[Device Model 2](#_Toc443594266)

[Linux Boot Sequence 4](#_Toc443594267)

[Interrupts in Linux: 6](#_Toc443594268)

[Bottom-half Mechanisms in Linux: 9](#_Toc443594269)

[Linux Memory Management 10](#_Toc443594270)

[Kernel Synchronization: 13](#_Toc443594271)

[Process Management and Process Scheduling 19](#_Toc443594272)

[Timers and Time Management 21](#_Toc443594273)

[Power Management in Linux 22](#_Toc443594274)

[Linux Kernel Modules: 23](#_Toc443594275)

[Linux Kernel Debugging 25](#_Toc443594276)

[VFS: 26](#_Toc443594277)

[Miscellaneous 27](#_Toc443594278)

[Study Link 30](#_Toc443594279)

# Device Model

**Explain about the Linux Device Model (LDM)? Page 348 of Robert love**

Ans:<http://linuxkernel51.blogspot.in/2011/02/linux-driver-model.html>

The Linux Device model is built around the concept of busses, devices and drivers. All devices in the system are connected to a bus of some kind. The bus does not have to be a real one; busses primarily exist to gather similar devices together and coordinate initialization, shutdown and power management.

When a device in the system is found to match a driver, they are bound together. The specifics about how to match devices and drivers are bus-specific. The PCI bus, for example, compares the PCI Device ID of each device against a table of supported PCI IDs provided by the driver. The platform bus, on the other hand, simply compares the name of each device against the name of each driver; if they are the same, the device matches the driver.

Binding a device to a driver involves calling the driver’s probe () function passing a pointer to the device as a parameter. From this point on, it’s the responsibility of the driver to get the device properly initialized and register it with any appropriate subsystems.

Devices that can be hot-plugged must be un-bound from the driver when they areremoved from the system. This involves calling the driver’s remove () function passing a pointer to the device as a parameter. This also happens if the driver is a dynamically loadable module and the module is unloaded. All device driver callbacks, including probe () and remove (), must follow the returnvalue.

**Explain about about ksets, kobjects and ktypes. How are they related? Page 352 robert love**

<http://www.makelinux.net/ldd3/chp-14-sect-1>

Ans:The kobject is the fundamental structure that holds the device model together. A kobject, in other words, is of little interest on its own; it exists only to tie a higher-level object into the device model.It includes

Reference counting of objects Hotplug event handling Sysfs representation

kset is a collection of kobjects embedded within structures of the same type. However, while struct kobj\_type concerns itself with the type of an object, struct kset is concerned with aggregation and collection. The two concepts have been separated so that objects of identical type can appear in distinct sets.

A subsystem is a representation for a high-level portion of the kernel as a whole. Subsystems usually (but not always) show up at the top of the sysfs hierarchy. Some example subsystems in the kernel include block\_subsys (/sys/block, for block devices), devices\_subsys (/sys/devices, the core device hierarchy), and a specific subsystem for every bus type known to the kernel.A subsystem, thus, is really just a wrapper around a kset, with a semaphore thrown in.

Sysfs entries for kobjects are always directories, so a call to kobject\_add results in the creation of a directory in sysfs. Usually that directory contains one or more attributes; we see how attributes are specified shortly.The name assigned to the kobject (with kobject\_set\_name) is the name used for the sysfs directory. Thus, kobjects that appear in the same part of the sysfs hierarchy must have unique names. Names assigned to kobjects should also be reasonable file names: they cannot contain the slash character, and the use of white space is strongly discouraged.The sysfs entry is located in the directory corresponding to the kobject's parent pointer.

If parent is NULL when kobject\_add is called, it is set to the kobject embedded in the new kobject's kset; thus, the sysfs hierarchy usually matches the internal hierarchy created with ksets. If both parent and kset are NULL, the sysfs directory is created at the top level, which is almost certainly not what you want.

**Difference bw proc and sysfs?**

The /proc filesystem was originally designed to publish process information and a few key system attributes, required by 'ps', 'top', 'free' and a few other system utilities. However, because it was easy to use (both from the kernel side and the user-space side), it became a dumping ground for a whole range of system information. Also, it started to gain read/write files, to be used to adjust settings and control the operation of the kernel or its various subsystems. However, the methodology of implementing control interfaces was ad-hoc, and /proc soon grew into a tangled mess.

The sysfs (or /sys filesystem) was designed to add structure to this mess and provide a uniform way to expose system information and control points (settable system and driver attributes) to user-space from the kernel. Now, the driver framework in the kernel automatically creates directories under /sys when drivers are registered, based on the driver type and the values in their data structures. This means that drivers of a particular type will all have the same elements exposed via sysfs.

Many of the legacy system information and control points are still accessible in /proc, but all new busses and drivers should expose their info and control points via sysfs.

# Linux Boot Sequence

**Explain about the Linux boot sequence in case of ARM architecture?**

[**https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwjA3LCSkMrKAhUDxI4KHR2xBysQFggjMAE&url=http%3A%2F%2Fwww.linux-arm.org%2Fpub%2FLinuxPlatform%2FRealViewLink%2FBooting\_ARM\_Linux\_SMP\_on\_MPCore.doc&usg=AFQjCNG3zOT9-LZ52Ps1oKOKAME7OJveeg&sig2=lrl091uhySGqPjfw9Ngjaw**](https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwjA3LCSkMrKAhUDxI4KHR2xBysQFggjMAE&url=http%3A%2F%2Fwww.linux-arm.org%2Fpub%2FLinuxPlatform%2FRealViewLink%2FBooting_ARM_Linux_SMP_on_MPCore.doc&usg=AFQjCNG3zOT9-LZ52Ps1oKOKAME7OJveeg&sig2=lrl091uhySGqPjfw9Ngjaw)

[**https://www.kernel.org/doc/Documentation/arm/Booting**](https://www.kernel.org/doc/Documentation/arm/Booting)

[**https://blackfin.uclinux.org/doku.php?id=bootloaders:u-boot:debugging**](https://blackfin.uclinux.org/doku.php?id=bootloaders:u-boot:debugging)

[**http://www.tldp.org/HOWTO/BootPrompt-HOWTO-3.html**](http://www.tldp.org/HOWTO/BootPrompt-HOWTO-3.html)

**Explain about command line arguments that are passed to linux kernel and how/where they are parsed in kernel code?How are the command line arguments passed to Linux kernel by the u-boot (bootloader)?**

Linux (kernel) takes parameters when it boots up. This resembles an application started in a console with command line parameters. So we call kernel parameters "kernel command line", although kernel is not really started the same way as a command line application.

Linux kernel defines quite a few boot parameters. The kernel-parameters.txt file under .../Documentation has a complete list of all supported kernel parameters for the specific kernel version. Some examples are boot\_delay, console, initrd, nfsroot, etc. These parameters affect the way the kernel boots up and behaves when it's up running.

Linux is typically booted (run) by a bootloader. For some bootloaders such as u-boot, they are linux aware and provide mechanism to pass command line to linux when it boots up the kernel. But for others that are not Linux aware and no way is available to pass command line, the kernel command line could be hardcoded in source code and compiled into the kernel image.

u-boot passes kernel command line by bootargs environment variable. All the key=value pairs in bootargs are passed over as command line to kernel. So if you want to pass certain kernel parameters, you put them in the bootargs environment variable (note this is u-boot environment variable). When u-boot starts kernel code by bootm for example, it will pass the start address and end address of the command line string, and kernel takes it from there.

The bootloader must pass parameters to the kernel to describe the setup it has performed, the size and shape of memory in the system and, optionally, numerous other values.

<http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html>

The tagged list should conform to the following constraints

|  |
| --- |
| The list must be stored in RAM and placed in a region of memory where neither the kernel decompresser nor initrd manipulation will overwrite it. The recommended placement is in the first 16KiB of RAM, usually the start of physical RAM plus 0x100 (which avoids zero page exception vectors). |
| The physical address of the tagged list must be placed in R2 on entry to the kernel, however historically this has not been mandatory and the kernel has used the fixed value of the start of physical RAM plus 0x100. This must not be relied upon in the future. |
| The list must not extend past the 0x4000 boundary where the kernel's initial translation page table is created. The kernel performs no bounds checking and will overwrite the parameter list if it does so. |
| The list must be aligned to a word (32 bit, 4byte) boundary (if not using the recommended location) |
| The list must begin with an [ATAG\_CORE](http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#ATAG_CORE) and end with [ATAG\_NONE](http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#ATAG_NONE) |
| The list must contain at least one [ATAG\_MEM](http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#ATAG_MEM) |

**Explain about ATAGS? Vs device tree**

ATAGS are ARM tags. They are used to carry information such as memory size from boot code to kernel.

