Linux Kernel

Agenda

- Prerequisites.
- A bird's-eye view
  - History.
  - Kernel basic elements.
- Kernel details.
- Kernel device drivers.

Prerequisites

- C programming
  - Recommended Ref: C Programming Language, Brian W. Kernighan, Dennis Ritchi.
- Microprocessors
  - Recommended Ref: INTEL 80386 PROGRAMMER'S REFERENCE MANUAL.
- Operating system
  - Recommended Ref: Operating System Concepts, Silberschatz, Galvin, Gagne.
- For Linux background, Very recommended
  - Prof: Ahmed Elarabawy courses

    Course101 : Introduction to Embedded Linux.

    Course102: Understanding Linux.

A bird's-eye View

The Linux philosophy is 'Laugh in the face of danger'.
Oops. Wrong One. 'Do it yourself'. Yes, that's it.
Linus Torvalds

History

- 1983, Richard Stallman, GNU project and the free software concept. Beginning of the development of gcc, gdb, glibc and other important tools.
- 1991, Linus Torvalds created Linux, then came to the GNU Movement to be the greatest SW project in the world!.
- Started as a free SW, this means
  - Run the software for any purpose.
  - Study the software and to change it.
  - Redistribute copies and distribute copies of modified versions.

Kernel Basic Elements

- What does the kernel do?
  - Manage all SW/HW resources (CPU, Memory, External and internal buses, Connected Devices, …).
  - Provide some standard API’s to deal with the different resources.
  - Manage different application requests and their processes
- Design Approaches! (Micro VS Monolithic Kernel).

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Subject: Re: LINUX is obsolete
Date: 29 Jan 92 23:14:26 GMT
Organization: University of Helsinki

Well, with a subject like this, I'm afraid I'll have to reply.
Apologies to minix-users who have heard enough about linux anyway.  I'd
like to be able to just "ignore the bait", but ...  Time for some
serious flamefesting!

In article <12595@star.cs.vu.nl> ast@cs.vu.nl (Andy Tanenbaum) writes:
>
>I was in the U.S. for a couple of weeks, so I haven't commented much on
>LINUX (not that I would have said much had I been around), but for what
>it is worth, I have a couple of comments now.
>
>As most of you know, for me MINIX is a hobby, something that I do in the
>evening when I get bored writing books and there are no major wars,
>revolutions, or senate hearings being televised live on CNN.  My real
>job is a professor and researcher in the area of operating systems.

You use this as an excuse for the limitations of minix? Sorry, but you
loose: I've got more excuses than you have, and linux still beats the
pants of minix in almost all areas.  Not to mention the fact that most
of the good code for PC minix seems to have been written by Bruce Evans.

Re 1: you doing minix as a hobby - look at who makes money off minix,
and who gives linux out for free.  Then talk about hobbies.  Make minix
freely available, and one of my biggest gripes with it will disappear.
Linux has very much been a hobby (but a serious one: the best type) for
me: I get no money for it, and it's not even part of any of my studies
in the university.  I've done it all on my own time, and on my own
machine.
```

Design Approaches!

- Micro kernel
  - Elementary functions only are implemented in the kernel space and the others running in the user space.
  - It seems more elegant.
  - Most of the Changes don’t require another kernel build.
- Monolithic Kernel
  - All the kernel functions implemented at the kernel space.
  - Faster.
  - But Every change most probably needs a kernel build.

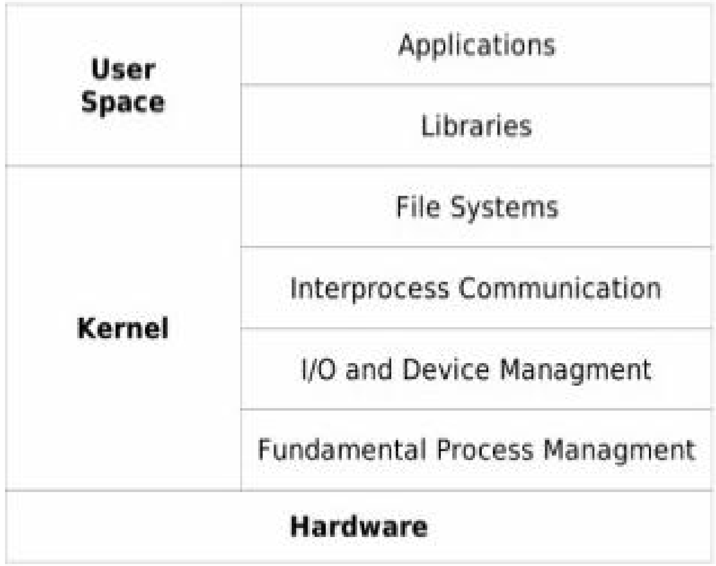


Figure 1: Monolithic kernel based operating system

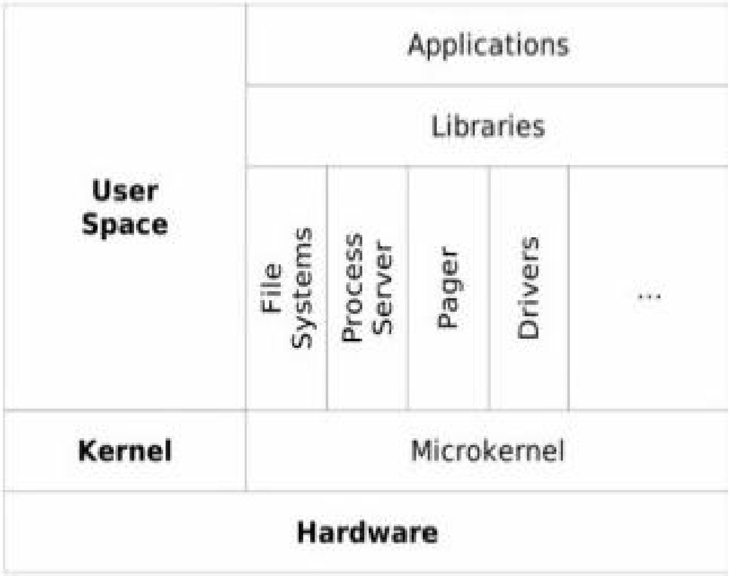


Figure 2: Microkernel based operating system

Design Approaches!, *Cont'd*

- Linux is Monolithic.
- MAC OS started as Micro kernel but, now it tends to be Hybrid.

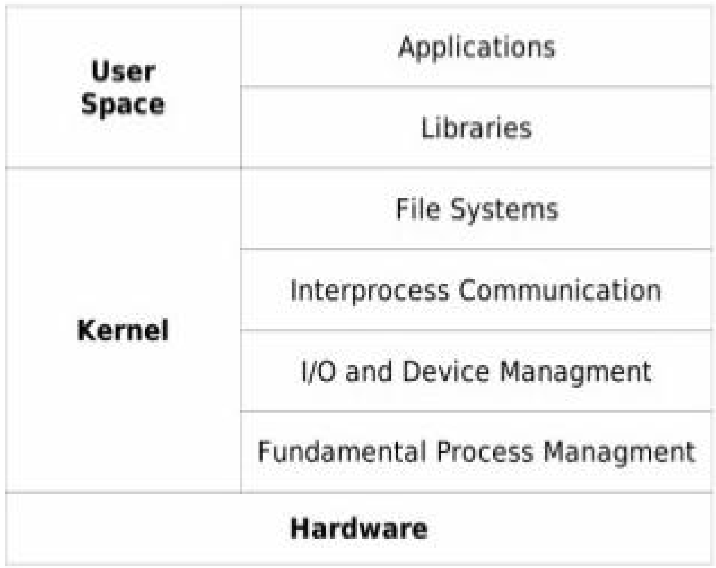


Figure 1: Monolithic kernel based operating system

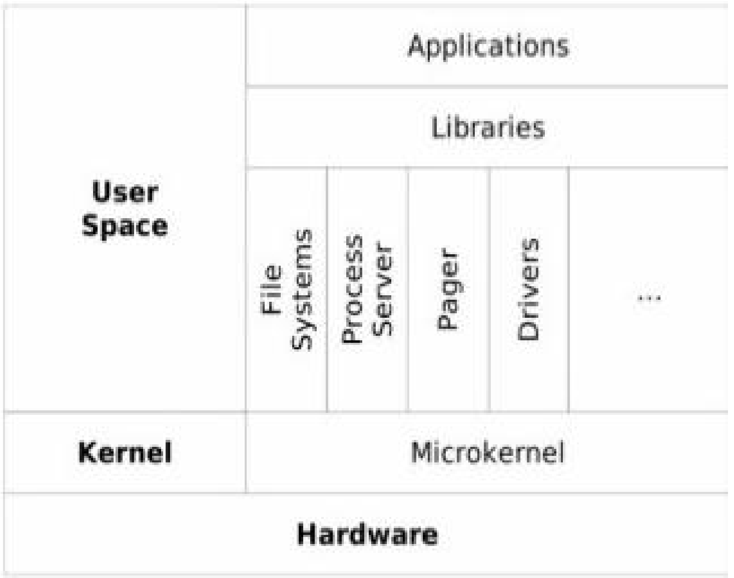
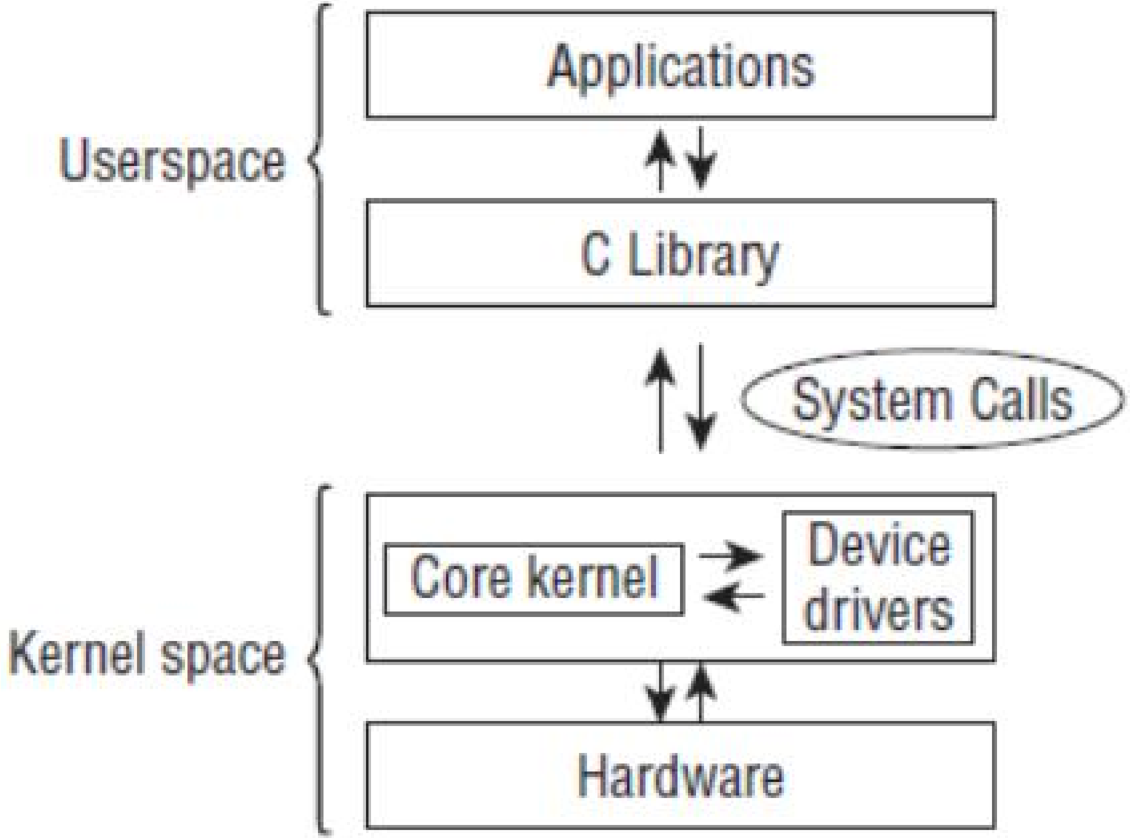


Figure 2: Microkernel based operating system

Kernel & User Space



Processes

- ❑ Application is running under one or more process.
- ❑ Linux is a multi-threading OS.
- ❑ Each process sees independent virtual address space provided by the kernel.
- ❑ Kernel is responsible for task switching and the scheduling.
- ❑ View the processes tree running using terminal,

  $ pstree

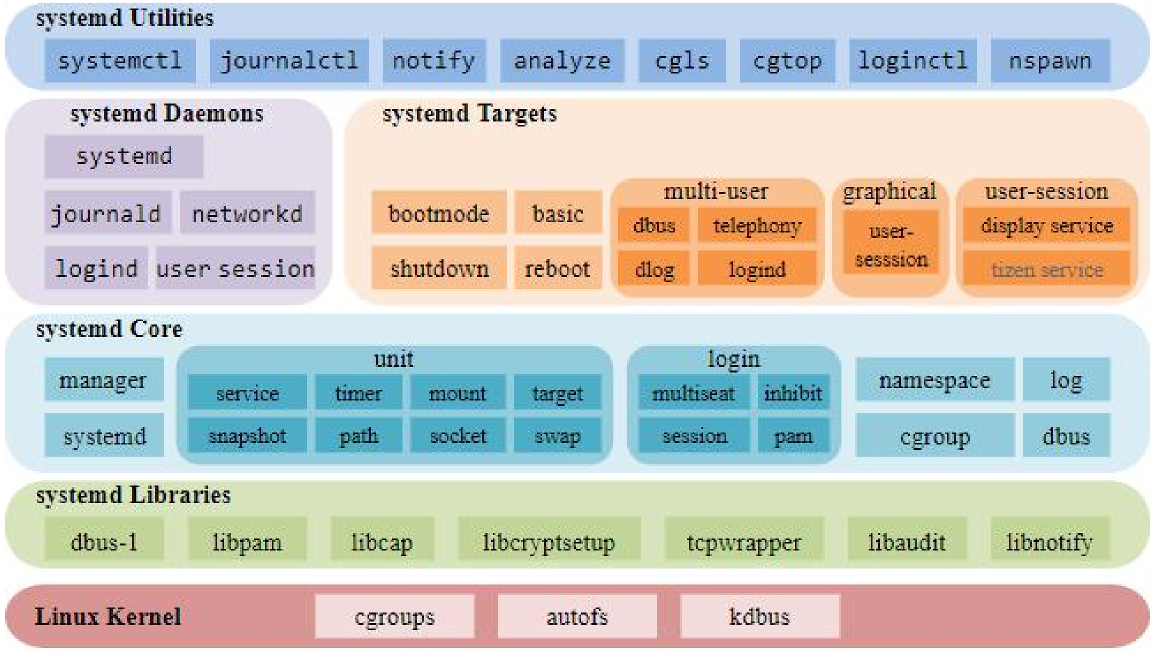
- ❑ If you still don't have any distro for linux (ex.Debian Ubuntu) you can follow this lecture on youtube.

  Course 101 : Lecture 6: Installing Ubuntu

```
karim_eshapa@karimeshapa-vm:~$ pstree
systemd─┬─ModemManager─┬─{gdbus}
        │              └─{gmain}
        ├─NetworkManager─┬─dhclient
        │                ├─dnsmasq
        │                ├─{gdbus}
        │                └─{gmain}
        ├─VBoxClient───VBoxClient
        ├─VBoxClient
        ├─accounts-daemon─┬─{gdbus}
        │                 └─{gmain}
        ├─acpid
        ├─agetty
        ├─anacron
        ├─apt.systemd.dai───apt.systemd.dai───unattended-upgr───unattended-upgr+
        ├─aptd───{gmain}
        ├─avahi-daemon───avahi-daemon
        ├─colord─┬─{gdbus}
        │        └─{gmain}
        ├─cron
        ├─cups-browsed─┬─{gdbus}
        │              └─{gmain}
        ├─cupsd
        ├─dbus-daemon
        ├─irqbalance
        ├─lightdm─┬─Xorg─┬─{llvmpipe-0}
        │         │      ├─{llvmpipe-1}
        │         │      ├─{llvmpipe-2}
        │         │      └─{llvmpipe-3}
        │         ├─lightdm─┬─upstart─┬─at-spi-bus-laun─┬─dbus-daemon
        │         │         │         │                 ├─{dconf worker}
        │         │         │         │                 ├─{gdbus}
        │         │         │         │                 └─{gmain}
        │         │         │         ├─at-spi2-registr─┬─{gdbus}
        │         │         │         │                 └─{gmain}
        │         │         │         ├─bamfdaemon─┬─{dconf worker}
        │         │         │         │            ├─{gdbus}
```

Processes, Cont'd

- ❑ ***Systemd*** is just a group of programs runnig on the top of the kernel by different distro (Like Ubuntu), it runs as first process on boot (as Process ID PID 1).
- ❑ ***Systemd*** has replaced ***SysV Init*** that initializes the first init process; Forget about it now.



Processes, Cont'd

- But, how these processes started, it's started by the first process ***PID 1*** that will be a parent for the upcomming processes, this happens by the following system calls,
  - Fork: Generates an exact copy of the current process that differs from the parent process only in its PID (Process ID).
  - Exec: Loads a new program into an existing content and then executes it.

