Introduction to Linux Kernel Architecture

On a purely technical level, the kernel is an intermediary layer between the hardware and the software. Its purpose is to pass application requests to the hardware and to act as a low-level driver to address the devices and components of the system. Nevertheless, there are other interesting ways of viewing the kernel.

- The kernel can be regarded as an enhanced machine that, in the view of the application,abstracts the computer on a high level. For example, when the kernel addresses a hard disk, it must decide which path to use to copy data from disk to memory, where the data reside, which commands must be sent to the disk via which path, and so on.

- Applications, on the other hand, need only issue the command that data are to be transferred. How this is done is irrelevant to the application — the details are abstracted by the kernel. Application programs have no contact with the hardware itself, only with the kernel, which, for them, represents the lowest level in the hierarchy they know — and is therefore an enhanced machine.

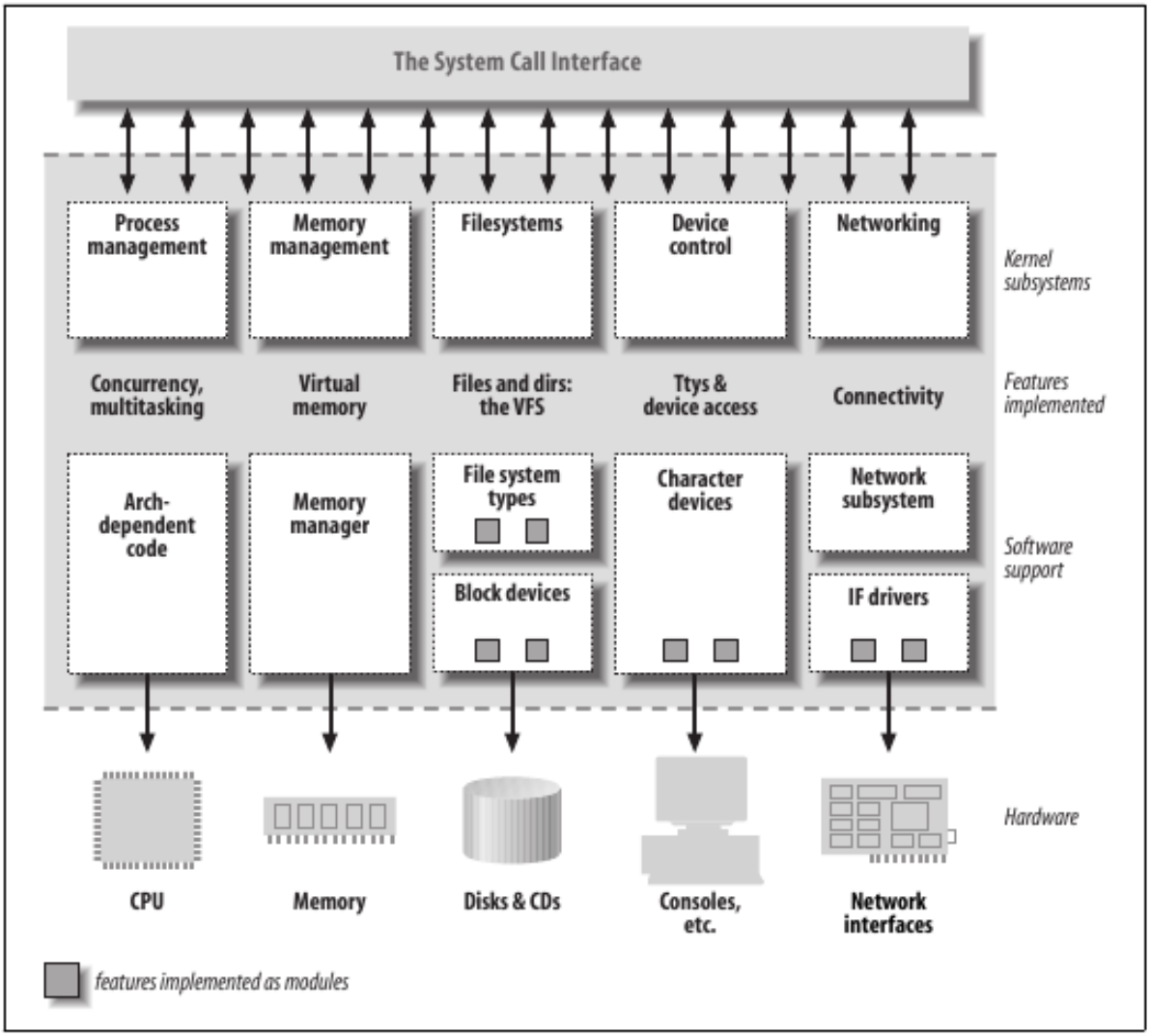
- Viewing the kernel as a resource manager is justified when several programs are run concurrently on a system. In this case, the kernel is an instance that shares available resources — CPU time, disk space, network connections, and so on — between the various system processes while at the same time ensuring system integrity.