A device tree describes everything about the hardware which the kernel uses to select which drivers to load, where all the MMIO interfaces are, etc... at runtime. ATAGs just describes stuff like where to find an initrd and kernel parameters, memory, etc... - everything else about the machine is hard coded into the kernel. The preferred method now is to use device trees instead of ATAGs. One of the advantages include the fact that adding a new platform doesn't always require adding new code to the kernel.

**Explain device tree concepts in linux.**

In order to remove same code across different board the device trees are used. These are structure that defines the available devices on a platform. These are created from a board definitions file to a binary file which is loaded in memory by the boot loader. The kernel populates the device tree structure based on this binary. Thus different boards can be supported without recompiling the kernel

# Interrupts in Linux:

**Explain about the interrupt mechanims in linux?**

<http://www.tldp.org/LDP/tlk/dd/interrupts.html>

<http://linuxburps.blogspot.in/2013/09/linux-kernel-interrupts-and-handlers.html>

<https://www.safaribooksonline.com/library/view/understanding-the-linux/0596005652/ch04s06.html>

**How do you find unhanded interrupts?**

**What are the APIs that are used to register an interrupt handler?**

[**http://www.makelinux.net/ldd3/chp-10-sect-2**](http://www.makelinux.net/ldd3/chp-10-sect-2)

int request\_irq(unsigned int irq,

irqreturn\_t (\*handler)(int, void \*, struct pt\_regs \*),

unsigned long flags,

const char \*dev\_name,

void \*dev\_id);

request\_threaded\_irq 🡺

void free\_irq(unsigned int irq, void \*dev\_id);

**How do you register an interrupt handler on a shared IRQ line?**

<http://www.makelinux.net/books/lkd2/ch06lev1sec3>

A shared handler is registered and executed much like a non-shared handler. There are three main differences:

* The SA\_SHIRQ flag must be set in the flags argument to request\_irq().
* The dev\_id argument must be unique to each registered handler. A pointer to any per-device structure is sufficient; a common choice is the device structure as it is both unique and potentially useful to the handler. You cannot pass NULL for a shared handler!
* The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt. This requires both hardware support and associated logic in the interrupt handler. If the hardware did not offer this capability, there would be no way for the interrupt handler to know whether its associated device or some other device sharing the line caused the interrupt.

All drivers sharing the interrupt line must meet the previous requirements. If any one device does not share fairly, none can share the line. When request\_irq() is called with SA\_SHIRQspecified, the call succeeds only if the interrupt line is currently not registered, or if all registered handlers on the line also specified SA\_SHIRQ. Note that in 2.6, unlike the behavior in older kernels, shared handlers can mix usage of SA\_INTERRUPT.

When the kernel receives an interrupt, it invokes sequentially each registered handler on the line. Therefore, it is important that the handler be capable of distinguishing whether it generated a given interrupt. The handler must quickly exit if its associated device did not generate the interrupt. This requires the hardware device to have a status register (or similar mechanism) that the handler can check. Most hardware does indeed have such a feature.

**Explain about the flags that are passed to request\_irq().**

* SA\_INTERRUPT This flag specifies that the given interrupt handler is a fast interrupt handler. Historically, Linux differentiated between interrupt handlers that were fast versus slow. Fast handlers were assumed to execute quickly, but potentially very often, so the behavior of the interrupt handling was modified to enable them to execute as quickly as possible. Today, there is only one difference: Fast interrupt handlers run with all interrupts disabled on the local processor. This enables a fast handler to complete quickly, without possible interruption from other interrupts. By default (without this flag), all interrupts are enabled except the interrupt lines of any running handlers, which are masked out on all processors. Sans the timer interrupt, most interrupts do not want to enable this flag.
* SA\_SAMPLE\_RANDOM This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do not set this if your device issues interrupts at a predictable rate (for example, the system timer) or can be influenced by external attackers (for example, a networking device). On the other hand, most other hardware generates interrupts at nondeterministic times and is, therefore, a good source of entropy. For more information on the kernel entropy pool, see [Appendix B](http://www.makelinux.net/books/lkd2/app02.html#app02), "Kernel Random Number Generator."
* SA\_SHIRQ This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line. More information on shared handlers is provided in a following section.

**Explain interrupt sequence in detail starting from ARM to registered interrupt handler.**

**Explain about the internals of Interrupt handling in case of Linux running on ARM?**

<http://venkateshabbarapu.blogspot.in/2012/09/interrupt-handling-in-arm.html>

<http://linuxburps.blogspot.in/2013/10/linux-interrupt-handling.html>

<http://lxr.free-electrons.com/source/kernel/irq/handle.c#L135>

<http://learnlinuxconcepts.blogspot.in/2014/02/interrupts.html>

<http://arm-linux-interview-questions.blogspot.in/2013/04/arm-linux-abort-handler-implementation.html>

**What are the precautions to be taken while writing an interrupt handler?**

<http://www.embeddedlinux.org.cn/essentiallinuxdevicedrivers/final/ch04lev1sec2.html>

1. Interrupt handlers cannot relinquish the processor by calling sleepy functions such as schedule\_timeout(). Before invoking a kernel API from your interrupt handler, penetrate the nested invocation train and ensure that it does not internally trigger a blocking wait. For example, input\_register\_device() looks harmless from the surface, but tosses a call tokmalloc() under the hood specifying GFP\_KERNEL as an argument. As you saw in [Chapter 2](http://www.embeddedlinux.org.cn/essentiallinuxdevicedrivers/final/ch02.html#ch02), "A Peek Inside the Kernel," if your system's free memory dips below a watermark, kmalloc() sleep-waits for memory to get freed up by the swapper, if you invoke it in this manner.
2. For protecting critical sections inside interrupt handlers, you can't use mutexes because they may go to sleep. Use spinlocks instead, and use them only if you must.
3. Interrupt handlers cannot directly exchange data with user space because they are not connected to user land via process contexts. This brings us to another reason why interrupt handlers cannot sleep: The scheduler works at the granularity of processes, so if interrupt handlers sleep and are scheduled out, how can they be put back into the run queue?
4. Interrupt handlers are supposed to get out of the way quickly but are expected to get the job done. To circumvent this Catch-22, interrupt handlers usually split their work into two. The slim top half of the handler flags an acknowledgment claiming that it has serviced the interrupt but, in reality, offloads all the hard work to a fat bottom half. Execution of the bottom half is deferred to a later point in time when all interrupts are enabled. You will learn to develop bottom halves while discussing softirqs and tasklets later.
5. You need not design interrupt handlers to be reentrant. When an interrupt handler is running, the corresponding IRQ is disabled until the handler returns. So, unlike process context code, different instances of the same handler will not run simultaneously on multiple processors.
6. Interrupt handlers can be interrupted by handlers associated with IRQs that have higher priority. You can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as afast handler. Fast handlers run with all interrupts disabled on the local processor. Before disabling interrupts or labeling your interrupt handler as fast, be aware that interrupt-off times are bad for system performance. More the interrupt-off times, more is the interrupt latency, or the delay before a generated interrupt is serviced. Interrupt latency is inversely proportional to the real time responsiveness of the system.

**What is bottom half and top half.**

* There are two goals that an interrupt handler needs to perform 1. execute quickly and 2. perform a large amount of work .Because of these conflicting goals, the processing of interrupts is split into two parts, or halves.The interrupt handler is the top half. It is run immediately upon receipt of the interrupt and performs only the work that is time critical, such as acknowledging receipt of the interrupt or resetting the hardware.
* Work that can be performed later is delayed until the bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled. Let us consider a case where we need to collect the data form a data card and then process it.The most important job is to collect the data from data card to the memory and free the card for incoming data and this is done in top half.The rest part which deals with the processing of data is done in the bottom half

**What is request\_threaded\_irq() ?**

<https://lwn.net/Articles/302043/>🡺 api params got switched as explained below

int request\_threaded\_irq(unsigned int irq, irq\_handler\_t handler,   
irq\_handler\_t thread\_fn, unsigned long irqflags,   
const char \*devname, void \*dev\_id)   
  
Note the presents of \*two\* handlers, called "handler" and "thread\_fn". The first, "handler", is called in interrupt context; it's job is usually to quiet the device and return; it cannot sleep. If it's return value is   
IRQ\_WAKE\_THREAD, the thread\_fn() will be called in process context; it \*can\* sleep. In the example you cite, there is no immediate handler, only the thread\_fn(); the call to a blocking function from within the thread\_fn() is correct.

Some driver code uses request\_threaded\_irq() with NULL as a value for handler, in that scenario kernel will invoke the default handler, which simply wakeup the thread function.

**When to use request\_threaded\_irq instead of bottom halves ?**

Answer lies in the driver's requirement, if it wants to sleep put the code in thread fn and user threaded function.