```
karim_eshapa@karimeshapa-vm:~$ pstree
systemd─┬─ModemManager─┬─{gdbus}
        │              └─{gmain}
        ├─NetworkManager─┬─dhclient
        │                ├─dnsmasq
        │                ├─{gdbus}
        │                └─{gmain}
        ├─VBoxClient───VBoxClient
        ├─VBoxClient
        ├─accounts-daemon─┬─{gdbus}
        │                 └─{gmain}
        ├─acpid
        ├─agetty
        ├─anacron
        ├─apt.systemd.dai───apt.systemd.dai───unattended-upgr───unattended-upgr+
        ├─aptd───{gmain}
        ├─avahi-daemon───avahi-daemon
        ├─colord─┬─{gdbus}
        │        └─{gmain}
        ├─cron
        ├─cups-browsed─┬─{gdbus}
        │              └─{gmain}
        ├─cupsd
        ├─dbus-daemon
        ├─irqbalance
        ├─lightdm─┬─Xorg─┬─{llvmpipe-0}
        │         │      ├─{llvmpipe-1}
        │         │      ├─{llvmpipe-2}
        │         │      └─{llvmpipe-3}
        │         ├─lightdm─┬─upstart─┬─at-spi-bus-laun─┬─dbus-daemon
        │         │         │         │                 ├─{dconf worker}
        │         │         │         │                 ├─{gdbus}
        │         │         │         │                 └─{gmain}
        │         │         │         ├─at-spi2-registr─┬─{gdbus}
        │         │         │         │                 └─{gmain}
        │         │         │         ├─bamfdaemon─┬─{dconf worker}
        │         │         │         │            ├─{gdbus}
```

Linux User Threads

- Linux thread is called a light weight process.
- A process may consist of several threads that all share the same data and resources.
- Clone system call is used to generate threads.
- List the threads of running process (ex. XORG PID 1009),

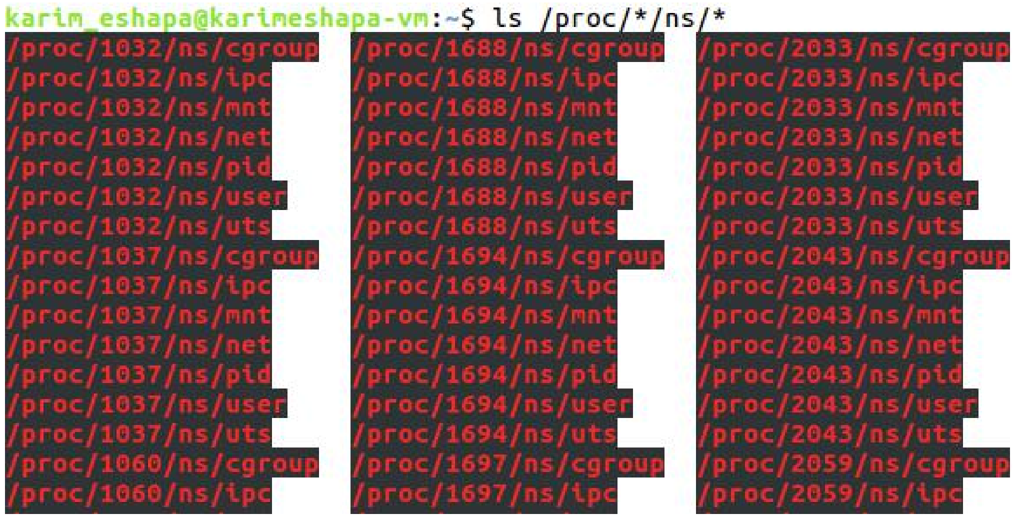
  $ ps -T -p 1009

```
karim_eshapa@karimeshapa-vm:~$ ps -T -p 1009
  PID  SPID TTY          TIME CMD
 1009  1009 tty7     00:00:29 Xorg
 1009  1018 tty7     00:00:00 llvmpipe-0
 1009  1019 tty7     00:00:00 llvmpipe-1
 1009  1020 tty7     00:00:00 llvmpipe-2
 1009  1021 tty7     00:00:00 llvmpipe-3
```

Namespaces

- A new OS feature is implemented.
- Each namespace is a separate view for the whole system.
- Used at the following,
  - Containers: Create multiple views of the system where each seems to be a complete Linux installation from within the container and does not interact with other containers.
  - Containers are separated and segregated from each other.
  - Unlike VM “Virtual Machine” (ex: KVM) doesn’t need to build separate kernels running with differnet features.
- List Namespaces,

  $ ls /proc//ns/**



Cgroups

- Control groups, usually referred to as ***cgroups***, allow processes to be organized into hierarchical groups whose usage of resources can be limited and monitored, such as
  - ***Limiting*** the amount of CPU time and memory available to a cgroup.
  - ***Accounting*** for the CPU time used by a cgroup, and freezing and resuming execution of the processes in a cgroup.
- The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs.
- cgroups for a controller are arranged in a hierarchy, this hierarchy is defined through subdirectories within the cgroup filesystem.
- So, any subhierarchy underneath a certain cgroup has the same limits, control, and accounting placed on the cgroup at a higher level in the hierarchy.
- List cgroups and its resource usage

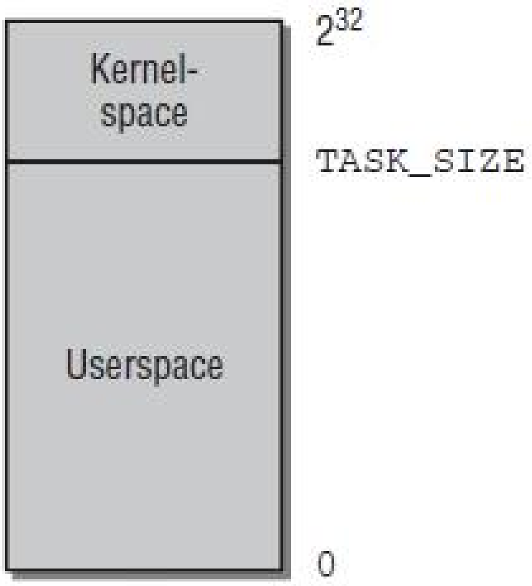
  $ systemd-cgtop

```
karim_eshapa@karimeshapa-vm:~$ systemd-cgtop

Control Group                          Tasks   %CPU   Memory  Input/s Output/s
/                                          -   13.6     1.0G        -        -
/init.scope                                1      -        -        -        -
/system.slice                             23      -        -        -        -
/user.slice                              241      -        -        -        -
/user.slice/user-1000.slice              241      -        -        -        -
```

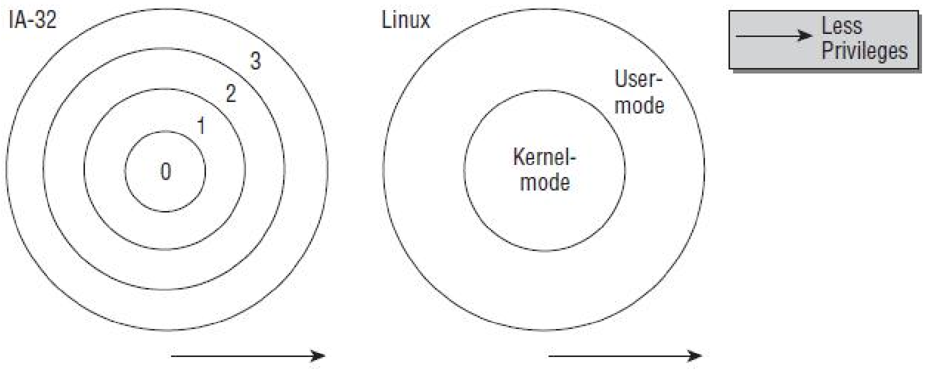
Virtual Address Space

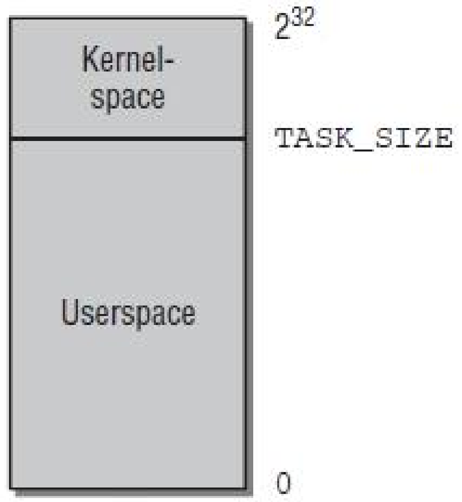
- Every process in the system has the impression that it would solely lives in this address space, and other processes are not present from their point of view, this address space called “Virtual address space”.
- Every user process in the system has its own virtual address range that extends from 0 to **TASK_SIZE**.
- **TASK_SIZE** is an architecture-specific.
- IA-32 systems for example sees the virtual address space for each process is 3 GB and 1 GB is available to the kernel.



Privilege Levels

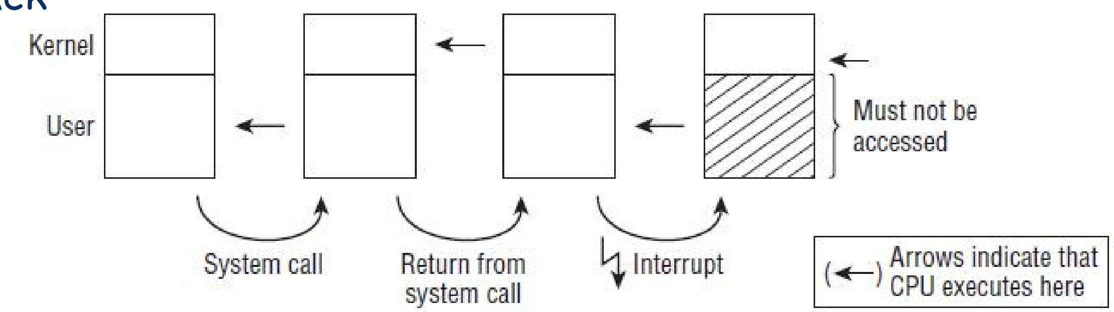
- For the following IA-32 systems, The inner rings are able to access more functions, the outer rings less, this architecture has 4 modes of priviledges.
- Linux uses only two different modes, kernel mode and user mode.
- kernel space above **TASK_SIZE** is forbidden in user mode.
- The switch from user to kernel mode is made by system calls.





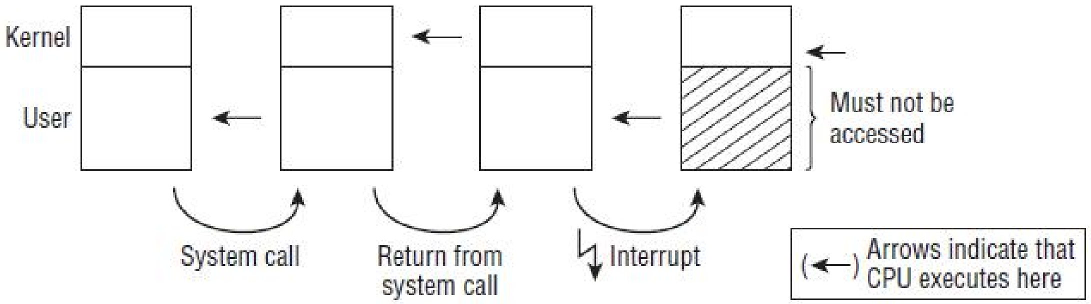
Privilege Levels, *Cont'd*

- Most of the time, the CPU executes code in userspace (add 2 numbers, acces variable in the process's memory allocated by the kernel to it,..).
- At this point, no intervention from kernel unless the process for example tried to access memory doesn't belong to it or did wrong calculations (***divide by 0***), so an exception will be raised and the Kernel here should take over of this.
- When the application performs a ***system call*** (trigger OS API), a switch to kernel mode is employed, and the kernel fulfills the request.
- During this, the kernel may access the user portion of the virtual address space, After the system call completes, the CPU switches back to user mode.



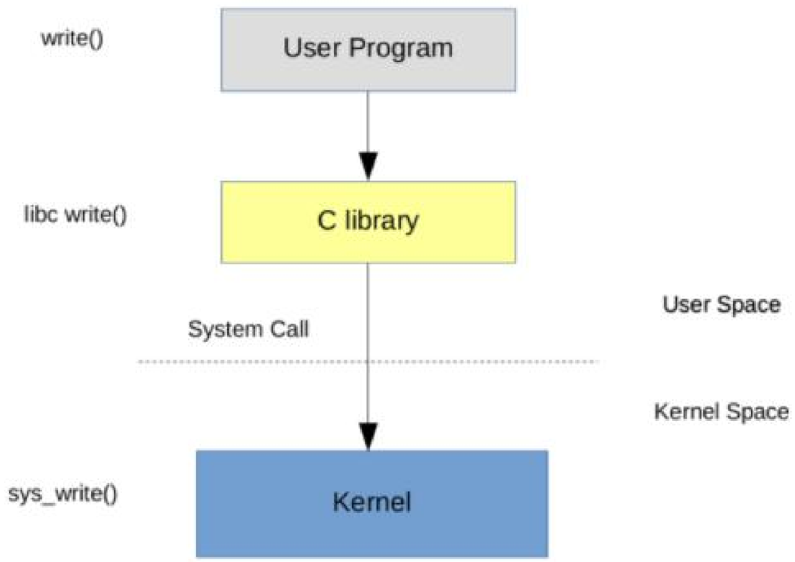
Privilege Levels, *Cont'd*

- A hardware interrupt may happen through this cycle as an impact to the user request or others, this interrupt also triggers a switch to kernel mode, but this time, the userspace portion must not be accessed by the kernel.



System Calls

- Enable user processes to interact with the kernel.
- Linux has POSIX compliant system calls.
- System calls are grouped into the following,
  - Process Management: Creating new tasks, querying information,...
  - Signals: Sending signals.
  - Files: Creating, opening, closing, reading from, writing to files, querying information and status.
  - Directories and Filesystem: Creating, deleting, and renaming directories, querying information, links, changing directories.
  - Protection Mechanisms: Reading and changing UIDs/GIDs, and namespace handling.
  - Timer Functions: Timer functions and statistical information.



Kernel Threads

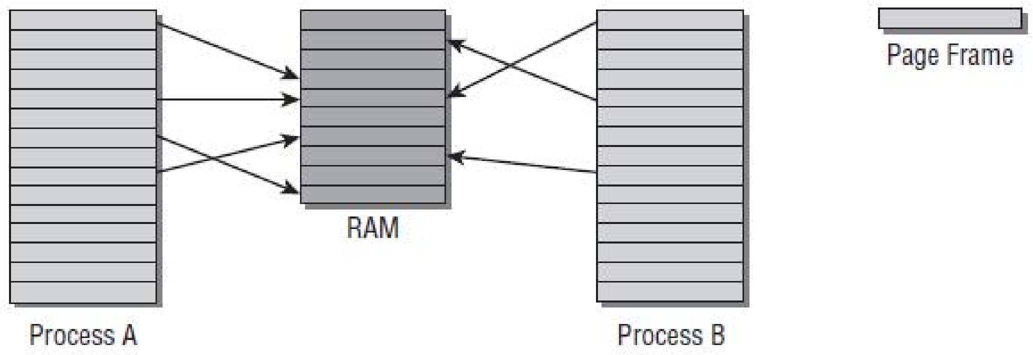
- Not associated with any userspace process.
- They are also tracked by the scheduler like every regular process.
- The kernel uses them for background kernel work, data synchronization of RAM, block devices,…
- All kernel threads are descendants of ***kthreadd*** **(PID 2)**.
- List the kernel threads spawned during the boot,

  $ ps -ef