- Linux kernel is implemented as a monolithic kernel ; the other type of implementation is micro-kernel – we will not be interested in this type of architecture

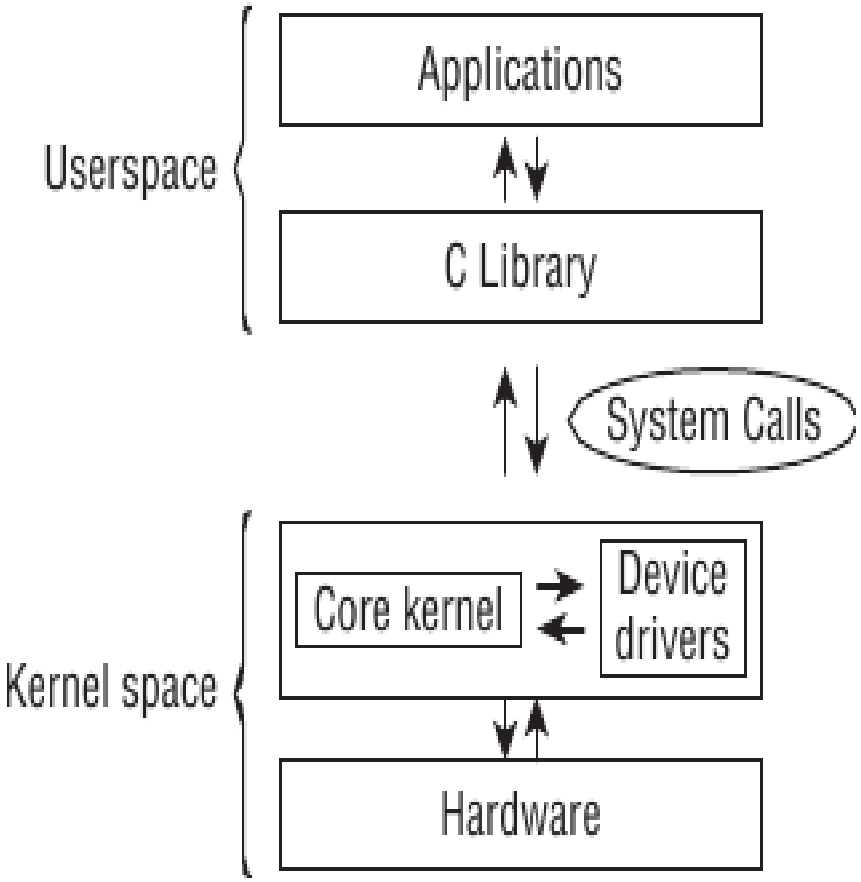
- Look at following diagram for clarity

----------intentionally left blank----------

Monolithic kernel architecture



user-space perspective of a Monolithic kernel



Linux Processes

- Applications, servers, and other programs running under Unix are traditionally referred to as processes.

- Each process is assigned address space in the virtual memory of the CPU. The address spaces of the individual processes are totally independent so that the processes are unaware of each other — as far as each process is concerned, it has the impression of being the only process in the system.

- If processes want to communicate to exchange data, for example, then special kernel mechanisms must be used.

- Because Linux is a multitasking system, it supports what appears to be concurrent execution of several processes. Since only as many processes as there are CPUs in the system can really run at the same time, the kernel switches (unnoticed by users) between the processes at short intervals to give them the impression of simultaneous processing.

Key points

- The kernel, with the help of the CPU, is responsible for the technical details of task switching. Each individual process must be given the illusion that the CPU is always available. This is achieved by saving all state-dependent elements of the process before CPU resources are withdrawn and the process is placed in an idle state. When the process is reactivated, the exact saved state is restored. Switching between processes is known as task switching.