**If same interrupts occurs in two cpu how are they handled?**

[**http://stackoverflow.com/questions/10235011/how-are-interrupts-handled-on-smp**](http://stackoverflow.com/questions/10235011/how-are-interrupts-handled-on-smp)

**How to synchronize data between 'two interrupts' and 'interrupts and process'.**

[**http://stackoverflow.com/questions/12907784/synchronization-between-user-space-process-and-interupt-context-code**](http://stackoverflow.com/questions/12907784/synchronization-between-user-space-process-and-interupt-context-code)

**How are nested interrupts handled?**

A reentrant interrupt handler must therefore take the following steps after an IRQ exception is raised and control is transferred to the interrupt handler in the way previously described.

• The interrupt handler saves the context of the interrupted program (that is, it pushes onto the alternative kernel mode stack any registers which will be corrupted by the handler, including the return address and SPSR\_IRQ).

Q> What is the alternative kernel mode stack here?

• It determines which interrupt source needs to be processed and clears the source in the external hardware (preventing it from immediately triggering another interrupt).

• The interrupt handler changes the processor to the other kernel mode, leaving the CPSR I bit set (interrupts are still disabled).

Q> From IRQ to SVC mode with CPSR.I =1 . Right?

• The interrupt handler saves the exception return address on the stack (a stack for the new mode, located in kernel memory) and re-enables interrupts.

Q> Are there 2 stacks here?

• It calls the appropriate C handler for the original interrupt (interrupts are still disabled).

• Upon completion, the interrupt handler disables IRQ and pops the exception return address from the stack.

• It restores the context of the interrupted program directly from the alternative kernel mode stack. This includes restoring the PC, and the CPSR which switches back to the previous execution mode.

Q> How is the nesting done here ? I am bit confused here...

1) Up to you, really. The requirement is that it is one that cannot be asynchronously invoked. So you can use System mode stack, which is shared with User mode - with some interesting implications. Or you can use the Supervisor mode stack, as long as you always properly store all context before executing an SVC instruction.

2) Yes.

3) Yes, you store the context on a stack for whichever mode picked in (1).

4) While executing in the alternative mode, you re-enable interrupts (as your text states). At this point, the processor will now react to new interrupts signaled to the core - generally ones of a higher priority as configured in your interrupt controller.

**How is task context saved during interrupt?**

[**http://electronics.stackexchange.com/questions/137895/what-is-the-difference-between-context-switching-and-interrupt-handling**](http://electronics.stackexchange.com/questions/137895/what-is-the-difference-between-context-switching-and-interrupt-handling)

**Tell me about interrupt handling mechanism in linux**  
**How to allocate memory in interrupt handler?**

GFP\_ATOMIC flag is the only option when the current code is unable to sleep, such as with interrupt handlers, softirqs, and tasklets.

**How to modify interrupt vector  
Who assigns interrupt numbers  
How interrupt handler is executed  
how interrupts are raised in the hypervisors?**

# Bottom-half Mechanisms in Linux:

What are the different bottom-half mechanisms in Linux?

Softirq, Tasklet and Workqueues

<http://linuxburps.blogspot.in/2013/09/linux-kernel-interrupts-and-handlers.html#!/2013/09/linux-kernel-interrupts-and-handlers.html>

**What are the tasklets? Can two drivers can use same tasklets**  
**What are the differences between Softirq/Tasklet and Workqueue? Given an example what you prefer to use?  
What are the differences between softirqs and tasklets?**

Softirq is guaranteed to run on the CPU it was scheduled on, where as tasklets don’t have that guarantee.   
The same tasklet can't run on two separate CPUs at the same time, where as a softirq can. 

When are these bottom halfs executed?

<https://www.safaribooksonline.com/library/view/understanding-the-linux/0596002130/ch04s07.html> very old

Explain about the internal implementation of softirqs?  
[Bottom-halves in Linux - Part 1: Softirqs](http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-1-softirqs.html)  
Explain about the internal implementation of tasklets?  
[Bottom-halves in Linux - Part 2: Tasklets](http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-2-tasklets.html)  
Explain about the internal implementation of workqueues?  
[Bottom-halves in Linux - Part 3: Workqueues](http://linuxblore.blogspot.in/2013/01/workqueues-in-linux.html)  
Explain about the concurrent work queues.

What is difference between Tasklets and work queues?

# 

# Linux Memory Management

**how to manage memory in linux kernel?**

[**http://www.tldp.org/LDP/khg/HyperNews/get/memory/linuxmm.html**](http://www.tldp.org/LDP/khg/HyperNews/get/memory/linuxmm.html)

[**https://www.kernel.org/doc/gorman/html/understand/understand006.html**](https://www.kernel.org/doc/gorman/html/understand/understand006.html)

[**http://lists.kernelnewbies.org/pipermail/kernelnewbies/2013-March/007697.html**](http://lists.kernelnewbies.org/pipermail/kernelnewbies/2013-March/007697.html)

A program’s VMAs are stored in its memory descriptor both as a linked list in the mmap field, ordered by starting virtual address, and as a red-black tree rooted at the mm\_rb field. The red-black tree allows the kernel to search quickly for the memory area covering a given virtual address. When you read file /proc/pid\_of\_process/maps, the kernel is simply going through the linked list of VMAs for the process and printing each one.

<http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/>

<http://www.win.tue.nl/~aeb/linux/lk/lk-9.html>

**What are the differences between vmalloc and kmalloc? Which is preferred to use in device drivers?**

Yes, physically contiguous memory is not required in many of the cases. Main reason for kmalloc being used more than vmalloc in kernel is performance. The book explains, when big memory chunks are allocated using vmalloc, kernel has to map the physically non-contiguous chunks (pages) into a single contiguous virtual memory region. Since the memory is virtually contiguous and physically non-contiguous, several virtual-to-physical address mappings will have to be added to the page table. And in the worst case, there will be *(size of buffer/page size)* number of mappings added to the page table.This also adds pressure on TLB (the cache entries storing recent virtual to physical address mappings) when accessing this buffer. This can lead to [thrashing](http://en.wikipedia.org/wiki/Thrashing_%28computer_science%29).

One of other differences is kmalloc will return logical address (else you specify GPF\_HIGHMEM). Logical addresses are placed in "low memory" (in the first gigabyte of physical memory) and are mapped directly to physical addresses (use \_\_pa macro to convert it). This property implies kmalloced memory is continuous memory.

In other hand, Vmalloc is able to return virtual addresses from "high memory". These addresses cannot be converted in physical addresses in a direct fashion (you have to use virt\_to\_page function).

<http://learnlinuxconcepts.blogspot.in/2014/02/linux-memory-management.html>

**What are the differences between slab allocator and slub allocator?**

[**http://events.linuxfoundation.org/sites/events/files/slides/slaballocators.pdf**](http://events.linuxfoundation.org/sites/events/files/slides/slaballocators.pdf)

**What is boot memory allocator?**

[**https://www.kernel.org/doc/gorman/html/understand/understand008.html**](https://www.kernel.org/doc/gorman/html/understand/understand008.html)

**How do you reserve block of memory?**

If you really need a huge buffer of physically contiguous memory, the best approach is often to allocate it by requesting memory at boot time. Allocation at boot time is the only way to retrieve consecutive memory pages while bypassing the limits imposed by \_\_get\_free\_pages on the buffer size, both in terms of maximum allowed size and limited choice of sizes. Allocating memory at boot time is a “dirty” technique, because it bypasses all memory management policies by reserving a private memory pool. This technique is inelegant and inflexible, but it is also the least prone to failure. Needless to say, a module can’t allocate memory at boot time; only drivers directly linked to the kernel can do that.

**Or use bigphy area.**

**What is virtual memory and what are the advantages of using virtual memory?**

**What's paging and swapping?**

**Is it better to enable swapping in embedded systems? and why?**

**What is the page size in Linux kernel in case of 32-bit ARM architecture?**

**What is page frame?**

**What are the different memory zones and why does different zones exist?**

**What is high memory and when is it needed?**

**Why is high memory zone not needed in case of 64-bit machine?**

**How to allocate a page frame from high memory?**

**In ARM, an abort exception if generated, if the page table doesn't contain a virtual to physical map for a particular page. How exactly does the MMU know that a virtual to physical map is present in the pagetable or not?**

**A Level-1 page table entry can be one of four possible types. The 1st type is given below:   
A fault entry that generates an abort exception. This can be either a prefetch or data abort, depending on the type of access. This effectively indicates virtual addresses that are unmapped.  
In this case the bit [0] and [1] are set to 0. This is how the MMU identifies that it's a fault entry.  
Same is the case with Level-2 page table entry.  
Does the Translation Table Base Address (TTBR) register, Level 1 page table and Level 2 page table contain Physical addresses or Virtual addresses?**  
TTBR: Contain physical address of the pgd base  
Level 1 page table (pgd): Physical address pointing to the pte base  
Level 2 page table (pte): Physical address pointing to the physical page frame  
Since page tables are in kernel space and kernel virtual memory is mapped directly to RAM. Using just an easy macro like \_\_virt\_to\_phys(), we can get the physical address for the pgd base or pte base or pte entry.