```
karim_eshapa@karimeshapa-vm:~$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 16:38 ?        00:00:02 /sbin/init splash
root         2     0  0 16:38 ?        00:00:00 [kthreadd]
root         3     2  0 16:38 ?        00:00:00 [ksoftirqd/0]
root         4     2  0 16:38 ?        00:00:00 [kworker/0:0]
root         5     2  0 16:38 ?        00:00:00 [kworker/0:0H]
root         7     2  0 16:38 ?        00:00:00 [rcu_sched]
root         8     2  0 16:38 ?        00:00:00 [rcu_bh]
root         9     2  0 16:38 ?        00:00:00 [migration/0]
root        10     2  0 16:38 ?        00:00:00 [watchdog/0]
root        11     2  0 16:38 ?        00:00:00 [watchdog/1]
root        12     2  0 16:38 ?        00:00:00 [migration/1]
root        13     2  0 16:38 ?        00:00:00 [ksoftirqd/1]
root        15     2  0 16:38 ?        00:00:00 [kworker/1:0H]
root        16     2  0 16:38 ?        00:00:00 [watchdog/2]
root        17     2  0 16:38 ?        00:00:00 [migration/2]
root        18     2  0 16:38 ?        00:00:00 [ksoftirqd/2]
root        19     2  0 16:38 ?        00:00:00 [kworker/2:0]
root        20     2  0 16:38 ?        00:00:00 [kworker/2:0H]
root        21     2  0 16:38 ?        00:00:00 [watchdog/3]
root        22     2  0 16:38 ?        00:00:00 [migration/3]
root        23     2  0 16:38 ?        00:00:00 [ksoftirqd/3]
root        25     2  0 16:38 ?        00:00:00 [kworker/3:0H]
```

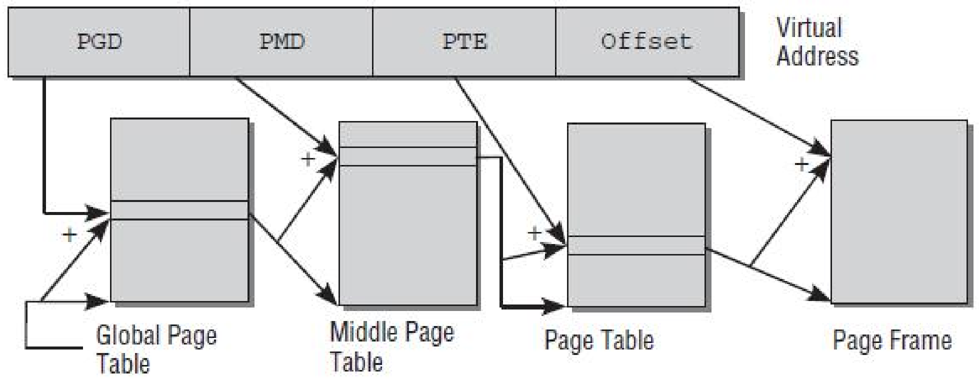
Process Virtual VS Physical Address Space

- Consider 2 processes, each process (A, B) sees a number of virtual pages that called "***Pages***".
- Some pages "Virtual pages" pointing to different **page frames** "Physical pages" in RAM.
- Page **5 of A** and page **1 of B** both point to the physical **page frame 5**.
- The kernel is responsible for mapping virtual address space to physical address space.
- The kernel and CPU must therefore consider how the physical memory that actually available, can be mapped onto virtual address areas using "**Page Table**" for each process.



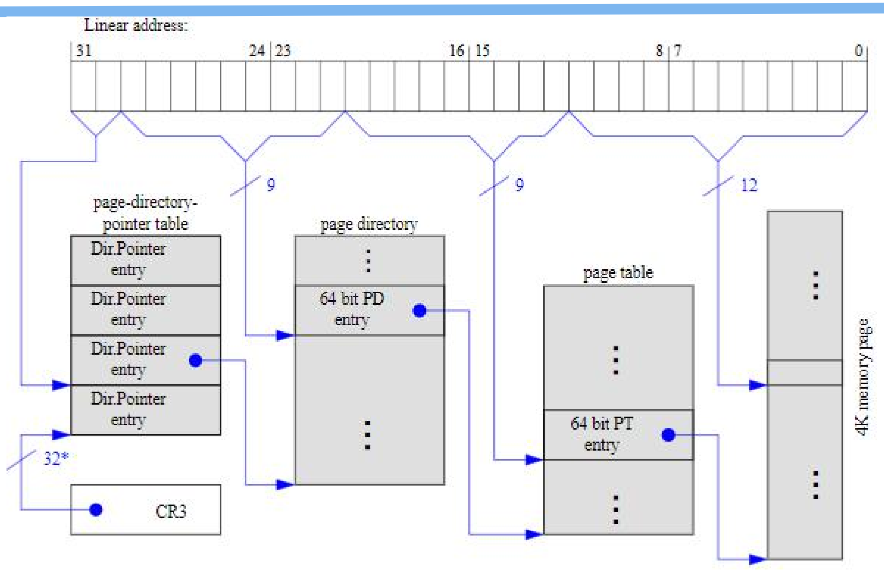
Page Table

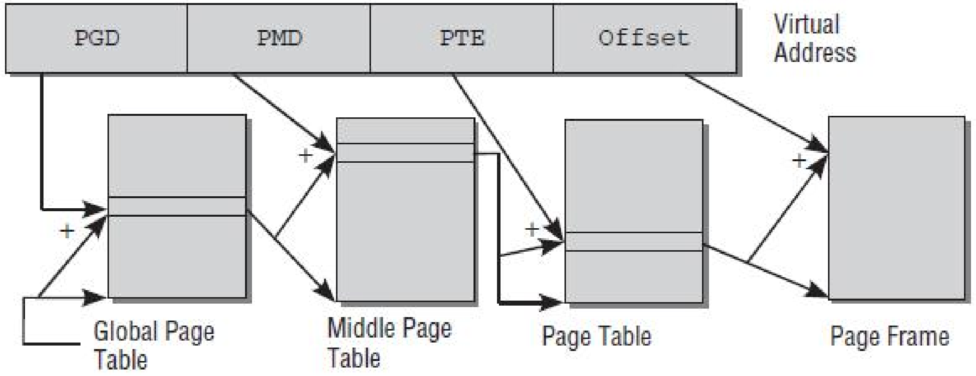
- IA-32 architecture uses, for example, page size 4 KB.
- For mapping virtual address space to physical address space, the easiest way of implementing the association between both would be to use an **array** containing an entry for each page in virtual address space.
- For example given RAM **4 GB** So, the virtual address space **4 GB**, this would produce an **array** with a **million entries**, and each process needs its own page tables, this would be **impossible**.
- So, page table is splitted into levels, most architectures offer, **three-level page table**, this will help also to allow unneeded areas to be ignored.
- The **virtual address** is represented as follows to extract the page frame "**Physical page**".



Page Table, *Cont'd*

- PGD: page global directory.
- PMD: page middle directories.
- PTE: page table entry.
- So, we have according to the **three-level page table** here, (2)^2 + (2)^9 + (2)^9 ~= (2)^10 Entries instead of (2)^20 in case of **single array**.
- This will be implemented using a fast CPU cache called a ***Translation Lookaside Buffer (TLB)***.





Process Memory Mapping

- In Linux environment, memory mapping is to map anything (files, devices,...) to the virtual address space that the process sees using system calls ***mmap***, ***munmap***.
- view a process's address space mapped, (ex. XORG PID 1009)

  $ sudo cat /proc/1009/maps

