch/mips/mti-sead3/)/[Makefile](http://androidxref.com/kernel_3.18/xref/arch/mips/mti-sead3/Makefile)

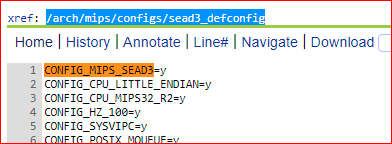


Add driver in kconfig.



Enable config in defconfig file :

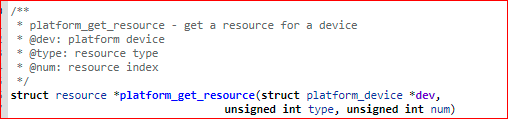
/[arch](http://androidxref.com/kernel_3.18/xref/arch/)/[mips](http://androidxref.com/kernel_3.18/xref/arch/mips/)/[configs](http://androidxref.com/kernel_3.18/xref/arch/mips/configs/)/[sead3\_defconfig](http://androidxref.com/kernel_3.18/xref/arch/mips/configs/sead3_defconfig)



## platform\_get\_resource

platform\_get\_resource() is used in the \_\_init function of a driver to get information on the structure of the device resource, like start address and end address, in order to find the resource memory size so you can map it in memory.

The declaration of platform\_get\_resource function is the following



The first parameter tells the function which device we are interested in, so it can extract the info we need.

The second parameter depends on what kind of resource you are handling. If it is memory( or anything that can be mapped as memory :-)) then it's IORESOURCE\_MEM. You can see all the macros at [**include/linux/ioport.h**](http://lxr.free-electrons.com/source/include/linux/ioport.h#L33)

The last parameter says which resource of that type is desired, with zero indicating the first one. Thus, for example, a driver could find its second MMIO region with:

r = platform\_get\_resource(pdev, IORESOURCE\_MEM, 1);

The return value is a pointer to a type struct resource var.

## Using device tree:

Refer example in Platform\_driver folder.