**DMA controller.**

<https://lwn.net/Articles/234617/>

<http://www.makelinux.net/ldd3/chp-15-sect-4>

<https://geidav.wordpress.com/2014/04/27/an-overview-of-direct-memory-access/>

<http://stackoverflow.com/questions/28571598/coherently-understand-the-software-hardware-interaction-with-regard-to-dma-and-b>

**Cache coherency.- MESI /MSI protocol**

**Cache coherency mechanism.**

<http://www.tldp.org/LDP/khg/HyperNews/get/memory/flush.html>

<http://renjucnair.blogspot.in/2012/01/writing-i2c-client-driver.html#!/2012/01/writing-i2c-client-driver.html>

[**What happens in the kernel during malloc?**](http://stackoverflow.com/questions/5716100/what-happens-in-the-kernel-during-malloc)

When user space applications call malloc(), that call isn't implemented in the kernel. Instead, it's a library call (implemented glibc or similar).

The short version is that the malloc implementation in glibc either obtains memory from the brk()/sbrk() system call or anonymous memory via mmap(). This gives glibc a big contiguous (regarding virtual memory addresses) chunk of memory, which the malloc implementation further slices and dices in smaller chunks and hands out to your application.

[Here](http://www.ibm.com/developerworks/linux/library/l-memory/)'s a small malloc implementation that'll give you the idea, along with many, many links.

Note that nothing cares about physical memory yet -- that's handled by the kernel virtual memory system when the process data segment is altered via brk()/sbrk() or mmap(), and when the memory is referenced (by a read or write to the memory).

To summarize:

1. malloc() willsearchitsmanagedpieces of memory to see if there's a piece of unused memory thatsatisfy the allocation requirements.
2. Failingthat, malloc() willtry to extend the process data segment(via sbrk()/brk() or in some cases mmap()). sbrk() ends up in the kernel.
3. The brk()/sbrk() calls in the kerneladjustsome of the offsets in the structmm\_struct of the process, so the process data segment willbelarger. At first, therewillbe no physical memory mapped to the additionalvirtualaddresseswhichextending the data segment gave.
4. Whenthatunmapped memory is first touched (likely a read/write by the malloc implementation) a faulthandlerwill kick in and trap down to the kernel, where the kernelwillassignphysical memory to the unmapped memory.

<http://www.righthandtech.com/embedded-linux-managing-memory.php>

**How virtual address is converted to physical address?  
what is the difference slab , slub memory algorithm?  
Explain memory thrashing?**

# Kernel Synchronization:

**What is spinlock and what are different types of spinlocks?  
Why do we need synchronization mechanisms in Linux kernel?  
What are the different synchronization mechanisms present in Linux kernel?**

for simple counter variables or for bitwise ------->atomic operations are best methods.   
atomic\_t count=ATOMIC\_INIT(0); or atomic\_set(&count,0);   
atomic\_read(&count);   
atomic\_inc(&count);   
atomic\_dec(&count);   
atomic\_add(&count,10);   
atomic\_sub(&count,10);   
  
spinlocks are used to hold critical section for short time and can use from interrupt context and locks can notsleep,also called busy wait loops.   
fully spinlocks and reader/writer spin locks are available.   
spinlock\_tmy\_spinlock;   
spin\_lock\_init( &my\_spinlock );   
spin\_lock( &my\_spinlock );   
  
// critical section   
  
spin\_unlock( &my\_spinlock );   
  
Spinlock variant with local CPU interrupt disable   
spin\_lock\_irqsave( &my\_spinlock, flags );   
  
// critical section   
  
spin\_unlock\_irqrestore( &my\_spinlock, flags );   
if your kernel thread shares data with a bottom half,   
spin\_lock\_bh( &my\_spinlock );   
  
// critical section   
  
spin\_unlock\_bh( &my\_spinlock );   
if you have more readers than writers for your shared resource   
Reader/writer spinlock can be used 

rwlock\_try\_rwlock;   
rwlock\_init( &my\_rwlock );   
write\_lock( &my\_rwlock );   
  
// critical section -- can read and write   
write\_unlock( &my\_rwlock );   
read\_lock( &my\_rwlock );   
// critical section -- can read only   
read\_unlock( &my\_rwlock );   
  
mutexs are used when you hold lock for longer time and if you use from process context.   
DEFINE\_MUTEX(my\_mutex );   
mutex\_lock( &my\_mutex );   
mutex\_unlock( &my\_mutex );

**How function pointers are shared across different processes? Using which iPCs?**

Two processes cannot share function pointers.   
if you want to use functions in two processes make library for that functions   
and use that library in your processes

Both process has a different virtual address space. If you send address of a function from process A to process B, in process B address from A does not make any sense. The memory mapping in process A will be different from that in process B. If you want to do so u need to know the memory mapping of process A and according to that you need to deference the memory in process B.

<http://ubuntuforums.org/archive/index.php/t-1671019.html>

The pointer has to actually point to somewhere in a shared memory segment, and both processes have to map that segment. Then the sending process has to subtract the base address of its mapping, and the receiving process has to add the base address of its mapping.  
  
Also you need to make sure that both processes don't try to access the object at the same time, like with a mutex.

[**Difference between binary semaphore and mutex**](http://stackoverflow.com/questions/62814/difference-between-binary-semaphore-and-mutex) **??**

<http://james.bond.edu.au/courses/inft73626@033/Assigs/Papers/kernel_locking_techniques.html>

They are NOT the same thing. They are used for different purposes!  
While both types of semaphores have a full/empty state and use the same API, their usage is very different.

Mutual Exclusion Semaphores  
Mutual Exclusion semaphores are used to protect shared resources (data structure, file, etc..).

A Mutex semaphore is "owned" by the task that takes it. If Task B attempts to semGive a mutex currently held by Task A, Task B's call will return an error and fail.

Mutexes always use the following sequence:

- SemTake

- Critical Section

- SemGive

Here is a simple example:

Thread A Thread B

Take Mutex

access data

... Take Mutex<== Will block

...

Give Mutex access data <== Unblocks

...

Give Mutex

Binary Semaphore  
Binary Semaphore address a totally different question:

* Task B is pended waiting for something to happen (a sensor being tripped for example).
* Sensor Trips and an Interrupt Service Routine runs. It needs to notify a task of the trip.
* Task B should run and take appropriate actions for the sensor trip. Then go back to waiting.

Task A Task B

... Take BinSemaphore<== wait for something

Do Something Noteworthy

Give BinSemaphore do something <== unblocks

Note that with a binary semaphore, it is OK for B to take the semaphore and A to give it.  
Again, a binary semaphore is NOT protecting a resource from access. The act of Giving and Taking a semaphore are fundamentally decoupled.  
It typically makes little sense for the same task to so a give and a take on the same binary semaphore

Do youusedsemaphorewhilecoding? explain scenario ??

YesBinarysemaphoremost of the time in protecting control block of device driver and in init and termfunction

Of devicedriver.STOS\_SemaphoreWait=>down\_interruptible and in OS21 \_semaphore\_wait

In timertaskthatis to synpurpousewe use guardsemaphore

But if i amusingit for critical section protection then i willwewaiting and signaling in the samefunction.

mutex\_init(mutex);

voidmutex\_lock(structmutex \*lock);

intmutex\_lock\_interruptible(structmutex \*lock);

intmutex\_trylock(structmutex \*lock);

voidmutex\_unlock(structmutex \*lock);

intmutex\_is\_locked(structmutex \*lock);

voidmutex\_lock\_nested(structmutex \*lock, unsignedintsubclass);

intmutex\_lock\_interruptible\_nested(structmutex \*lock,

unsignedintsubclass);

intatomic\_dec\_and\_mutex\_lock(atomic\_t \*cnt, structmutex \*lock);

/\*\*

\* mutex\_lock - acquire the mutex

\* @lock: the mutex to be acquired

\*

\* Lock the mutex exclusively for this task. If the mutex is not

\* available right now, it will sleep until it can get it.

\*

\* The mutex must later on be released by the same task that

\* acquired it. Recursive locking is not allowed. The task

\* may not exit without first unlocking the mutex. Also, kernel

\* memory where the mutex resides mutex must not be freed with

\* themutex still locked. The mutex must first be initialized

\* (or statically defined) before it can be locked. memset()-ing

\* themutex to 0 is not allowed.

\*

\* ( The CONFIG\_DEBUG\_MUTEXES .config option turns on debugging

\* checks that will enforce the restrictions and will also do

\* deadlock debugging. )

\*

\* This function is similar to (but not equivalent to) down().