```
karim_eshapa@karimeshapa-vm:~$ sudo cat /proc/1009/maps
004b3000-0071c000 r-xp 00000000 08:01 2095       /usr/lib/xorg/Xorg
0071d000-0071e000 r--p 00269000 08:01 2095       /usr/lib/xorg/Xorg
0071e000-00725000 rw-p 0026a000 08:01 2095       /usr/lib/xorg/Xorg
00725000-00733000 rw-p 00000000 00:00 0
01daf000-02531000 rw-p 00000000 00:00 0          [heap]
aa82c000-aab69000 rw-p 00000000 00:00 0
aab69000-aaea6000 rw-s 00000000 00:05 2097164    /SYSV00000000 (deleted)
aaea6000-ab1e3000 rw-s 00000000 00:05 2064398    /SYSV00000000 (deleted)
ab344000-ab681000 rw-p 00000000 00:00 0
aba59000-aceea000 rw-p 00000000 00:00 0
aceea000-aeeea000 rw-s 00000000 00:05 950280     /SYSV00000000 (deleted)
aefdb000-af05b000 rw-s 00000000 00:05 2195467    /SYSV00000000 (deleted)
af05b000-af25b000 rw-s 00000000 00:05 1179657    /SYSV00000000 (deleted)
af25b000-af5cc000 rw-p 00000000 00:00 0
af5cc000-b05cc000 rw-s 00000000 00:05 393219     /SYSV00000000 (deleted)
b05cc000-b0f3c000 r-xp 00000000 08:01 7533       /usr/lib/i386-linux-gnu/dri/swrast_dri.so
b0f3c000-b0f8e000 r--p 0096f000 08:01 7533       /usr/lib/i386-linux-gnu/dri/swrast_dri.so
b0f8e000-b0f98000 rw-p 009c1000 08:01 7533       /usr/lib/i386-linux-gnu/dri/swrast_dri.so
```

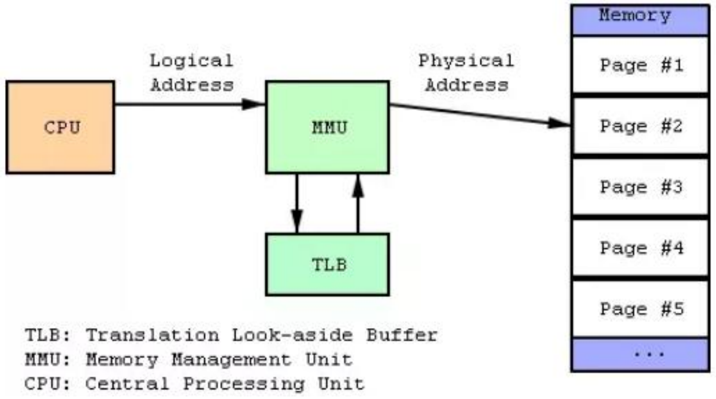
Process Memory Mapping, *Cont'd*

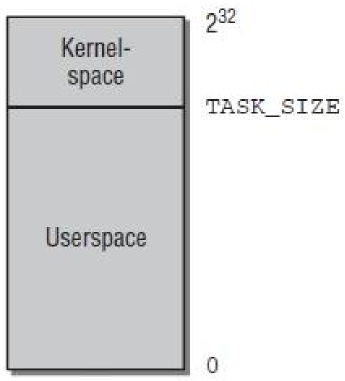
```
Ex.
address            perms  offset    dev    inode  pathname
0004B3000-0071C000 r-xp   00000000  08:01  2095   /usr/lib/xorg/Xorg
```

- **address**: Start and end address of the region in the process's address space.
- **permissions**: Describes how pages in the region can be accessed.
- **offset**: If the region was mapped from a file (using mmap),
  this is the offset in the file where the mapping begins.
- **device**: If the region was mapped from a file,
  this is the major and minor device number (in hex) where the file lives.
- **inode**: If the region was mapped from a file, this is the file number.
- **pathname**: If the region was mapped from a file, this is the name of the file

Virtual Address Space & Kernel Early Code

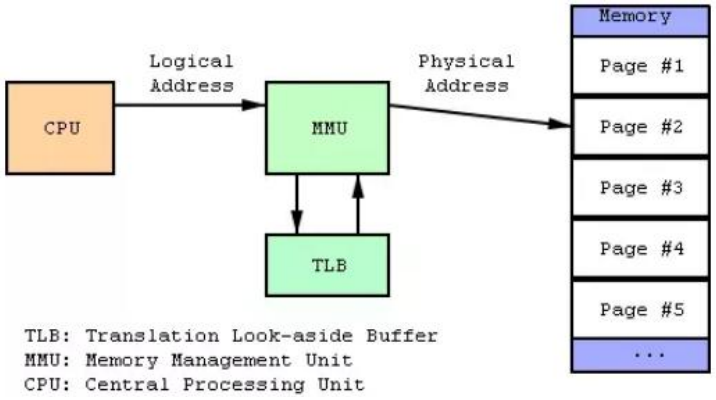
- Once the Bootloader calls the early code of the kernel "***assembly entry***",
- The first part does the following,
  - Create the initial ***page table*** for the kernel space to map itself before ***MMU*** is launched.
  - Turn on ***MMU***, then jumb to kernel initialization code.
  - Take care, Before ***MMU*** is turned on, every address issued by CPU is ***physical address***, while After ***MMU*** is turned on, every address issued by CPU is ***virtual address***.
  - So , an initial ***page table*** should be set up before turning on ***MMU*** as we already done by the first step.

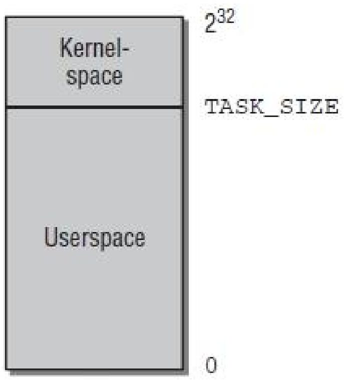




Virtual Address Space & Kernel Early Code, Cont'd

- The second part, Kernel initialization code overwrites the kernel space ***page table*** another time to mapping all the ***virtual kernel space*** to the ***physical memory space.***
  - Every time a user process is created, a new ***page table*** will be generated for that process, and the mapping done before for the virtual kernel space, will be **inserted** into a specific portion of this ***page table*** to have **a complete page table** describing the mapping of (Kernel, user virtual spaces).





Address Types

- **User virtual addresses:**

  User code runs in a process and each process has its own virtual address space, user space processes make full use of the ***MMU*** to get that virtual address space.

- **Physical addresses:**

  The addresses used between the processor and the system's memory.

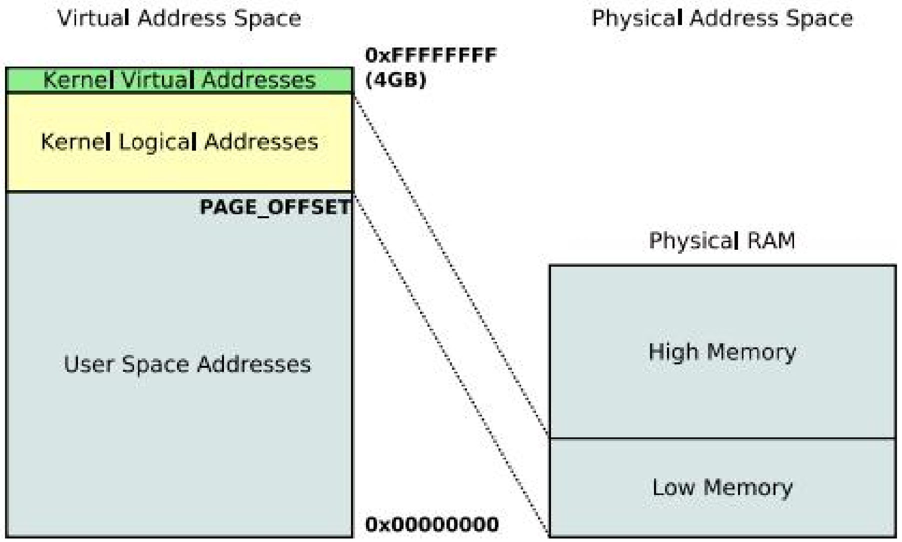
- **Bus addresses (HW Specific):**

  The addresses used between peripheral buses and memory.

  Often, they are the same as the physical addresses used by the processor, but that is not necessarily the case. Some architectures can provide an I/O memory management unit (IOMMU) that remaps addresses between a bus and main memory.

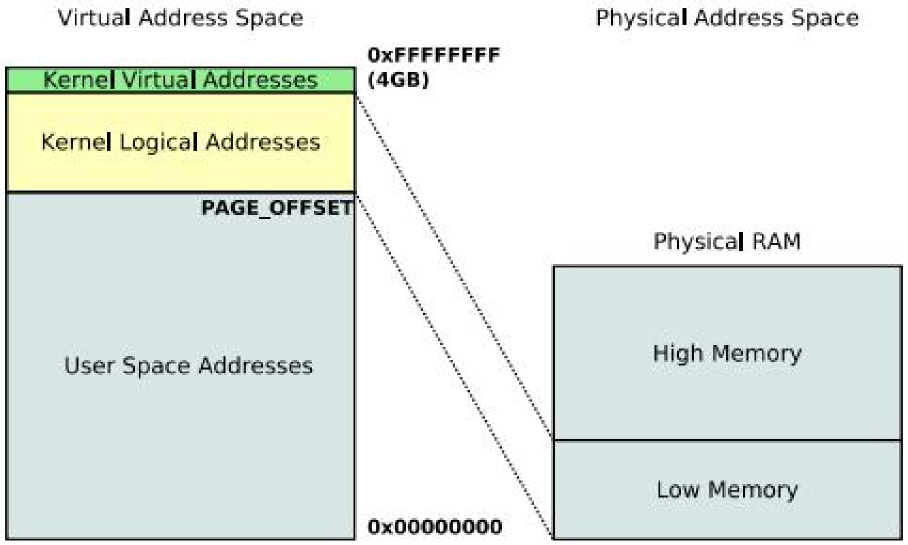
Address Types, *Cont'd*

- **Kernel logical addresses:**
  - On most architectures, logical addresses and their associated physical addresses differ only by a constant OFFSET.
    Memory returned from ***kmalloc()*** has a kernel logical address.
    i.e Kernel logical address = Physical address + PAGE_OFFSET.
    Virt: 0xC0000000 → Phys: 0x00000000
  - In a large memory situation (> 1G), only the bottom part of physical RAM (***Low Memory***) is mapped directly into kernel logical address space.
  - the ***kmalloc()*** allocation will be physically **<u>contiguous</u>** and so that, it shouldn't be used for large buffers because most probably can't get the huge buffer size physically available in this area.



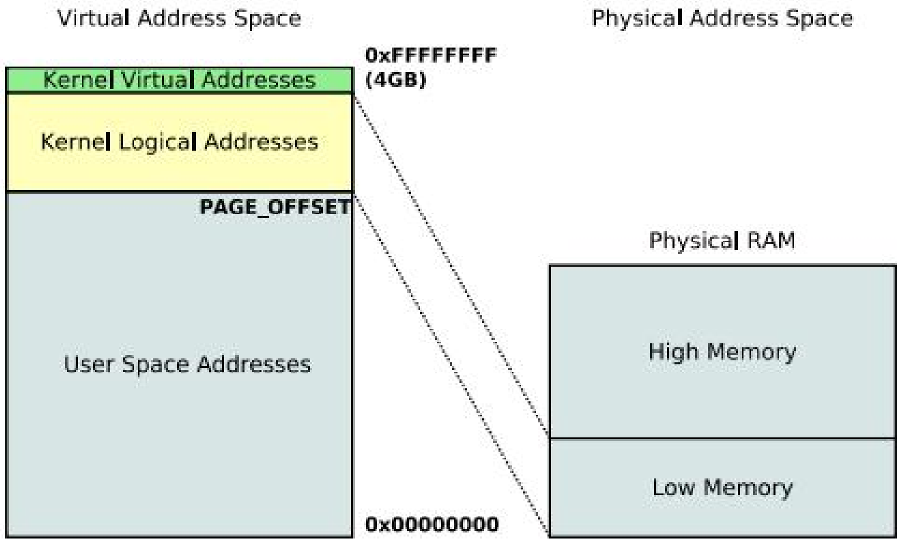
Address Types, *Cont'd*

- **Kernel virtual addresses:**
  - Kernel virtual addresses are similar to logical addresses in that they are a mapping from a kernel-space address to a physical address.
  - But, the kernel virtual addresses do not necessarily have the linear, one-to-one mapping to physical addresses.
  - memory allocated by ***vmalloc()*** has a virtual address (but no direct physical mapping).
- The Kernel virtual addresses mapping could be go to the ***Low Memory*** or the ***High Memory*** up on request.
- The ***vmalloc()*** allocation will be physically **<u>non-contiguous</u>**.



High and Low Memory

- With 32 bits, it is possible to address 4 GB of memory (***Kernel started with very small memory***), but in response to commercial pressure to support more memory **while not breaking 32- bit application** and the **system's compatibility**, the processor manufacturers have added "address extension" features to their products. The result is that, in many cases, even 32-bit processors can address more than 4 GB of physical memory.
- So, the kernel divides the **4GB** memory i.e IA32 in that case physically into ***Low memory*** and ***high memory*** such that,
  - ***Low memory:*** is a ***physically contiguous*** around **896MB** and it's a **physical memory which has a kernel logical address**.
  - ***High memory:*** the kernel decides it will be the **physical memory beyond ~896MB** and **has no logical address** and **Not physically contiguous used in the kernel**.

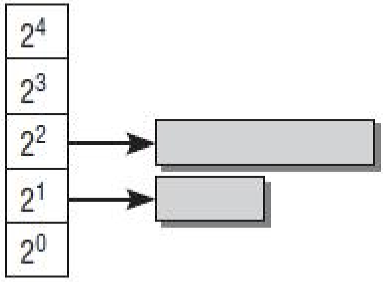


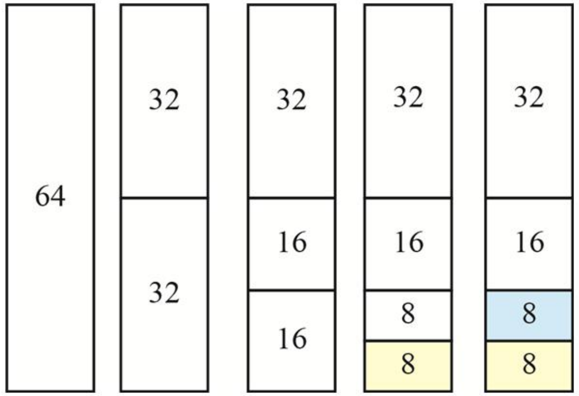
Buddy System

- Memory blocks in the system are always grouped as two buddies.
- Ex-1.If the system requires 8 page frames (32 KB), it splits the block consisting of 16 page frames (64 KB) into two buddies. While one of the blocks is passed to the application that requested memory, the remaining 8 page frames (32 KB) are placed in the list for 8-page (32 KB) memory blocks.
- Ex-2.If the system requires 5.5 KB, and the minuimum memory block can be allocated is 8 KB. so, the system will allocate the whole 8 KB, so we have 2.5 KB can't be linked to any list and can't be used.
- Advantage: Coalescing, Disadvantage: Internal fragmentation.
- The kernel uses sort of ***Memory Compaction*** automatically to overcome such framgmentation. To force compaction,

  $ sudo su

  $ echo 1 > /proc/sys/vm/compact_memory

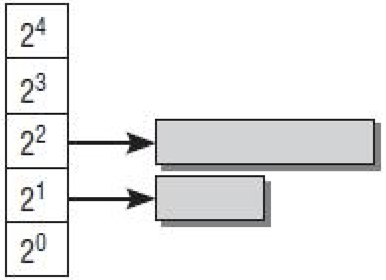




Slab Cache

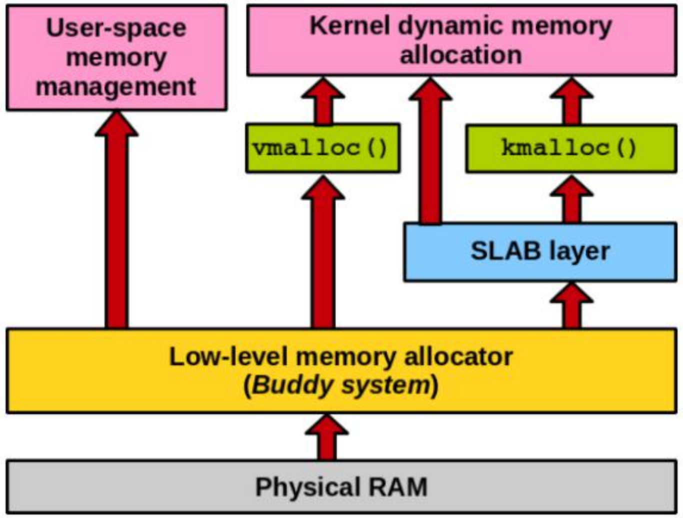
- Usually Linux Buddy System lists has allocation resolution up to 1 page.
- The Kernel itself needs memory blocks (Different Kernel Obj) much smaller than a whole page frame.
- So, Intermediate layer called ***Slab layer*** is injected to provide smaller chunks of memory to be allocated.
- For more info about slabs allocated in the system,

  $ sudo slabtop -o




```
karim_eshapa@karimeshapa-vm:~$ sudo slabtop -o
 Active / Total Objects (% used)    : 394340 / 397510 (99.2%)
 Active / Total Slabs (% used)      : 9907 / 9907 (100.0%)
 Active / Total Caches (% used)     : 70 / 97 (72.2%)
 Active / Total Size (% used)       : 79084.09K / 79776.66K (99.1%)
 Minimum / Average / Maximum Object : 0.01K / 0.20K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
 83074  83074 100%    0.05K   1138       73      4552K buffer_head
 82240  82240 100%    0.12K   2570       32     10280K dentry
 56856  56856 100%    0.70K   2472       23     39552K ext4_inode_cache
 23040  23040 100%    0.03K    180      128       720K ext4_extent_status
 22144  21090  95%    0.03K    173      128       692K kmalloc-32
 20202  20040  99%    0.10K    518       39      2072K vm_area_struct
 18200  18200 100%    0.07K    325       56      1300K kernfs_node_cache
 12800  12800 100%    0.02K     50      256       200K kmalloc-16
 11462  11375  99%    0.35K    521       22      4168K inode_cache
  9435   8222  87%    0.05K    111       85       444K anon_vma
  7497   7410  98%    0.19K    357       21      1428K kmalloc-192
  7488   7488 100%    0.30K    288       26      2304K radix_tree_node
  6720   6720 100%    0.06K    105       64       420K kmalloc-64
  4560   4491  98%    0.38K    228       20      1824K proc_inode_cache
  4096   4096 100%    0.01K      8      512        32K kmalloc-8
  4080   4080 100%    0.02K    _24      170        96K Acpi-Namespace
```



Slab Cache, Cont'd

- For the ***highlighted*** entry,

  The kernel ***OBJ*** called ***inode_cache*** has a size ~= 0.35 KB

  #of ***OBJS*** = 11462

  CACHE SIZE = 11462 * .35 KB ~= 4168 KB

  Each slab (***OBJ/SLAB***) has 22 ***OBJ*** and #of slabs (***SLABS***) 521 so, #of obj's (***OBJS***)= 11462

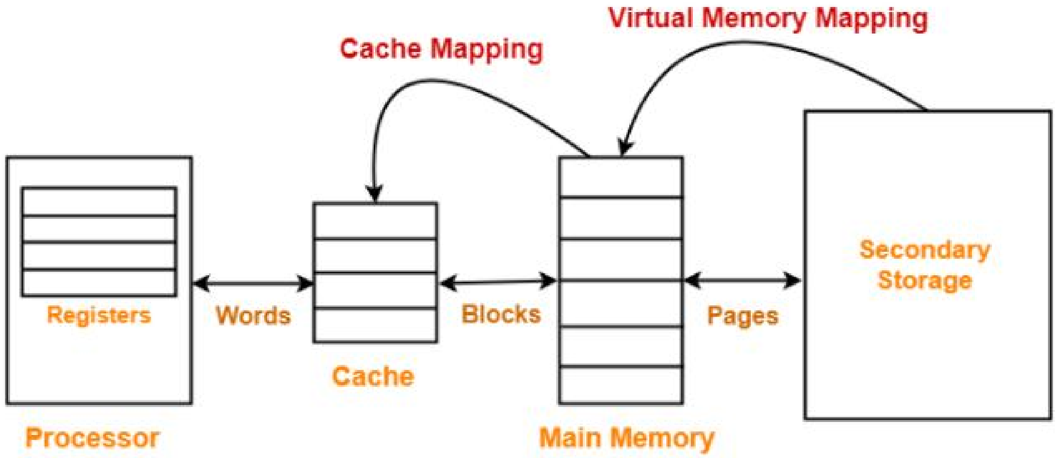
  Active obj's (***ACTIVE***): means those are already in use.