\*/

/\*\*

\* mutex\_unlock - release the mutex

\* @lock: the mutex to be released

\*

\* Unlock a mutex that has been locked by this task previously.

\*

\* This function must not be used in interrupt context. Unlocking

\* of a not locked mutex is not allowed.

\*

\* This function is similar to (but not equivalent to) up().

\*/

/\*\*

\* mutex\_lock\_interruptible - acquire the mutex, interruptible

\* @lock: the mutex to be acquired

\*

\* Lock the mutex like mutex\_lock(), and return 0 if the mutex has

\* been acquired or sleep until the mutex becomes available. If a

\* signal arrives while waiting for the lock then this function

\* returns -EINTR.

\*

\* This function is similar to (but not equivalent to) down\_interruptible().

[spin\_can\_lock](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#373)  
[spin\_is\_contended](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#368)  
[spin\_is\_locked](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#363)  
[spin\_lock](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#283)  
[spin\_lock\_bh](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#288)  
[spin\_lock\_irq](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#308)  
[spin\_trylock](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#293)  
[spin\_trylock\_bh](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#343)  
[spin\_trylock\_irq](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#348)  
[spin\_unlock](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#323)  
[spin\_unlock\_bh](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#328)  
[spin\_unlock\_irq](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#333)  
[spin\_unlock\_irqrestore](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#338)  
[spin\_unlock\_wait](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#358)  
[spinlock\_check](http://updopengrok.gnb.st.com:8888/source/xref/kernel3/include/linux/spinlock.h#272)

1. preempt\_disable() doesn't disable IRQ. It just increases a thread\_info->preempt\_count variable
2. Disabling interrupts also disables preemption because scheduler isn't working after that - but only on a single-CPU machine. On the SMP it isn't enough because when you close the interrupts on one CPU the other / others still does / do something asynchronously.
3. The Big Lock (means - closing all interrupts on all CPUs) is slowing the system down dramatically - so it is why it not anymore in use. This is also the reason why preempt\_disable() doesn't close the IRQ.

You can see what is preempt\_disable(). Try this: 1. Get a spinlock. 2. Call schedule()

In the dmesg you will see something like "BUG: scheduling while atomic". This happens when scheduler detects that your process in atomic (not preemptive) context but it schedules itself.

If the scheduleriscalledfrom an ISR, everythingshouldbe the same. The schedulershould change the context and return to the ISR and the ISR shouldthen return using IRET. It will return to a differentprocess/thread if the schedulerchooses to switch to adifferent one and thereforeloadsitscontext and saves the old one.

**What are the differences between spinlock and mutex?**

<http://www.linux.com/community/forums/kernel-development/in-depth-explanation-of-spinlock-semaphore-mutex>

**The Theory**

In theory, when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. When a thread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

**The Problem**

The problem with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time. If now the mutex was only locked for a very short amount of time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread has actually slept by far and it might even exceed the time the thread would have wasted by constantly polling on a spinlock. On the other hand, polling on a spinlock will constantly waste CPU time and if the lock is held for a longer amount of time, this will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.

**The Solution**

Using spinlocks on a single-core/single-CPU system makes usually no sense, since as long as the spinlock polling is blocking the only available CPU core, no other thread can run and since no other thread can run, the lock won't be unlocked either. IOW, a spinlock wastes only CPU time on those systems for no real benefit. If the thread was put to sleep instead, another thread could have ran at once, possibly unlocking the lock and then allowing the first thread to continue processing, once it woke up again.

On a multi-core/multi-CPU systems, with plenty of locks that are held for a very short amount of time only, the time wasted for constantly putting threads to sleep and waking them up again might decrease runtime performance noticeably. When using spinlocks instead, threads get the chance to take advantage of their full runtime quantum (always only blocking for a very short time period, but then immediately continue their work), leading to much higher processing throughput.

**The Practice**

Since very often programmers cannot know in advance if mutexes or spinlocks will be better (e.g. because the number of CPU cores of the target architecture is unknown), nor can operating systems know if a certain piece of code has been optimized for single-core or multi-core environments, most systems don't strictly distinguish between mutexes and spinlocks. In fact, most modern operating systems have hybrid mutexes and hybrid spinlocks. What does that actually mean?

A hybrid mutex behaves like a spinlock at first on a multi-core system. If a thread cannot lock the mutex, it won't be put to sleep immediately, since the mutex might get unlocked pretty soon, so instead the mutex will first behave exactly like a spinlock. Only if the lock has still not been obtained after a certain amount of time (or retries or any other measuring factor), the thread is really put to sleep. If the same system runs on a system with only a single core, the mutex will not spinlock, though, as, see above, that would not be beneficial.

A hybrid spinlock behaves like a normal spinlock at first, but to avoid wasting too much CPU time, it may have a back-off strategy. It will usually not put the thread to sleep (since you don't want that to happen when using a spinlock), but it may decide to stop the thread (either immediately or after a certain amount of time) and allow another thread to run, thus increasing chances that the spinlock is unlocked (a pure thread switch is usually less expensive than one that involves putting a thread to sleep and waking it up again later on, though not by far).

**Summary**

If in doubt, use mutexes, they are usually the better choice and most modern systems will allow them to spinlock for a very short amount of time, if this seems beneficial. Using spinlocks can sometimes improve performance, but only under certain conditions and the fact that you are in doubt rather tells me, that you are not working on any project currently where a spinlock might be beneficial. You might consider using your own "lock object", that can either use a spinlock or a mutex internally (e.g. this behavior could be configurable when creating such an object), initially use mutexes everywhere and if you think that using a spinlock somewhere might really help, give it a try and compare the results (e.g. using a profiler), but be sure to test both cases, a single-core and a multi-core system before you jump to conclusions (and possibly different operating systems, if your code will be cross-platform).

@Mecki If I'm not mistaken, I believe you suggested in your answer that time slicing only occurs on single-processor systems. This is not correct! You can use a spin lock on a single-processor system and it will spin until its time quantum expires. Then another thread of the same priority can take over (just like what you described for multi-processor systems). – fumoboy007 Apr 8 at 21:44

@fumoboy007 "and it will spin until its time quantum expires" // Which means you waste CPU time/battery power for absolutely nothing w/o any single benefit, which is utterly moronic. And no, I nowhere said that time slicing only happens on single core systems, I said on single core systems there is ONLY time slicing, while there is REAL parallelism om multicore systems (and also time slicing, yet irrelevant for what I wrote in my reply); also you entirely missed the point on what a hybrid spinlock is and why it works well on single and multicore systems. – Mecki Apr 9 at 13:42

There are different versions of spinlock: spin\_lock\_irqsave(&xxx\_lock, flags); ... critical section here .. spin\_unlock\_irqrestore(&xxx\_lock, flags); In Uni processor spin\_lock\_irqsave() should be used when data needs to shared between process context and interrupt context, as in this case IRQ also gets disables. spin\_lock\_irqsave()work under all circumstances, but partly *because* they are safe they are also fairly slow. However, in case data needs to be protected across different CPUs then it is better to use below versions, these are cheaper ones as IRQs dont get disabled in this case: spin\_lock(&lock); ... spin\_unlock(&lock);

In uniprocessor systems calling spin\_lock\_irqsave(&xxx\_lock, flags); has the same effect as disabling interrupts which will provide the needed interrupt concurrency protection without unneeded SMP protection. However, in multiprocessor systems this covers both interrupt and SMP concurrency issues.

**What is lockdep?🡺**Deadlock detection mechanism

**which synchronization mechanism is safe to use in interrupt context and why?**

Spinlocks

**When to use conditional variables?**

**Difference between Semaphores, Mutex & Spin Locks**

**How synchronization between two interrupt handlers?**

**How to synchronization between tasks and interrupt handler**

**How to sync between two tasks in Kernel**

**What is a difference between Binary Semaphore and mutex**

**How RCU are implemented?**

**how to use Read/writes are implemented**

**what is lockdep?  
  
maps all the lock dependencies as they occur in a live kernel and will warn about the following classes of locking bugs:  
- lock inversion scenarios  
- circular lock dependencies  
- hardirq/softirq safe/unsafe locking bugs  
  
Bugs are reported even if the current locking scenario does not cause any deadlock at this point.i.e. if anytime in the past two locks were taken in a different order, even if it happened for another task, even if those were different locks (but of the same class as this lock), this code will detect it.**

**Hardware spin locks?**

**Explain about the implementation of spinlock in case of ARM architecture.  
Explain about the implementation of mutex in case of ARM architecture.  
Explain about the notifier chains.  
Explain about RCU locks and when are they used?  
Explain about RW spinlocks locks and when are they used?**  
**Which are the synchronization technoques you use 'between processes', 'between processe and interrupt' and 'between interrupts'; why and how ?**

**Difference between binary and counting semaphores.?   
What are the differences between binary semaphores and spinlocks?**

mention IPCs used in user level process in linux

<http://www.chandrashekar.info/articles/linux-system-programming/introduction-to-linux-ipc-mechanims.html>