```
karim_eshapa@karimeshapa-vm:~$ sudo slabtop -o
 Active / Total Objects (% used)    : 394340 / 397510 (99.2%)
 Active / Total Slabs (% used)      : 9907 / 9907 (100.0%)
 Active / Total Caches (% used)     : 70 / 97 (72.2%)
 Active / Total Size (% used)       : 79084.09K / 79776.66K (99.1%)
 Minimum / Average / Maximum Object : 0.01K / 0.20K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
 83074  83074 100%    0.05K   1138       73      4552K buffer_head
 82240  82240 100%    0.12K   2570       32     10280K dentry
 56856  56856 100%    0.70K   2472       23     39552K ext4_inode_cache
 23040  23040 100%    0.03K    180      128       720K ext4_extent_status
 22144  21090  95%    0.03K    173      128       692K kmalloc-32
 20202  20040  99%    0.10K    518       39      2072K vm_area_struct
 18200  18200 100%    0.07K    325       56      1300K kernfs_node_cache
 12800  12800 100%    0.02K     50      256       200K kmalloc-16
 11462  11375  99%    0.35K    521       22      4168K inode_cache
  9435   8222  87%    0.05K    111       85       444K anon_vma
  7497   7410  98%    0.19K    357       21      1428K kmalloc-192
  7488   7488 100%    0.30K    288       26      2304K radix_tree_node
  6720   6720 100%    0.06K    105       64       420K kmalloc-64
  4560   4491  98%    0.38K    228       20      1824K proc_inode_cache
  4096   4096 100%    0.01K      8      512        32K kmalloc-8
  4080   4080 100%    0.02K     24      170        96K Acpi-Namespace
```

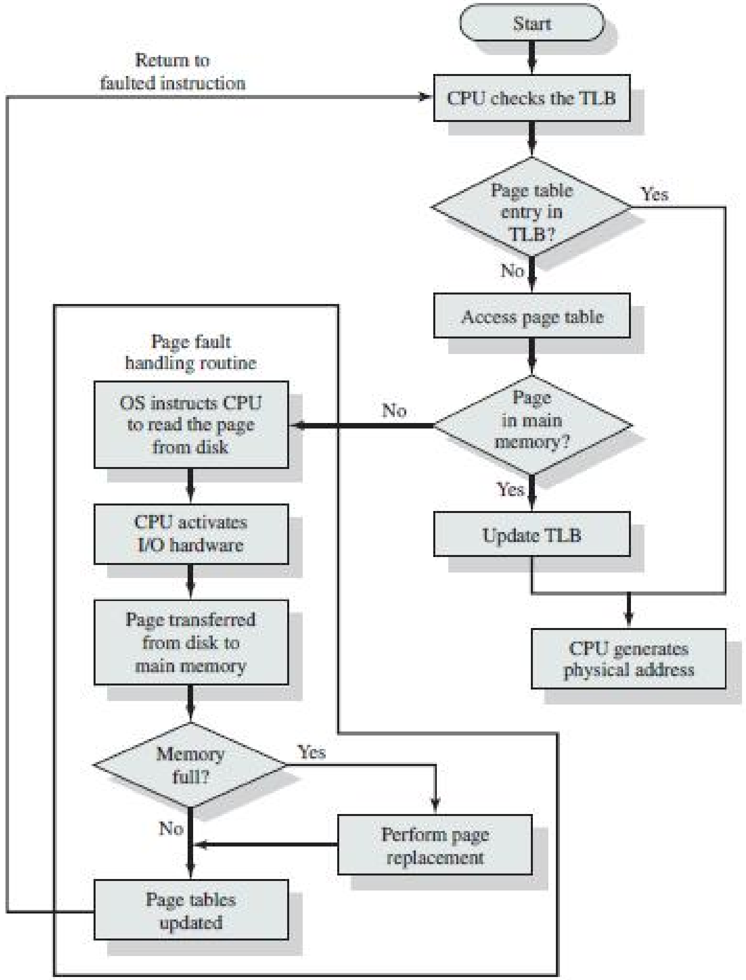
Cache Memories & Swapping

- CPU has a closer and fast memory called ***Cache*** memory.
- ***Caches*** mapping some lines from main memory **RAM** to esnsure fast acces to CPU.
- Cache mapping types
  - Direct Mapping.
  - Fully associative Mapping.
  - K-way set associative Mapping.
- One of the most familiar Cache replacement policies is the ***LRU (Least Recently Used)***.
- Like the RAM, Cache memories are organized from kernel perspective in pages as well called "***Page cache***".
- we will take about caches later in details from kernel perspective.



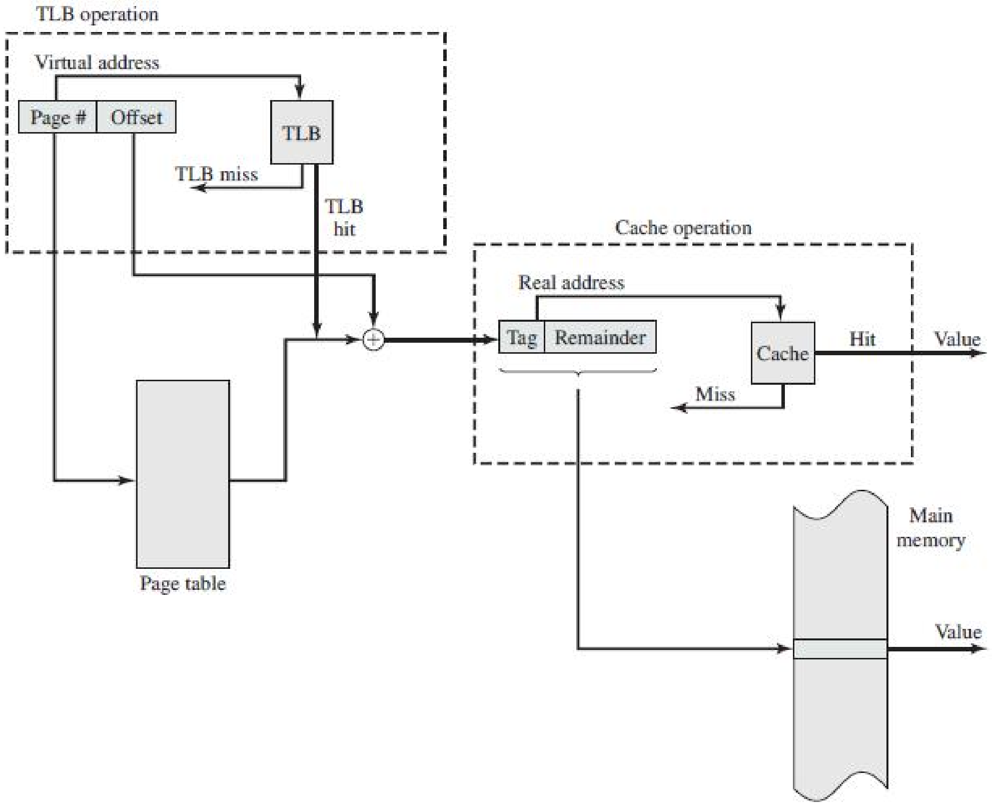
Cache Memories & Swapping, Cont'd

- what happens when the kernel doesn't have any more physical memory left?, ***Swapping** is the answer.*
- **Swapping**: Enables available RAM to be **enlarged virtually** by using disk space as extended memory.
- The process of writing pages out to disk to free memory is called ***swapping-out***.
- If later a ***page fault*** is raised because the page is on disk, in the swap area rather than in memory, then the kernel will read back in the page from the disk and satisfy the page fault and this is called ***swapping-in***.
- Let's take the TLB cache as an example.



Cache Memories & Swapping, Cont'd

- What happened with ***TLB*** Caches, is happening with the ***CPU Caches*** and this is the whole picture.
- When we can't find the page in HW Caches (TLB or CPU Caches), we call it ***Cache Miss***.
  If we can get it from the HW caches, we call it ***Cache Hit***.
- For exploring the virtual memory generally and ***Swapping Memory***.
  $ vmstate
  si: Amount of memory swapped in from disk.
  so: Amount of memory swapped to a block device.



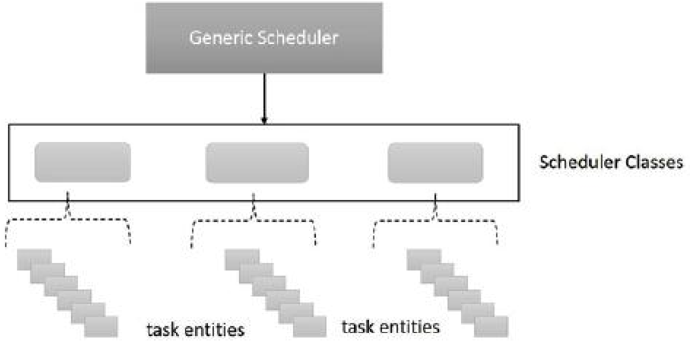

```
karim_eshapa@karimeshapa-vm:~$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 22896084  93092 1042268    0    0     6     8   44  138  2  0 97  0  0
```

Process Scheduler

- The process scheduler is the core component of the kernel, which computes and decides when and for how long a process gets CPU time.
- The decision of which process to run depends on the **priority** of the process.
- Priorities are fundamentally classified into **dynamic** and **static** priorities.
- **Dynamic priorities** are basically applied to normal processes dynamically by the kernel, considering various factors such as the **nice value** of the process, its **historic behavior** (I/O bound or processor bound), **lapsed execution**, and **waiting time**.
- The ***nice value*** of any normal process ranges between 19 (lowest ***priority***) and -20 (highest ***priority***), with 0 being the default value.
- Higher ***nice value*** indicates a lower priority (the process is being nicer to other processes).
- **Static priorities** are applied to **real-time** processes by the user and the kernel does not change their priorities dynamically regardless any historic behavior.

Process Scheduler, *Cont'd*

- Real-time processes are prioritized between 0 and 99 (static priority).
- So, the ***Dynamic priority*** for normal process range will be 100 - 139 (MAX_RT_PRIO + 40).
- Linux scheduler has **3** main scheduling ***classes*** for processes.
  - Completely Fair Scheduling class (CFS).
  - Real-Time Scheduling class.
  - Deadline Scheduling class.



- Real-Time processes are always given preference over normal tasks.
- Linux kernel is a ***preemptive***, that means, it is possible to preempt a task at any point and then reschedule.
- For some process statistics, (ex. XORG PID 1042)

  $ cat /proc/1042/sched

  Policy: 0 CFS (SCHED_NORMAL)

  Priority: 120 (Between 100 - 139)

```
karim_eshapa@karimeshapa-vm:~$ cat /proc/1042/sched
Xorg (1042, #threads: 5)
-------------------------------------------------------------------
se.exec_start                                :     13052687.949225
se.vruntime                                  :        27485.028668
se.sum_exec_runtime                          :        92151.639112
se.statistics.sum_sleep_runtime              :     12942044.087055
se.statistics.wait_start                     :            0.000000
se.statistics.sleep_start                    :     13052687.949225
se.statistics.block_start                    :            0.000000
se.statistics.sleep_max                      :        59978.213708
se.statistics.block_max                      :           52.626211
se.statistics.exec_max                       :           15.397464

 :
 se.avg.load_sum                             :             2291328
 se.avg.util_sum                             :             1910482
 se.avg.load_avg                             :                  42
 se.avg.util_avg                             :                  34
 se.avg.last_update_time                     :      19829253249934
 policy                                      :                   0
 prio                                        :                 120
 clock-delta                                 :                 121
```

Process Scheduler, *Cont'd*

- Process scheduling **states**,
  - **Running/Runnable** (R):

    Running: means it's running on CPU right now.

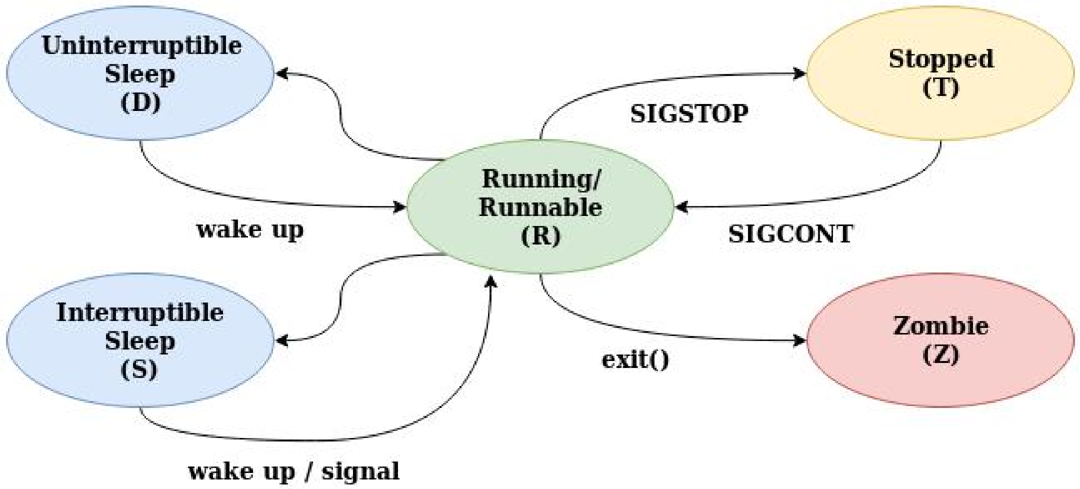
    Runnable: means it's ready to be run (***Ready*** state).
  - **Sleeping (S)**: waiting for a resource (***Waiting*** state)

    1.Interruptible: it will wake up to handle signals.

    2. Uninterruptible: Can't wake it up by signals.
  - **Stopped (T)**: a process becomes stopped when

    it receives the ***SIGSTOP signal***.
  - **Zombie (Z)**: process whose execution is completed

    and exit() but it still has an entry in the process table.



Time Management

- Kernel provides measuring time and time differences at various points.
- Time is represented in three different ways
  - **Wall time** (or real time): This is the actual time and date in the real.
  - **Process time**: This is the time consumed by a process in its life span.
  - **Monotonic time**: This is the time elapsed since system bootup.
- **Jiffies** variable is the famous time base, holding the number of ticks elapsed since system bootup.
- **jiffies_64** variable and its **32-bit** counterpart jiffies are incremented with frequency **HZ** "Tick rate".
- Usually **jiffies** are incremented between 1,000 and 100 times per second (Tick **1ms** or **100us**).
- If nanoseconds time is needed, so **high-resolution** timers are used.
- For more info about timers for each CPU in the system,

  $ cat /proc/timer_list

```
karim_eshapa@karimeshapa-vm:~$ sudo cat /proc/timer_list
[sudo] password for karim_eshapa:
Timer List Version: v0.8
HRTIMER_MAX_CLOCK_BASES: 4
now at 18759005210109 nsecs

cpu: 0

:
active timers:
  .expires_next   : 18759008000000 nsecs
  .hres_active    : 1
  .nr_events      : 290937
  .nr_retries     : 92
  .nr_hangs       : 0
  .max_hang_time  : 0
  .nohz_mode      : 2
  .last_tick      : 18758996000000 nsecs
  .tick_stopped   : 0
  .idle_jiffies   : 4614749
  .idle_calls     : 698406
  .idle_sleeps    : 606733
  .idle_entrytime : 18759004767700 nsecs
  .idle_waketime  : 18759003736387 nsecs
  .idle_exittime  : 18759003739176 nsecs
  .idle_sleeptime : 18291129173034 nsecs
  .iowait_sleeptime: 5434078536 nsecs
  .last_jiffies   : 4614752
  .next_timer     : 18759100000000
  .idle_expires   : 18759100000000 nsecs
jiffies: 4614752
```

Interprocess Communication

- Processes need various resources to communicate, share data, and synchronize their execution to achieve desired results.
- These are provided by the operating system's kernel as services called interprocess communication (**IPC**).
- Linux provides different ***IPC Methods***,
  - Signals.
  - Pipes and FIFOs.
  - Message queues.
  - Shared memory.

IPC Methods

- Signals
  - Any process can be notified of a system event asynchronously.
  - Also prcesses use it as communication mechanism to notify each others with certain event.
- Signal types

  $ kill -l

```
karim_eshapa@karimeshapa-vm:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- ***2) SIGINT***: Interrupt from keyboard.