# Process Management and Process Scheduling

**What are the different scheduler’s class present in the linux kernel?**

[**https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf**](https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf)

[**http://cse.iitkgp.ac.in/~agupta/OS/MySlide-Scheduling.pdf**](http://cse.iitkgp.ac.in/~agupta/OS/MySlide-Scheduling.pdf)

[**http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560\_Proj\_main/**](http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/)

**How to create a new process?  
What is the difference between fork( ) and vfork( )?  
Which is the first task what is spawned in linux kernel?**

**What are the processes with PID 0 and PID 1?**PID 0 - idle task  
PID 1 - init

**How to extract task\_struct of a particular process if the stack pointer is given?**

**How scheduler does picks particular task?🡺**5 integer 140 queues

**When scheduler does picks a task?  
How is timeout managed?  
How does load balancing happens?  
Explain about any scheduler class?  
Explain about wait queues and how they implemented? Where and how are they used?  
What is process kernel stack and process user stack? What is the size of each and how are they allocated?  
Why do we need separate kernel stack for each process?  
What all happens during context switch?  
What is thread info? Why is it stored at the end of kernel stack?  
What is the use of preempt count variable?  
What is the difference between interruptible and uninterruptible task states?  
How processes and threads are created? (from user level till kernel level)  
How is virtual run time (runtime) calculated?**

**What are different IO schedulers available with Linux kernel?**

**Difference between 'noop' scheduler and a 'deadline scheduler?**

**Explain the how scheduler works?**

**Who schedules the Workqueues?**

**Who schedules TASKLETS?**

**How context switch happens?**

**When task switch happens?**

**How process got created in linux?**

**What are the different scheduler’s class present in the Linux kernel?**

**How to extract task\_struct of a particular process if the stack pointer is given?**

**How does scheduler picks particular task?**

**How does load balancing happens?**

**Explain about wait queues and how they implemented? Where and how are they used?**

**Why do we need separate kernel stack for each process?**

**What all happens during context switch?**

**What is thread\_info? Why is it stored at the end of kernel stack?**

**What is the use of preempt\_count variable?**

**How is virtual run time (vruntime) calculated?**

**How SCHD\_FIFO and SCHD\_RR works in RT scheduler**

**how a process exit without calling return or exit function?**

**What is the Difference between Task and Thread?**

**Difference between process and threads?**

**what are threads and process and what are the difference between them?**

**Process**  
Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

**Thread**   
A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

**How a thread is a light weight process?**

[**https://www.kernel.org/doc/ols/2002/ols2002-pages-330-337.pdf**](https://www.kernel.org/doc/ols/2002/ols2002-pages-330-337.pdf)

<http://www.tutorialspoint.com/operating_system/os_multi_threading.htm>

<https://perso.ens-lyon.fr/eric.fleury/CPS/ASR2/slides/L3_ASR2_04_threads.pdf>

<http://stackoverflow.com/questions/27581747/pthread-vs-kthread-in-linux-kernel-v2-6>

[**http://stackoverflow.com/questions/15601155/does-linux-schedule-a-process-or-a-thread**](http://stackoverflow.com/questions/15601155/does-linux-schedule-a-process-or-a-thread)

# Timers and Time Management

**What are jiffies and HZ?  
What is the initial value of jiffies when the system has started?  
Explain about HR timers and normal timers?  
On what hardware timers, does the HR timers are based on?  
How to declare that a specific hardware timers is used for kernel periodic timer interrupt used by the scheduler?  
How software timers are implemented?**

**How to calculate the timeticks?  
what is OSticks?**

# Power Management in Linux

**Explain about cpuidle framework.  
Explain about cpufreq framework.  
Explain about clock framework.  
Explain about regulator framework.  
Explain about suspened and resume framwork.  
Explain about early suspend and late resume.  
Explain about wakelocks.**

**explain architecture of platform driver?  
explain clock driver architecture?**

# Linux Kernel Modules:

**How insmod works**

<http://gomathikumar1006.blogspot.in/2013/09/linux-kernel-module-internals-of-insmod.html>

<http://www.tldp.org/HOWTO/html_single/Module-HOWTO/#MEMALLOC>

insmod makes an init\_module system call to load the LKM into kernel memory. Loading it is the easy part, though. How does the kernel know to use it? The answer is that the init\_module system call invokes the LKM's initialization routine right after it loads the LKM. insmod passes to init\_module the address of the subroutine in the LKM named init\_module as its initialization routine.

The LKM author set up init\_module to call a kernel function that registers the subroutines that the LKM contains. For example, a character device driver's init\_module subroutine might call the kernel'sregister\_chrdev subroutine, passing the major and minor number of the device it intends to drive and the address of its own "open" routine among the arguments. register\_chrdev records in base kernel tables that when the kernel wants to open that particular device, it should call the open routine in our LKM.

1) Insmod is a small program, which calls init\_module() to intimate the kernel that a module is attempted to be loaded and transfers the control to the kernel.  
2) In kernel, sys\_init\_module() is run. It does a sequence of operations as follows  
    a) Verifies if the user who attempts to load the module has the permission to do so or not.  
    b) After verification, load\_module function is called.  
        b.1) The load\_module function assigns temporary memory and copies the elf module from user space to kernel memory using copy\_from\_user.  
        b.2) It then checks the sanity of the ELF file (Verification if it is a proper ELF file etc)  
        b.3) Then based on the ELF file interpretation, it generates offset in the temporary memory space allocated. This is called the convenience variables.  
        b.4) User arguments to the module are also copied to the kernel memory  
        b.5) The state of the module is updated to MODULE\_STATE\_COMING  
        b.6) The actual location in the kernel memory is allocated using SHF\_ALLOC  
        b.7) Symbol resolution is done.  
        b.8) The load\_module function returns a reference to the kernel module.  
    c) The reference to the module returned by load\_module is added to a doubly linked list that has a list of all the modules loaded in the system.  
    d) Then the module\_init function in the module code is called.  
    e) Module state is updated to MODULE\_STATE\_LIVE

**Modprobe vs Insmod?**

1. modprobe calculates all of the module dependencies and then load the module along with the dependencies, while insmod does not care about dependencies, insmod only loads the module in question.

2. How modprobe calculates module dependencies? First let me introduce you the depmod tool, this tool’s function is to calculate module dependencies for all modules located in /lib/modules/`uname -r`/ and then keep the dependencies information in /lib/modules/`uname -r`/modules.dep. depmod usually runs when system is booted or when there is new module installed or when you call depmod -a from shell. For example, consider module A, and this module depends on module B and module C. depmod finds that module A needs module B and module C, and then depmod keeps this information in form of “A:B C” in /lib/modules/`uname -r`/modules.dep. Now, let us return to modprobe. modprobe basically reads the modules.dep file and then loads, using insmod, the module and all of the dependencies.

3. So why modprobe /path/to/module.ko didn’t work? It is simply because modprobe only deals with modules in /lib/modules/`uname -r`/. So, if you want it to work, create soft link from /path/to/module.ko to /lib/modules/`uname -r`/.

ln -s /path/to/module.ko /lib/modules/`uname -r`

and don’t forget to refresh the modules.dep

depmod -a

and call the modprobe, also loose the .ko

modprobe module

**How to make a module as loadable module?🡺**

in .config give m or in menuconfig select m or in make file

To restrict a component build to module-only, qualify its config symbol

with "depends on m". E.g.:

config FOO

depends on BAR && m

**How to make a module as in-built module?**

**Explain about Kconfig build system?**

The Linux Kernel Build System has four main components:

Config symbols: compilation options that can be used to compile code conditionally in source files and to decide which objects to include in a kernel image or its modules.

Kconfig files: define each config symbol and its attributes, such as its type, description and dependencies. Programs that generate an option menu tree (for example, make menuconfig) read the menu entries from these files.

.config file: stores each config symbol's selected value. You can edit this file manually or use one of the many make configuration targets, such as menuconfig and xconfig, that call specialized programs to build a tree-like menu and automatically update (and create) the .config file for you.

Makefiles: normal GNU makefiles that describe the relationship between source files and the commands needed to generate each make target, such as kernel images and modules.

**Explain about the init call mechanism.**

**What is the difference between early init and late init?  
Early init:**Early init functions are called when only the boot processor is online.  
Run before initializing SMP.  
Only for built-in code, not modules.  
Late init:  
Late init functions are called \_after\_ all the CPUs are online.

# Linux Kernel Debugging

[**https://www.safaribooksonline.com/library/view/linux-device-drivers/0596005903/ch04.html**](https://www.safaribooksonline.com/library/view/linux-device-drivers/0596005903/ch04.html)

**What is Oops and kernel panic?**

[**http://venkateshabbarapu.blogspot.in/2012/11/oops-vs-kernel-panic.html#!/2012/11/oops-vs-kernel-panic.html**](http://venkateshabbarapu.blogspot.in/2012/11/oops-vs-kernel-panic.html#!/2012/11/oops-vs-kernel-panic.html)

**Does all Oops result in kernel panic?  
What are the tools that you have used for debugging the Linux kernel?  
What are the log levels in printk?  
Can printk's be used in interrupt context?  
How to print a stack trace from a particular function?  
What's the use of early\_printk( )?  
Explain about the various gdb commands.**

**Debugging OOM 🡺**[**http://linux-mm.org/OOM**](http://linux-mm.org/OOM)

**how do you understand Kernel OOPs?**

**1) What are the tools that can be used for debugging the Linux kernel?**-kdb  
-kgdb  
-printk  
-Linux Traces   
-JTAG,ICE etc,  
-Sniffers  
-profiling  
- Crash dump  
-UML ( User Mode Linux)  
- simulators - Qemu  
- Hypervisor - VMware/kvm etc  
- Magic sysrq  
- lockdep  
 **2) How debugging works**

# VFS:

[**https://www.ibm.com/developerworks/library/l-initrd/**](https://www.ibm.com/developerworks/library/l-initrd/)

[**http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html**](http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html)

[**http://stackoverflow.com/questions/10603104/the-difference-between-initrd-and-initramfs**](http://stackoverflow.com/questions/10603104/the-difference-between-initrd-and-initramfs)

How to resize ext4 file system?

Why sysfs is required?

Explain about debugfs?

How to create a file directly from the kernel?

Different types of DMA transfer

1) flyby DMA transfer

2) fetch-and-deposit DMA transfer

# Miscellaneous

1) Is it possible to save RAM memory contents across reboots?

Yes... It is possible... At the time of reboot if we won't stop the refresh cycle of RAM it is possible store the RAM contents safe. There is file system build on the same theory is PRAMS file system. To find the complete information use the link bellow.

<http://pramfs.sourceforge.net/>  
  
6) How to add new system call

7) Why page cache is been removed from the kernel

9) What is the significance of asmlinkage

<https://www.quora.com/Linux-Kernel/What-does-asmlinkage-mean-in-the-definition-of-system-calls>

10) How to do declare variable local to the cpu only? 🡺<http://www.makelinux.net/ldd3/chp-8-sect-5>

11) **How to you declare a variable, which is local to the thread only?**

long register\_variable asm ("r15" );

is already declaring a thread-local variable, merely due to the fact that each thread has its own set of register values.

There is no possible way to make GCC's global register-storage variables shareable between threads.

12) Can you generate Kernel configuration file after building the image?

13) How size of RAM memory is detected in the Linux kernel?

14) What is boot memory? <https://www.kernel.org/doc/gorman/html/understand/understand008.html>

15) how to find interrupt latency?

16) What is need of copy\_to\_user to copy\_from\_user,cannot we use memcpy?

20) Difference between PCI and PCIe?

22) How to you protect buffer over flow in Linux kernel

23) How physical to virtual memory conversion happens?

26) How to pass event from kernel space to user level?

27) What are different communication methods available from user space to kernel space?

28) When to use which type of kernel synchronization methods?

29) How to solve problems for cache coherency ?

30) How plug and play works in the Linux kernel?

35) Different boot loader can be used?

36) What is difference between Kernel space virtual address and user space virtual address difference?

53) how debugger works?  
54) Linux Booting process?  
55) Explain stack winding and unwinding happens?  
56) Difference between Edge triggering, level triggering, MSI interrupts?  
57) how usb devices enumerate?  
58) Explain how Linux device trees?  
59) How pcie devices enumerate  
  
How to protect a single variable in a c data structure from cache coherency?

how read mostly works

why static variable initilizes only once?

When to use likely and unlikely

**How are the atomic functions implemented in case of ARM architecture?**

[**http://linuxkernelarticles.blogspot.in/2013/02/atomic-function-implementation-in-arm.html**](http://linuxkernelarticles.blogspot.in/2013/02/atomic-function-implementation-in-arm.html)

**How is container\_of( ) macro implemented?**

**Explain about system call flow in case of ARM Linux.**

**What 's the use of \_\_init and \_\_exit macros?**

**How to ensure that init function of a particular driver was called before our driver's init function is**

**called (assume that both these drivers are built into the kernel image)?**

**What's a segmentation fault and what are the scenarios in which segmentation fault is triggered?**

**If the scenarios which triggers the segmentation fault has occurred, how the kernel identifies it and what are the actions that the kernel takes?**

Q1. When is that we want to use "user virtual address" instead of "kernel virtual address"? List some situations when we cannot go with kernel virtual address.

In our ioctl interface we use it very heavily to copy params from user space to kernel structures and vice versa same is for buffers.

copy\_to\_user() copy\_from\_user() requires user virtual address.

if ((err = copy\_from\_user(&UserParams, (STBLAST\_Ioctl\_CopyCmdsToBuffer\_t \*)arg, sizeof(STBLAST\_Ioctl\_CopyCmdsToBuffer\_t)) )){

/\* Invalid user space address \*/

goto fail;

}

TransferParams\_p = GetTransferBlockFromHandle(UserParams.Handle);

UserParams.ErrorCode = STBLAST\_CopyCmdstoBuffer(UserParams.Handle,

TransferParams\_p->KernelSpaceBufCmd\_p,

UserParams.NumCmdtoCopy,

&(UserParams.NumCmdCopied));

/\* Copy Kernel buffer contents to user buffer \*/

if((err=copy\_to\_user(UserParams.Cmd\_p,TransferParams\_p->KernelSpaceBufCmd\_p,

(UserParams.NumCmdtoCopy \* sizeof(U32) )))){

/\* Invalid user space address \*/

SET\_TRACE(STBLAST\_IOCTL\_CID,ERROR,LEVEL0,"Invalid user space address");

goto fail;

}

if((err = copy\_to\_user((STBLAST\_Ioctl\_CopyCmdsToBuffer\_t\*)arg, &UserParams,

sizeof(STBLAST\_Ioctl\_CopyCmdsToBuffer\_t)))) {

As kernel can access user space memory, why should copy\_from\_user is needed?

one of the major requirement in system call implementation is to check the validity of user parameter pointer passed as argument, kernel should not blindly follow the user pointer as the user pointer can play tricks in many ways. Major concerns are: 1. it should be a pointer from that process address space - so that it cant get into some other process address space. 2. it should be a pointer from user space - it should not trick to play with a kernel space pointer. 3. it should not bypass memory access restrictions.   
  
That is why copy\_from\_user() is performed. It is blocking and process sleeps until page fault handler can bring the page from swap file to physical memory.

User memory is untrusted.

It is possible that a thread doing a syscall will get scheduled out after validating the user memory but before actually completing the syscall. A different process thread might now replace the user memory with a different one, invalidating the memory. (Or if it is a multiprocessor machine...)

These functions do a few things:

* They check if the supplied userspace block is entirely within the user portion of the address space (access\_ok()) - this prevents userspace applications from asking the kernel to read/write kernel addresses;
* They return an error if any of the addresses are inaccessible, allowing the error to be returned to userspace (EFAULT) instead of crashing the kernel;
* They allow architecture-specific magic, for example to ensure consistency on architectures with virtually-tagged caches or to disable SMAP protection.

How mb differ from wmb in Linux? Is mb equal to wmb & rmb ?

General memory barriers(mb) make a guarantee that all memory accesses specified before the barrier will happen before all memory accesses specified after the barrier.  
  
Read memory barriers(rmb) make a guarantee that all memory reads specified before the barrier will happen before all memory reads specified after the barrier.  
  