  18) SIGCONT: Continue process if stopped.

  9) SIGKILL: Kill running process.

  19) SIGSTOP: Stop process.
- we will talk about them later in details.

IPC Methods, *Cont'd*

- Pipes and FIFOs
  - Pipes and FIFOs provide a ***unidirectional*** interprocess communication channel.
  - Pipes and FIFOs are created and managed by a special filesystem called ***pipefs***.
  - A pipe has a read end and a write end, and returns pair of file descriptor ***fd***, one for the read end and the other for the write end, ***struct fd_pair pipe().***
  - Pipes may be considered open files that have no corresponding image in the mounted filesystems.
  - FIFOs is similar to a pipe, except that it is accessed as part of the filesystem but has no content and it is entered into the filesystem by calling ***mkfifo()***.
  - Shell uses **|** character to transport the output of one process as input for another process.

    $ cat file-name | more

    more: limting the output to only one page.

IPC Methods, *Cont'd*

- Message queues
  - Allow processes to exchange data in the form of messages.
  - An internal kernel message queue structure (***mqd_t***) refers to the open message queue between processes.
- Shared memory
  - Allows processes to communicate information by sharing a region of memory.
  - It creates a ***shared memory object*** need to be mapped to the processes's virtual address space that use the shared memory for communication.

    shm_open() then ***mmap()***.

Synchronization and Locking

- Kernel should enable concurrent access to kernel services and data structures.
- Kernel code that accessing global data structures need to be synchronized to ensure consistency and validity of shared data.
- Linux provides different **synch and locking** Methods,
  - Atomic operations.
  - Spinlocks.
  - Standard mutexes.
  - Wait/wound mutex.
  - RCU (Read Copy Update).
  - Semaphores.
  - Completions.

Synch and Locking Methods

- Atomic operations
  - Atomicity guarantees indivisible and uninterruptible execution of the operation initiated.
  - Most CPU instruction set architectures define instruction opcodes that can perform atomic read-modify-write operations on a memory location.
- Spinlocks
  - Data structures should be protected from being concurrently accessed by kernel control paths that run on different CPUs.
  - Once trigger lock routine it spins at this point of code with the interrupts disabled on the specific coreuntil obtaining the lock.
- Standard mutexes
  - Busy-waiting the calling process for an indefinite duration for releasing the lock.

Synch and Locking Methods, *Cont'd*

- Wait/wound mutex
  - Imagine two threads (we'll call them **T1** and **T2**) that attempt to lock **2** buffers in the opposite order: **T1** starts with ***Buffer A***, while **T2** starts with ***Buffer B***.
  - a **Dead Lock** will happen, and both threads will stuck.
  - Using Wait/wound mutex is the solution for such a case, as The thread that "got there first" will simply sleep until the remaining buffer becomes available.
  - If **T1** started the process of locking the 2 buffers first, it will be the thread that waits.
  - The other thread will be "**wounded**," meaning that it will be told it must release any locks it holds and start over from scratch and wait until that lock becomes available again.

```
T1
{
    get_mutex(a);
    acces_buff(A);
    ...
    get_mutex(b);
    access_buff(B);
    ...
    release_mutex(a);
    release_mutex(b);
}

T2
{
    get_mutex(b);
    acces_buff(B);
    ...
    get_mutex(a);
    access_buff(A);
    ...
    release_mutex(b);
    release_mutex(a);
}
```

Synch and Locking Methods, *Cont'd*

- RCU (Read Copy Update)
  - Designed to protect data structures that are mostly accessed for reading by ***several CPU's***.
  - Allows ***many readers*** and ***many writers*** to proceed concurrently.
  - Limiting the scope of ***RCU***,

    1. Only data structures that are ***dynamically*** allocated and referenced by means of ***pointers*** can be protected by RCU.

    2. No kernel control path can ***sleep*** inside a critical region protected by ***RCU***.
  - when a writer wants to update the data structure, it dereferences the pointer and makes a ***copy*** of the whole data structure. Next, the writer modifies the copy. Once finished, the writer changes the pointer to the data structure to make it point to the updated copy.

Synch and Locking Methods, *Cont'd*

- RCU (Read Copy Update), *Cont'd*
  - The old copy of the data structure cannot be freed right away when the writer updates the pointer because the readers that were accessing the data structure when the writer started its update could still be reading the old copy.
  - The old copy will be destroyed in a ***deffered*** work after making sure that all the readers finished.

Synch and Locking Methods, *Cont'd*

- Semaphores
  - A semaphore is simply a counter associated with a data structure; it is checked by all kernel threads before they try to access the protected region and if it can't take the semaphore, it will ***wait for interval*** of time and the semaphore will keep all the threads that went to ***wait*** state because of the semaphore in a specific **list** contained in the semaphore structure, and once the semaphore is released, the semaphore uses this **list** to wake them **one by one**.
- Completions
  - If you have one or more threads that must wait for some kernel activity to be reached a point or a specific state, we can use the completion API's.
  - It looks like a semaphore with ***waiting interval*** but Completion can wake the waited threads all at once by ***complete_all()***.

Interrupts and Exceptions

- Interrupts and Exceptions
  - Intel documentation classifies interrupts and exceptions as follows,
  - Interrupts:

    Maskable interrupts: Most of the interrupts.

    Nonmaskable interrupts: i.e Reset interrupt.
  - Exceptions:

    Processor-detected exceptions:

    Faults: i.e Page Fault Exception.

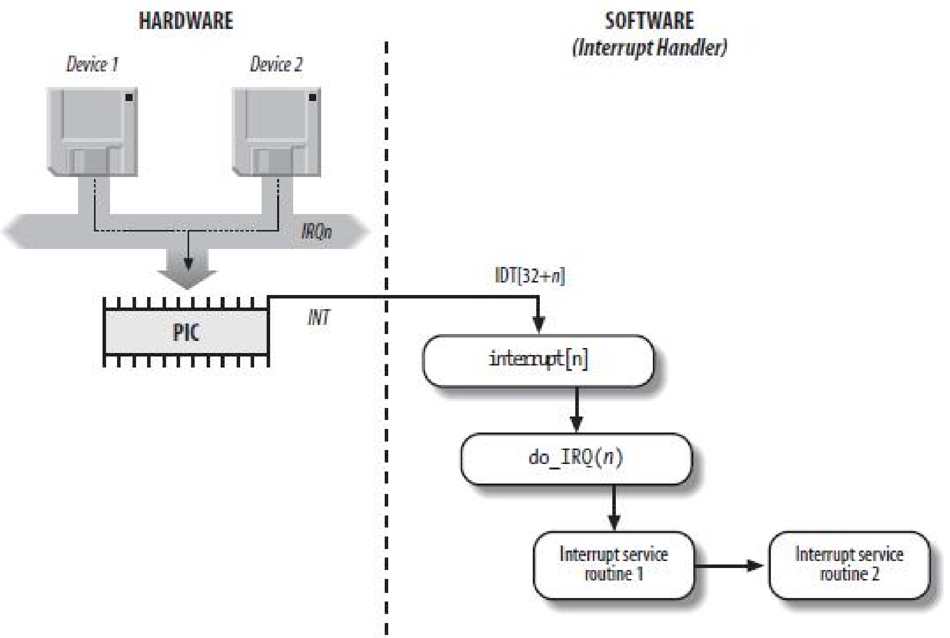
    Traps: Used in debugging, i.e a breakpoint has been reached within a program.

    Aborts: Report severe errors, i.e hardware failures.

    Programmed exceptions: User triggers an intended exception and this is used in i.e system calls assembly instruction, ***int3***.

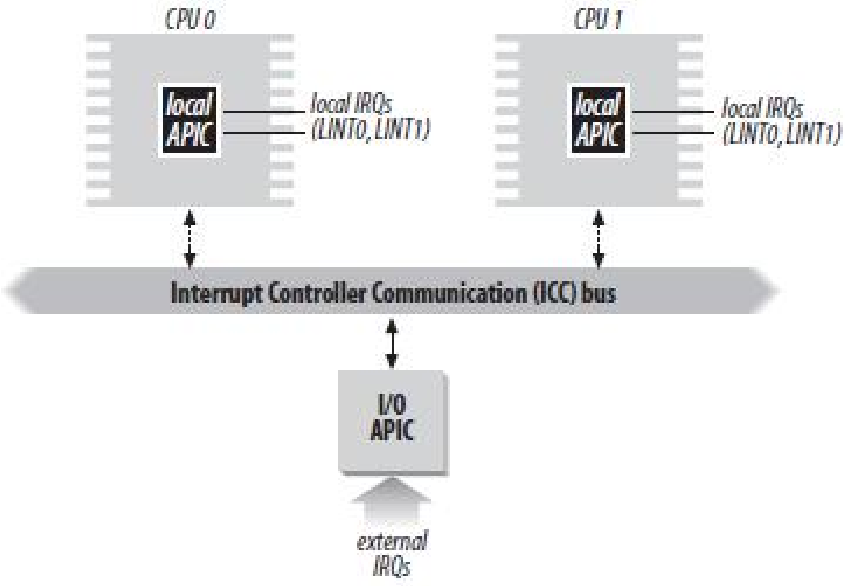
Interrupts and Exceptions, *Cont'd*

- IRQs and Interrupts
  - Each hardware device controller capable of issuing interrupt requests usually has a single output line designated as the Interrupt Request (IRQ) line.
  - All existing IRQ lines are connected to the input pins of a hardware circuit called the Programmable Interrupt Controller (**PIC**).
- Interrupt Descriptor Table (**IDT**)
  - Associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler.
  - The ***do_IRQ( )*** function is invoked to execute all interrupt service routines associated with an interrupt



Interrupts and Exceptions, *Cont'd*

- Advanced Programmable Interrupt Controller (**APIC**)
  - If the system includes two or more CPUs, so we have a different approach to the external **IRQ**.
  - I/O APIC at that time acts as a router with respect to the local APIC's.



Deferred Work

- Softirq (Software interrupt request)
  - There is some operations need to be done in the real HW interrupt handler but it takes time, so we need to ***defer*** them to be handled.
  - Softirq can be called a ***deffered*** interrupt so, softirq runs in an interrupt context and thus, softirq can't ***sleep***.
  - The ***softirq*** is ***reentrant***, so the same softirq can run on several CPUs at the same time.
  - Sometimes the real HW interrupt called ***top half***, and softirq called ***bottom half***.
- Tasklets
  - Tasklet handlers are executed by softirqs.
  - But tasklets are ***non reentrant***, so the same tasklet handler ***can never*** run concurrently.
- Workqueues
  - Work queues are considerd as a form of ***deferring*** work executed by special kernel threads.
  - So, work queues are schedulable and can therefore ***sleep***.
  - it runs in a process context unlike Softirq.

Device Drivers

- Kernel provides a bunch of drivers for the different HW (Hard disks, Interfaces, USB, Sound Cards,...).
- In Linux, ***Everything is a file***, when we deal with any device, we write to file and read from a file.
- Linux device drivers are grouped into 3 classes from **Microscopic** view,
  - **Character** device driver: Deal with the devices that allow data to be read and written character-by-character, like (input, sound, graphics, serial,...).
  - **Block** device driver: Deal with the devices that allow data to be read or written only in multiples of block units, like (Hard disk, CDROM,...).
  - **Network interface** driver: An "interface" means communication with HW device through an interface bus like "**eth0**,.." not a file. Communication between the kernel and a network device driver is completely different from that used with **char** and **block** drivers. Instead of read and write, the kernel calls functions related to **packet** transmission.
- There is a **Macroscopic** view for Linux device drivers (***Device Model***, ***Device Drivers Frameworks***, ***Device trees***).

Device Model

- The device model provides a single mechanism for representing devices and describing their topology in the system.
- Benifits
  - Minimization of code duplication.
  - Generate a complete and valid tree of the entire device structure of the system, including all buses and interconnections.
  - The capability to categorize devices by their class, such as input devices.
- The main elements of the device model
  - ***device***: Each HW or (*a **virtual thing** dealt the same as HW*) should contain "device" structure that represnt it to the system as a new device. it appears under /sys/devices/.
  - ***device_driver***: Each HW or more than one HW should be handled with a "devie_driver".

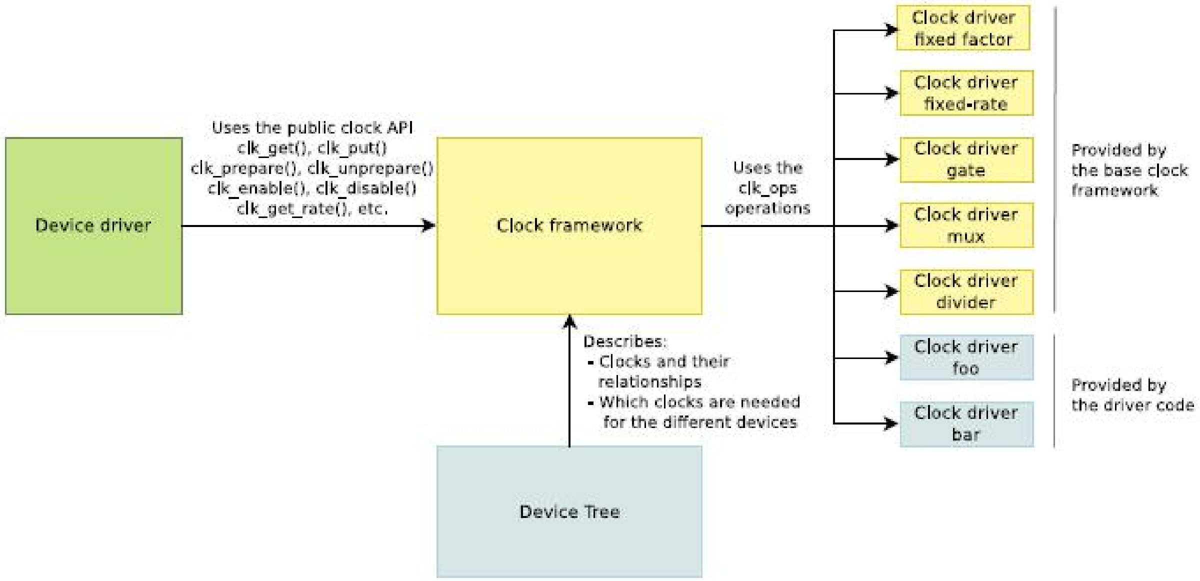
Device Model, *Cont'd*

- The main elements of the device model, *cont'd*
  - ***bus_type***: Each HW or (a ***virtual thing*** *dealt the same as HW*) attached to certain bus "bus_type" (USB, I2c,…). it appears under /sys/bus/.
  - ***class***: Each HW or (a ***virtual thing*** *delt the same as HW*) belongs to a "class", this class has a group of ***attributes*** to be implemented like ("leds" class supports the blinking, flashing, and brightness control features of physical LED's, "disk" class supports block sizes and I/O, removable). it appears under */sys/class/*.

*Note: (a **virtual thing** dealt the same as HW), this statement means, there is special cases of specific kernel structures can have a struct "device" and define a "bus_type" but don't have a device driver in any sense and the kernel needs them to be exposed in the hierarchy i.e "workqueue" .*
/sys/devices/virtual/workqueue
/sys/bus/workqueue
So, we can coclude the defintion of what a device is from kernel perspective!
*the answer is, "Anything has a **device** structure".*

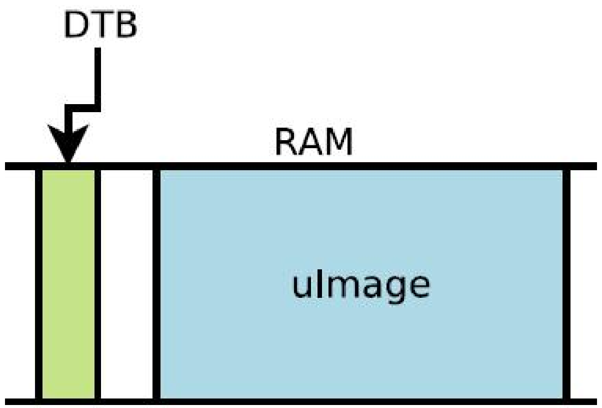
Device Drivers Frameworks

- At this moment every ***device_driver*** implementation still not restricted to a common standard or shape to provide its functionalities.
- So, the Framework "Abstract Layer" comes to provide a coherent "Framework API's" to user space.
- It very looks like the abstract "***VFS***" layer with different "***Filesystems***".
- So, we expect from each device to point to the Framework that belongs to.
- And also each ***device_driver*** registers with the belonging framework to implement the Framework API's.
- i.e "USB network adapter", it needs sort of USB driver and also needs to be logically under network framework, so, It's categorized in the kernel that way, */drivers/net/usb*.
- i.e clk framework.



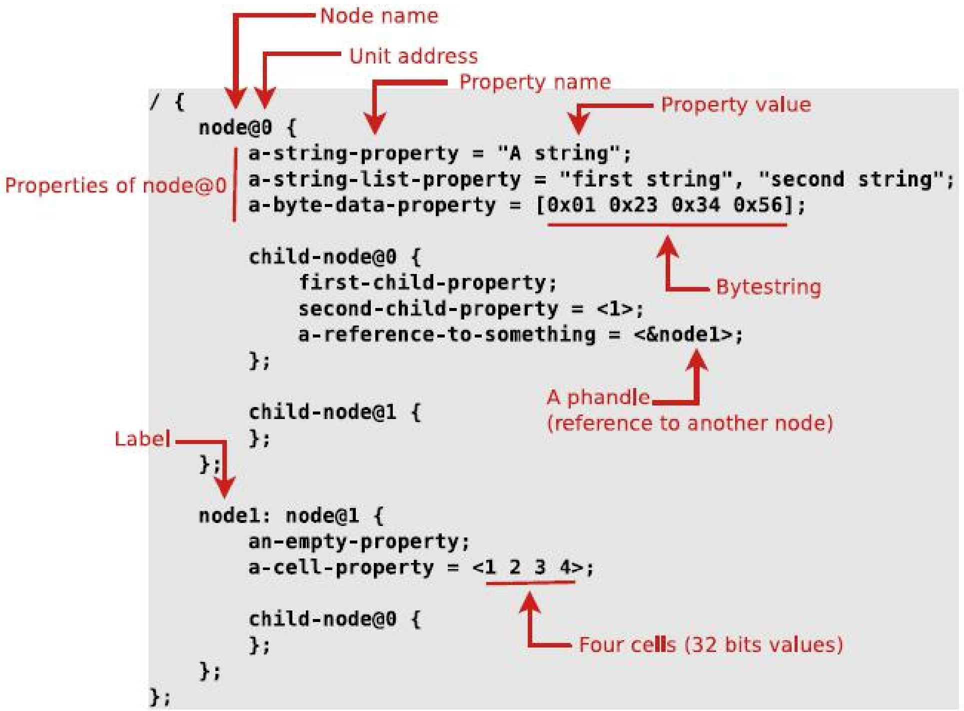
Device Trees

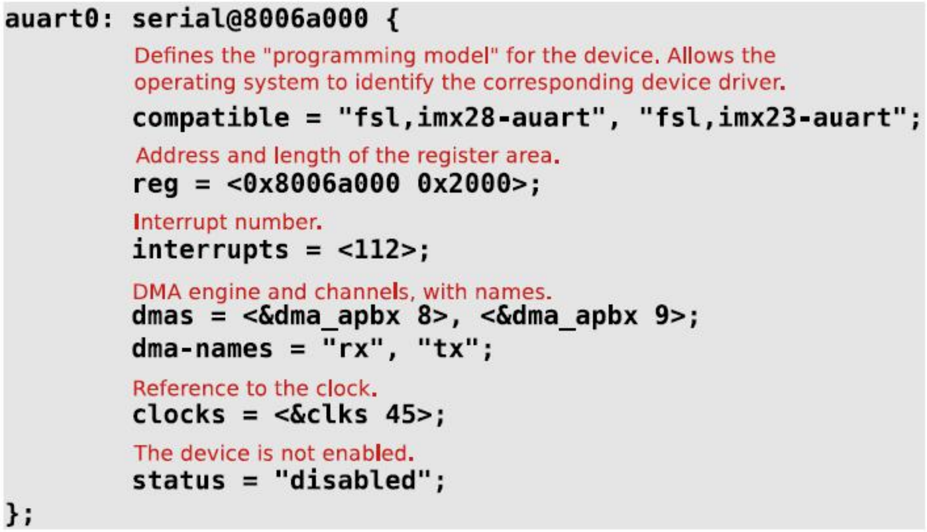
- It's a data structure and ***language*** for describing the hardwar so that the kernel doesn't need to hard code details of the machine.
- Device tree describes device information in a system that cannot be dynamically detected by a client program (i.e USB, PCI,... devices).
- Device Tree Source Files
  - .dts files for board-level defnitions.
  - .dtsi files for included files, generally containing SoC-level defnitions.
- Device Tree Compiler compiles the source into a binary form called "Device Tree Blob" then gets loaded by the bootloader and parsed by the kernel at boot time.
- The bootloader loads two binaries,
  - Kernel image ***uImage*** or ***zImage***.
  - ***DTB*** located in: arch/***arch_specific***/boot/dts, one per board.



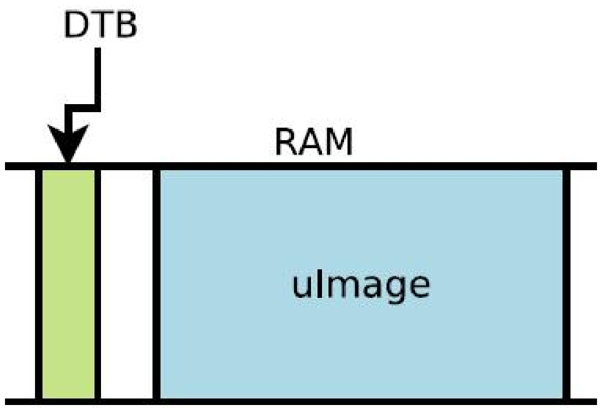
Device Trees, *Cont'd*

- The bootloader passes the DTB address through a HW-Register to the kernel.
- i.e U-Boot "Universal Bootloader" uses command line: ***bootm*** <kernel img addr> - <dtb addr>.
  to boot application image and dtb from certain memory address after loading them.
- The ***DTB*** will be placed in the memory by the bootloaer after the kernel image.
- Device Tree syntax,





Taken from arch/arm/boot/dts/imx28.dtsi

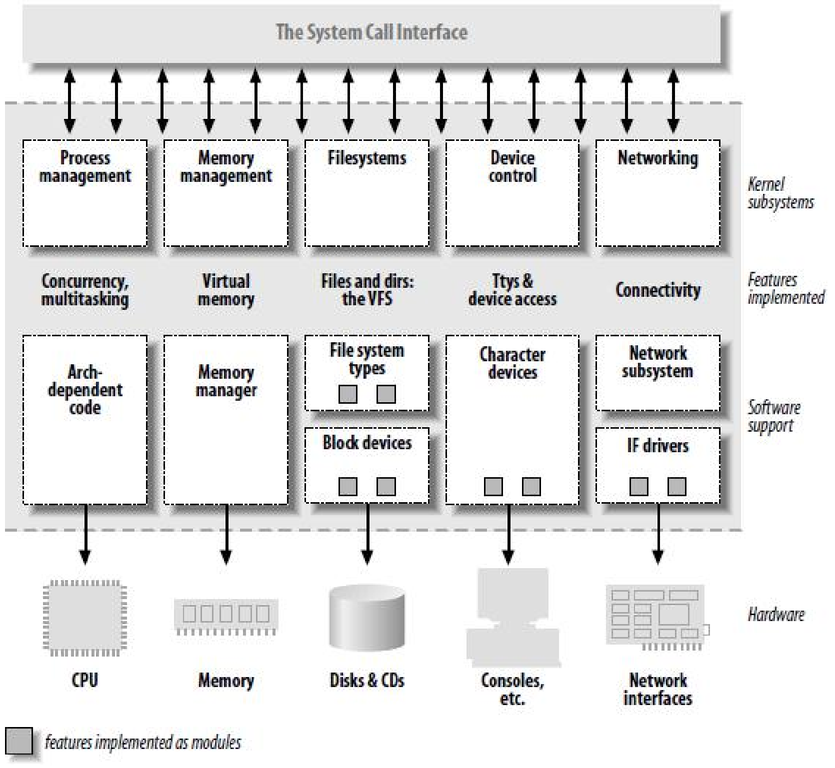


Loadable Modules

- Each piece of code that can be added to the kernel at runtime is called a module.
- Each module is made up of object code **.ko** (not linked into a complete executable) that can be dynamically linked to the running kernel by the ***insmod*** program and can be unlinked by the ***rmmod*** program.
- Kernel supports different classes of modules not limited to device driver only, like tracing, and debugging modules.
- Different classes are shown.
- List the loaded kernel modules,

  $ find /lib/modules/$(uname -r) -type f -name '.ko'*

```
karim_eshapa@karimeshapa-vm:~$ find /lib/modules/$(uname -r) -type f -name '*.ko'
/lib/modules/4.4.0-198-generic/kernel/lib/ts_fsm.ko
/lib/modules/4.4.0-198-generic/kernel/lib/lru_cache.ko
/lib/modules/4.4.0-198-generic/kernel/lib/pm-notifier-error-inject.ko
/lib/modules/4.4.0-198-generic/kernel/lib/notifier-error-inject.ko
/lib/modules/4.4.0-198-generic/kernel/lib/memory-notifier-error-inject.ko
/lib/modules/4.4.0-198-generic/kernel/lib/test_static_key_base.ko
/lib/modules/4.4.0-198-generic/kernel/lib/bch.ko
/lib/modules/4.4.0-198-generic/kernel/lib/842/842_decompress.ko
/lib/modules/4.4.0-198-generic/kernel/lib/842/842_compress.ko
/lib/modules/4.4.0-198-generic/kernel/lib/test_printf.ko
/lib/modules/4.4.0-198-generic/kernel/lib/crc7.ko
/lib/modules/4.4.0-198-generic/kernel/lib/test-kstrtox.ko
/lib/modules/4.4.0-198-generic/kernel/lib/test_static_keys.ko
/lib/modules/4.4.0-198-generic/kernel/lib/test_bpf.ko
```



Hotplug

- Some buses (e.g., **USB**) allow devices to be connected while the system is running without requiring a system reboot, it's discovered dynamically.
- When the system detects a new device, the requisite driver can be automatically added to the kernel by loading the corresponding module.
- There is something still exists in the kernel and violates the open source code environment, what is called "***binary-only modules***".
- **Binary-only modules**, some commercial companies still see providing its modules as binaries to be loaded in the kernel is the best choice for their market and not to open it.

FileSystems & Virtual Filesystems

- Allow stored data to be organized into directory structures and also have the job of linking other meta-information (owners, access rights, etc...).
- Linux supports many filesystems (Ext2 and Ext3, ReiserFS, XFS, FAT, Tmpfs,...).
- The Kernel provides an abstract layer to hide the filesystem details from the application called ***Virtual Filesystem*** "VFS" to (open, write, read, seek,...) from the represented files.
- View file systems disk usage,

  $ df -T

```
karim_eshapa@karimeshapa-vm:~$ df -T
Filesystem     Type      1K-blocks    Used Available Use% Mounted on
udev           devtmpfs   12375064       0  12375064   0% /dev
tmpfs          tmpfs       2479160   17476   2461684   1% /run
/dev/sda1      ext4       12253360 8645880   2962004  75% /
tmpfs          tmpfs      12395788     172  12395616   1% /dev/shm
tmpfs          tmpfs          5120       4      5116   1% /run/lock
tmpfs          tmpfs      12395788       0  12395788   0% /sys/fs/cgroup
tmpfs          tmpfs       2479160      56   2479104   1% /run/user/1000
/dev/sr0       iso9660       56618   56618         0 100% /media/karim_eshapa/VBox_GAs_5.2.18
```

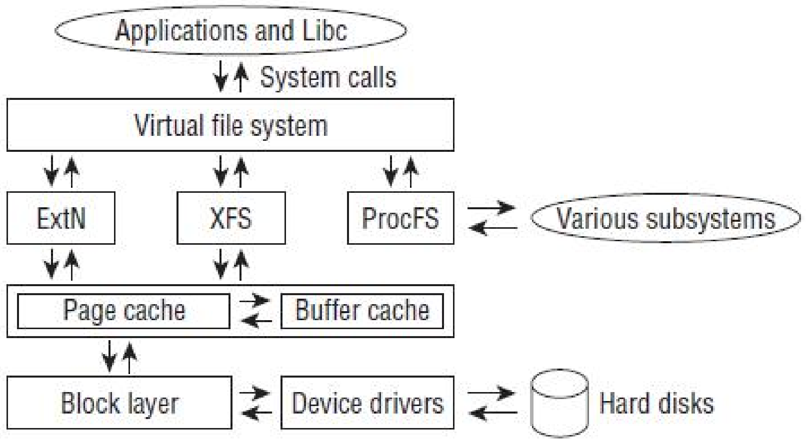
- Please refer to Prof.Ahmed Elarabawy Course for the ***File Handling*** and the different ***Filesystems*** introduction, it's so better than I would ever introduce :)

  Course 102: Lecture 5: File Handling Internals

  Course 102: Lecture 26: FileSystems in Linux (Part 1)

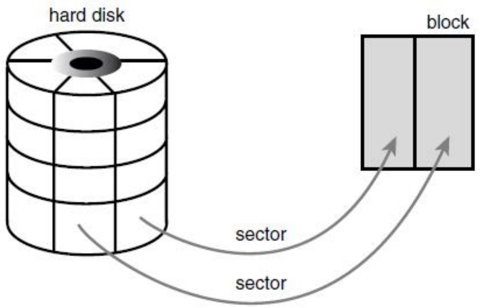
  Course 102: Lecture 27: FileSystems in Linux (Part 2)

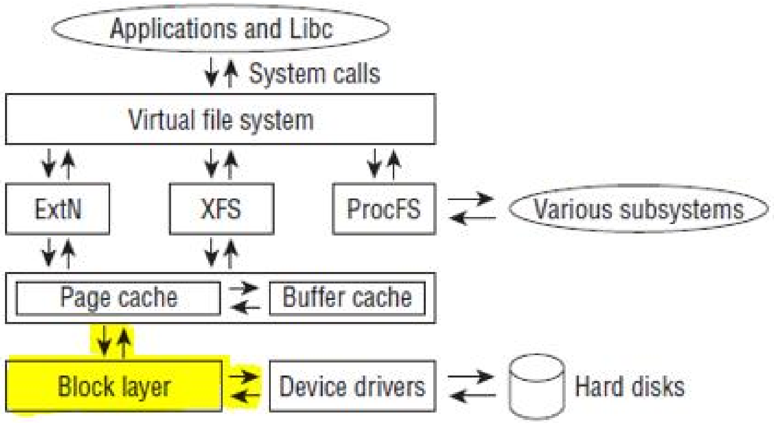
  Course 102: Lecture 28: Virtual FileSystems

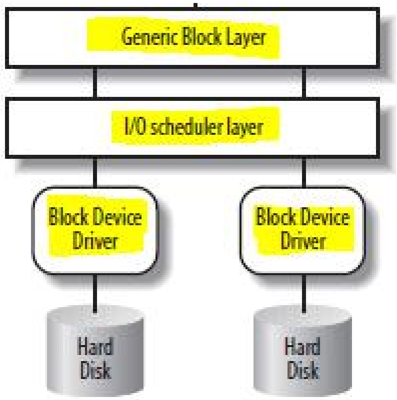


Block Layer

- As we agreed, the block devices i.e Hard Disk needs the filesystem layer (EXT2,3, FAT,..) to be organized into a form of ***partitions*** and then the files into ***blocks*** and provide that to the whole system (Please refer to the last mentioned ***Course 102*** lectures).
- The kernel performs all disk operations in terms of ***blocks***.
- Because the device's smallest addressable unit is the ***sector***, the block size can be no smaller than the ***sector*** and must be a multiple of a sector.
- block sizes are a power-of-two multiple of the ***sector*** size and are not greater than the ***page*** size, the Common ***block*** sizes are 512 bytes, 1 KB, and 4 KB.
- But how the kernel core manages the block devices and their ***block requests*** back and forth, the answer is the ***block I/O layer***.