Write memory barriers(wmb) make a guarantee that all memory writes specified before the barrier will happen before all memory writes

# 

# Study Link

Generic Site :

<http://www.geeksforgeeks.org/>  Generic site good for c and ds

<http://www.indiabix.com/technical/operating-systems/2>  Generic site for C and DS

[http://chinswe.webs.com](http://chinswe.webs.com/) Good generic Site

<http://nptel.iitm.ac.in/syllabus/syllabus.php?subjectId=106104025>  Good generic Site for video lectures

<http://www.eventhelix.com/RealtimeMantra/Basics/debugging_software_crashes_2.htm>  Debugging

Embedded Interview questions quick go through:

<http://sites.google.com/site/sumedhshende/operatingsystems>  OS entry level questions

<http://a4academics.com/interview-questions/57-c-plus-plus/722-embedded-c-interview-questions>

<http://gelliphanindraviswanadhaprasad.org/interview-questions/embedded-interview-questions/>  Some basic questions

<http://www.careercup.com/page?pid=linux-kernel-interview-questions>  Good site to start with

<http://embeddedlinuxadventures.blogspot.in/2006/02/frequent-linux-kernel-interview.html>  Good site to start with

<http://embedded-telecom-interview.blogspot.in/2010/07/linux-and-linux-device-driver-interview.html>  c + Linux mix

<http://www.devbistro.com/tech-interview-questions/Hardware.jsp>  Interview questions ok

<http://linuxinterviewpreperation.blogspot.in/>  Some Advance questions

<http://www.sanjayahuja.com/Interview%20questions.pdf>  Embedded questions

<http://www.techinterviews.com/embedded-systems-interview-questions>

<http://www.globalguideline.com/forum/viewtopic.php?id=293>  25 questions on Embedded

[http://embedded-thoughts.blogspot.in](http://embedded-thoughts.blogspot.in/)  Generic topics on embedded and linux

<http://embeddedcoaching.blogspot.in/p/homepage.html>  Links to other sites

<http://www.careerride.com/embedded-systems-interview-questions.aspx>

<http://stackoverflow.com/questions/tagged/linux-device-driver+embedded-linux+linux-kernel?sort=votes&pageSize=15>  Practical questions

Linux:

<http://www.electronics.dit.ie/staff/tscarff/DMA/dma.htm>

<http://www.informit.com/articles/article.aspx?p=414983>

<http://venkateshabbarapu.blogspot.in/2012/09/interrupt-handling-in-arm.html>  good one...

<http://www.tldp.org/LDP/lki/lki-1.html#ss1.2>   Booting overview

<http://www.hep.by/gnu/kernel/kernel-locking/>  All u need to know abt kernel locking

<http://www.osdever.net/bkerndev/Docs/gettingstarted.htm>  Linux site

<http://destinationlinux.blogspot.in/p/kernel-drivers-for-i2c-devices.html>

<http://linux.die.net/lkmpg/x892.html>   ioctl explanation

<http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html>

<http://my.safaribooksonline.com/book/operating-systems-and-server-administration/linux/0596002130/interrupts-and-exceptions/linuxkernel2-chp-4-sect-6> very helpfull....

<http://lirc.sourcearchive.com/documentation/0.8.3/dir_13ec18a84893da13e253109a5710f0f3.html>

<http://www.makelinux.net/books/lkd2/?u=ch09lev1sec2>  Linux books

<http://www.makelinux.net/ldd3/chp-14-sect-2>  Linux books

<https://www.youtube.com/user/linuxkernellearning?feature=g-high-u>

<http://free-electrons.com/blog/elce-2010-tutorial-videos/>

<https://www.youtube.com/playlist?list=PL16941B715F5507C5>

<http://stackoverflow.com/questions/11802055/fork-outputs-after-the-process-execution-is-over>

<http://elinux.org/images/4/4f/02-linux-quick-start.pdf>

<http://www.ibm.com/developerworks/library/l-user-space-apps/>🡺 kernel to user space

<http://www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/forelesninger-rt/2011-2_POSIX_Threads_Programming.pdf> ==>

<http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html#SCHEDULING>

Embedded:

<http://etutorials.org/Linux+systems/embedded+linux+systems/Building+Embedded+Linux+Systems/>**🡺 rootfs+application**

<http://www.embeddedrelated.com/showarticle/59.php>

<http://www.embeonline.com/uploads/O'Reilly-programming_embedded_systems_in_C_and_C++.pdf>

<http://www.rt-embedded.com/blog/archives/resolving-crashes-and-segmentation-faults/>

<http://www.eeherald.com/section/design-guide/esmod.html>  Online course on embedded system

<http://embeddedlinuxadventures.blogspot.in/2006/02/frequent-linux-kernel-interview.html>

Device Driver:

<http://compgroups.net/comp.unix.programmer/are-device-drivers-reentrant-or-daemons/51038>

HW ARCH:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344b/BEIIEECF.html> Good Arm Site

<http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems>  Multi core revision

C Langage:

<http://www.rmbconsulting.us/a-c-test-the-0x10-best-questions-for-would-be-embedded-programmers>

[https://docs.google.com/document/d/1defosE0lwexTWG7KyWFx74xhh93F6pGYHGSqqS01C\_4/edit?hl=en\_GB#](https://docs.google.com/document/d/1defosE0lwexTWG7KyWFx74xhh93F6pGYHGSqqS01C_4/edit?hl=en_GB)

<http://www.csa.iisc.ernet.in/sem-evts/opendays2012/C_Questions.pdf>

<http://www.sanfoundry.com/interview-questions-answers-c-bitwise-operators/>

<http://lambda.uta.edu/cse5317/notes/node33.html>  🡺 Run Time Storage Organization

<http://courses.engr.illinois.edu/ece390/books/labmanual/c-prog-mixing.html>  ok ok site

<http://pic.dhe.ibm.com/infocenter/zos/v1r13/index.jsp?topic=%2Fcom.ibm.zos.r13.cbclx01%2Fcplr242.htm>  IBM C

<https://www.doc.ic.ac.uk/lab/cplus/cstyle.html>

<http://videolectures.net/stanfordcs107s08_programming_paradigms/>  Good video lectures for c concepts

<http://www.codingunit.com/c-tutorial-compilers-gnu-and-visual-studio>  C tutorial

<http://www.ibm.com/developerworks/library/l-reent/index.html>  re entrant functions

<http://blog.regehr.org/archives/28>🡺 Volatile

Wiki links:

<http://en.wikipedia.org/wiki/Interrupt_latency>

<http://en.wikipedia.org/wiki/Pointer_%28computing%29#Null_pointer>

<http://en.wikipedia.org/wiki/Real-time_operating_system>

<http://en.wikipedia.org/wiki/Static_variable>

<http://en.wikipedia.org/wiki/Volatile_variable>

<http://en.wikipedia.org/wiki/Device_driver>

<http://en.wikipedia.org/wiki/Semaphore_%28programming%29>

<http://en.wikipedia.org/wiki/Critical_section>

<http://en.wikipedia.org/wiki/Embedded_system>

<http://en.wikipedia.org/wiki/Data_structure_alignment>

<http://en.wikipedia.org/wiki/Bus_error>

<http://en.wikipedia.org/wiki/Memory_leak>

Misc :

<http://randu.org/tutorials/threads/>

<http://ubuntuforums.org/archive/index.php/t-1671019.html>

assembly :

<http://brahe.canisius.edu/~meyer/253/BOOK/ch15/FULLPAGE/ch15-5.html>

http://www.massey.ac.nz/~mjjohnso/notes/59304/l3.html

[**http://www.slideshare.net/AsmaaLafi/memory-allocation-for-real-time-operating-system**](http://www.slideshare.net/AsmaaLafi/memory-allocation-for-real-time-operating-system)

Linux is a general-purpose OS (GPOS); its application to embedded systems is usually motivated by the availability of device support, file-systems, network connectivity, and UI support. All these things can be available in an RTOS, but often with less broad support, or at additional cost or integration effort.

Many RTOS are not full OS in the sense that Linux is, in that they comprise of a static link library providing only task scheduling, IPC, synchronisation timing and interrupt services and little more - essentially the scheduling kernel only. Such a library is linked with your application code to produce a single executable that your system boots directly (or via a bootloader). Most RTOS do not directly support the loading and unloading of code dynamically from a file system as you would with Linux - it is all there at start-up and runs until power down.

Critically Linux is not real-time capable. An RTOS provides scheduling guarantees to ensure deterministic behaviour and timely response events and interrupts. In most cases this is through a priority based pre-emptive scheduling algorithm, where the highest priority task ready to run always runs - immediately - pre-empting any lower priority task without a specific yield or relinquishing of the CPU, or completion of a time-slice.

Linux has a number of scheduling options, including a real-time scheduler, but this is at best "soft" real-time - a term I dislike since it is ill-defined, and essentially means real-time, most of the time, but sometimes not. If your application has no need of "hard" real-time, that's fine, but typical latencies in real-time Linux will be in the order of tens or hundreds of microseconds, whereas a typical RTOS real-time kernel can achieve from latencies from zero to a few microseconds.

Another issue with embedded Linux is that it needs significant CPU resources, perhaps >200MIPS, 32bit processor, ideally with an MMU, 4Mb of ROM and 16MB or RAM to just about boot (which may take several seconds). An RTOS on the other hand can be up in milliseconds, run in less than 10Kb, on microcontrollers from 8-bit up. This can have a significant impact on system cost for volume production despite being ostensibly "free".

There are larger RTOS products that exhibit some of the features of a GPOS such dynamic loading, filesystems, networking, GUI (for example as QNX), and many RTOS provide a POSIX API (usually secondary to their native real-time API) for example VxWorks and agian QNX, so that a great deal of code developed for Linux and Unix can be ported relatively easily. These larger more comprehensive RTOS products remain scalable, so that functionality not required is not included. Linux in comparison has far more limited scalability.