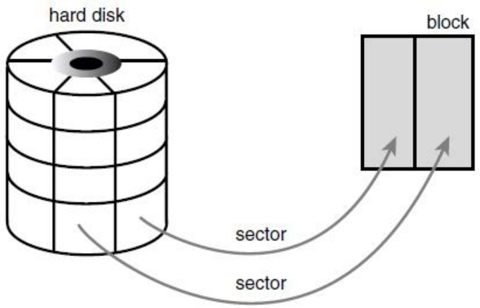


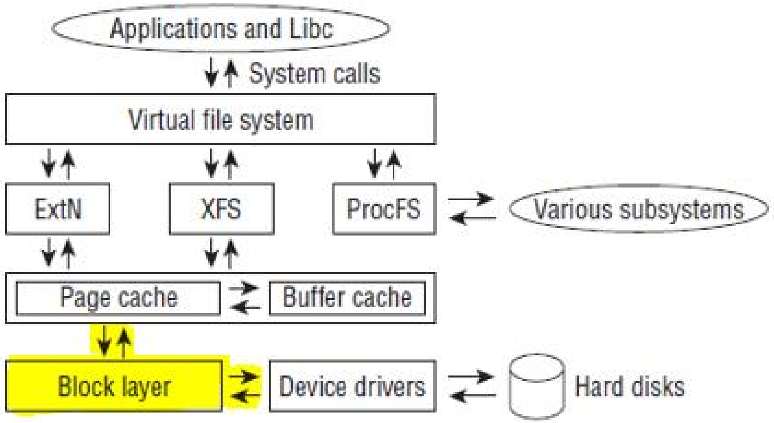


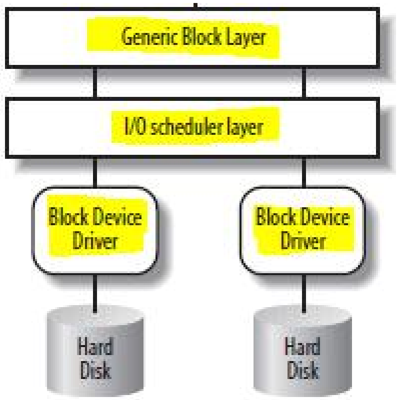


Block Layer

- The ***block I/O layer*** consists of two main layers ***(Generic Block Layer and I/O scheduler Layer).***
- The block device details (i.e Hard Disk) are handled by the ***Block device driver*** component that has to know how to communicate with the device (***Bus Characteristics, Data rate, DMA operations,*** and so on).







Generic Block Layer

- It manages the following,
  - **Buffers and Buffer Heads**:

    When a block is stored in memory it is stored in a ***buffer***, each ***buffer*** is associated with exactly one block. Each ***buffer*** is associated with a descriptor called ***buffer_head*** that holds all the information that the kernel needs to manipulate the buffer (associated page, associated block device, ...).
  - **BIO Structure:**

    It's the basic container for block I/O within the kernel that describes an **ongoing** I/O block device operation.

    It has an identifier for a disk storage area—the initial sector number and the number of sectors included in the storage area—and one or more segments describing the memory areas involved in the I/O operation.
  - **Request Queues:**

    Block devices maintain request queues to store their pending block I/O requests.

    Requests are added to the queue by higher-level code in the kernel, such as ***filesystems***.

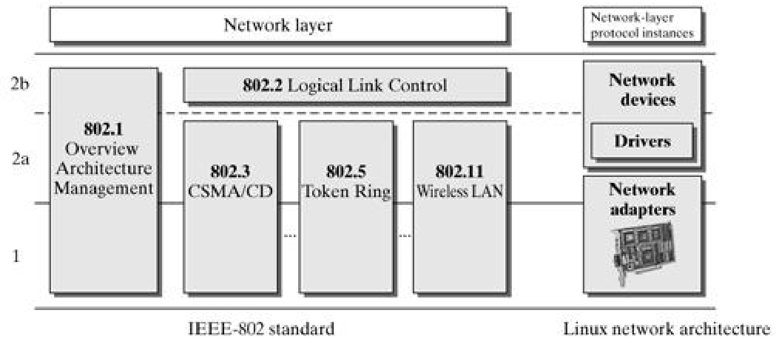
    Each request can be composed of more than one ***bio structure*** because individual requests can operate on multiple consecutive disk blocks.

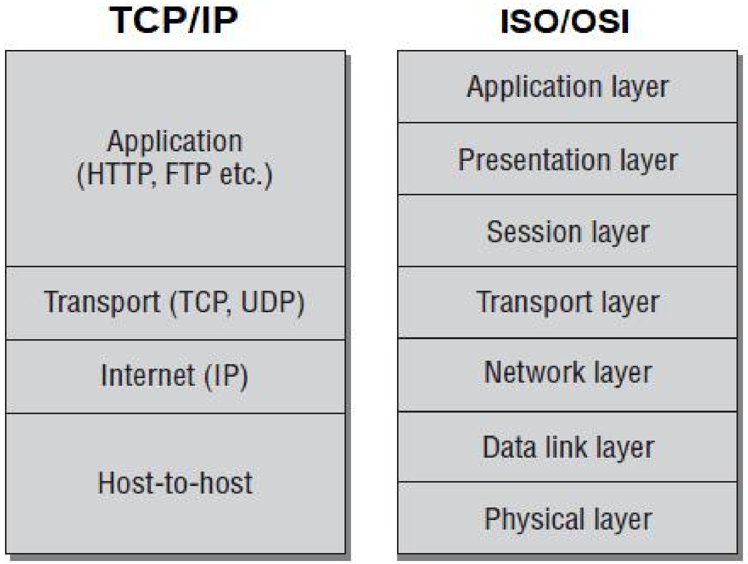
I/O Scheduler Layer

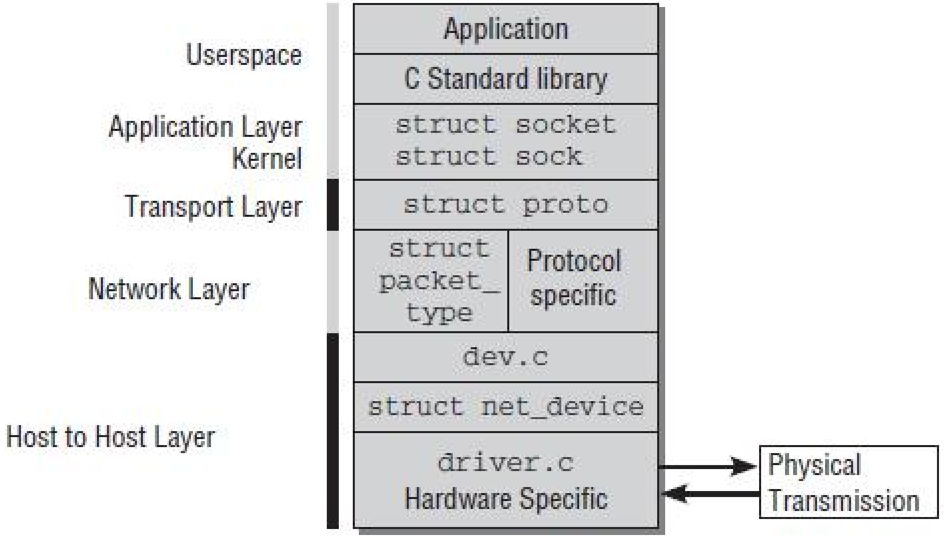
- Sending out requests to the block devices in the order that the kernel issues them, as soon as it issues them, results in poor performance. One of the slowest operations in a modern computer is disk seeks.
- Each seek—positioning the hard disk's head at the location of a specific block—takes many milliseconds so, Minimizing seeks is absolutely crucial to the system's performance.
- The kernel does not issue block I/O requests to the disk in the order they are received or as soon as they are received. Instead, ***I/O scheduler*** performs operations called ***merging*** and ***sorting*** to greatly improve the performance of the system.
- It manages the ***request queue*** with the goal of reducing seeks, which results in greater global throughput.
- There are multiple implementations for the ***I/O Scheduler,*** each one has been created to overcome a certain problem, by default, block devices use the Complete Fair Queuing I/O scheduler.
    - Deadline I/O Scheduler.
    - Anticipatory I/O Scheduler.
    - Complete Fair Queuing I/O Scheduler.
    - Noop I/O Scheduler.

Networks

- Kernel uses the **TCP/IP** stack model and the kernel layer model described as shown.
- kernel implements the protocols up to the ***transport layer***, while application layer protocols are tipically implemented in user space (HTTP, FTP, SSH, etc.).
- **Host-to-Host** Layer,
  - The ***Network Adapter Card*** implements (Phy + MAC layers according to 802.x standard).
  - The kernel core manages the bus comm with the ***Network Adapter Card*** as a “PCI node”.
  - The device driver is responsible for the ***LLC*** layer that provides a uniform interface for the upper layer.
- LLC: Logical Link Control.

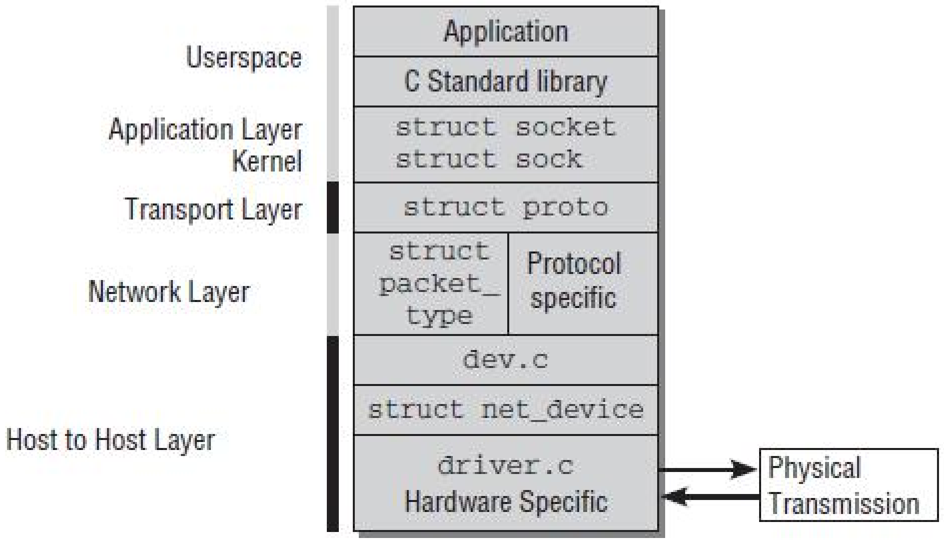






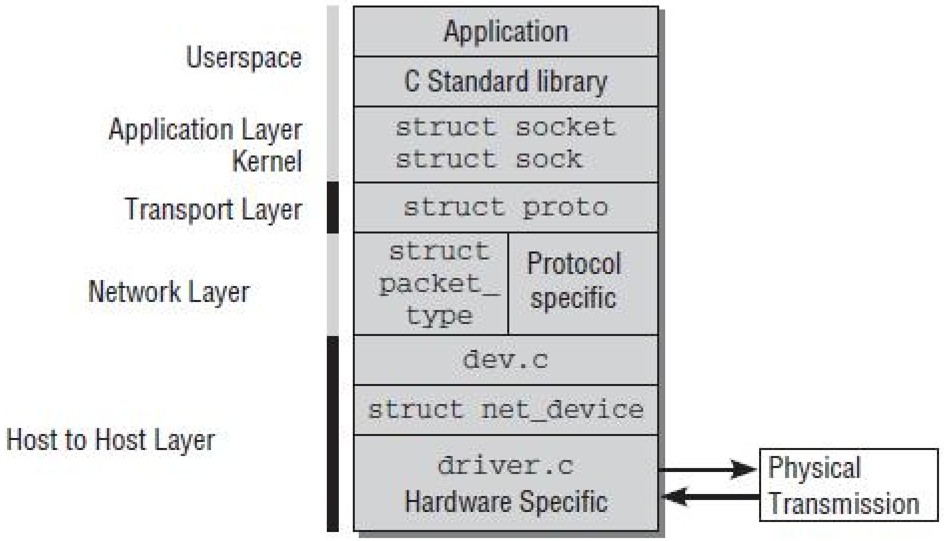
Networks, *Cont'd*

- The ***socket*** abstracts a communication channel and considered as the kernel-based TCP/IP stack interaction interface.
- User space sees common function to use socket: creation (***socket***), initialization (bind), connecting (***connect***), waiting for a connection (***listen***, ***accept***), closing a socket (***close***).
- ***sock*** describes an INET socket, used to store information about the status of a connection.
- ***proto*** is considered the transport layer interface, contains set of different protocol management functions composed according to the protocol type of the socket (parameter three protocol: IPPROTO_TCP, IPPROTO_UDP, IPPROTO_ICMP internet control message).



Networks, *Cont'd*

- ***packet_type*** decribes a ***protocol handler*** to the networking stack, i.e if we have two low-level (IP layer) protocols that are of interest to messages, one is ***ARP*** and the other is ***IP***, so we need to distinguish between the ***IP*** message or an ***ARP*** message.
- ***sk_buff*** (socket buffer) describes a network packet, the header, packet contents, the protocols used, the ***net_device*** used,...
- ***dev.c*** provides protocol independent device support routines.
- ***net_device*** is a part of kernel device driver model structures.
- ***driver.c*** provides some ***device_driver structure*** management.



Kernel Source Tree

- Please download the kernel source, “***next-tree***” for the next development running,

  $ git clone https://kernel.googlesource.com/pub/scm/linux/kernel/git/next/linux-next

Directory	Description
arch	Architecture-specific source
block	Block I/O layer
crypto	Crypto API
Documentation	Kernel source documentation
drivers	Device drivers
firmware	Device firmware needed to use certain drivers
fs	The VFS and the individual filesystems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
samples	Sample, demonstrative code
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)
tools	Tools helpful for developing Linux
virt	Virtualization infrastructure