- The kernel must also decide how CPU time is shared between the existing processes. Important processes are given a larger share of CPU time, less important processes a smaller share. The decision as to which process runs for how long is known as scheduling.

- Linux supports preemptive scheduling and a preemptive kernel

- Linux employs a hierarchical scheme in which each process depends on a parent process. The kernel starts the init program as the first process that is responsible for further system initialization actions and display of the login prompt.

- init process is therefore the root from which all processes originate; this may be observed using ps command or pstree command (to be dicussed during the upcoming practical sessions)

How this hierarchical structure spreads is closely connected with how new processes are generated. For this purpose, Linux uses two mechanisms called fork and exec.

- fork :

  Generates an exact copy of the current process that differs from the parent process only in its PID (process identification). After the system call has been executed, there are two processes in the system, both performing the same actions.

  The memory contents of the initial process are duplicated — at least in the view of the program. Linux uses a well-known technique known as copy on write that allows it to make the operation much more efficient by deferring the copy operations until either parent or child writes to a page — read-only accessed can be satisfied from the same page for both.

- exec:

  Loads a new program into an existing content and then executes it. The memory pages reserved by the old program are flushed, and their contents are replaced with new program contents data. The new program then starts executing.

Threads in Linux

- Processes are not the only form of (program) execution supported by the kernel. In addition to (heavy-weight) processes — another name for classical Unix/Linux processes — there are also threads, sometimes referred to as light-weight processes.

- They have also been around for some time, and essentially, a process may consist of several threads that all share the same data and resources but take different paths through the program code.

- Because the threads and the main program share the same address space, data received automatically reside in the main program. There is therefore no need for any communication effort whatsoever, except to prevent the threads from stepping onto their feet mutually by accessing identical memory locations, for instance.

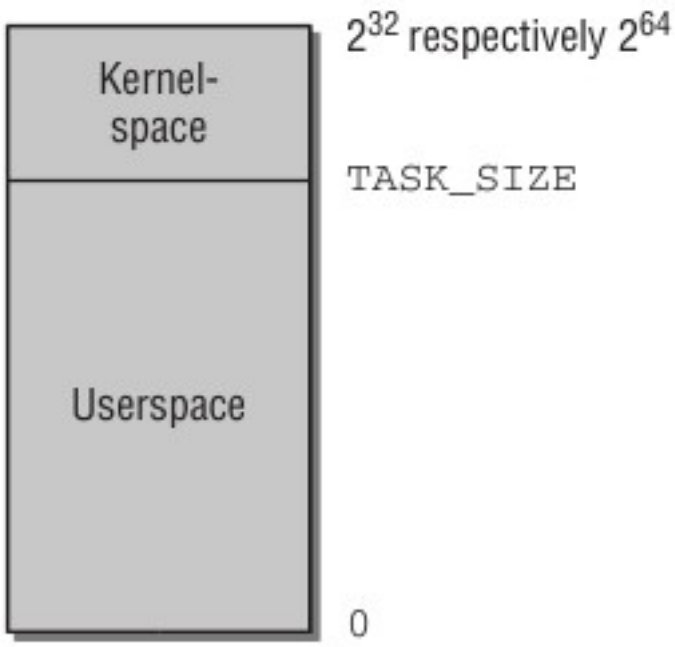
- Linux provides the clone() system call to generate threads. This works in a similar way to fork but enables a precise check to be made of which resources are shared with the parent process and which are generated independently for the thread. This fine-grained distribution of resources extends the classical thread concept and allows for a more or less continuous transition between thread and processes – this is a non-standard, Linux-specific system call

virtual address spaces

- because memory areas are addressed by means of pointers, the word length of the CPU determines the maximum size of the address space that can be managed. On 32-bit systems such as IA-32, PPC, and m68k, these are 4 GiB, whereas on more modern 64-bit processors such as Alpha, Sparc64, IA-64,and AMD64, very large address-space can be managed.

- The maximal size of the address space is not related to how much physical RAM is actually available, and therefore it is known as the virtual address space. One more reason for this terminology is that every process in the system has the impression that it would solely live in this address space, and other processes are not present from their point of view.

- Linux divides virtual address space into two parts known as kernel-space and user-space as illustrated in the diagram below:



user-space vs kernel-space

- every user process in the system has its own virtual address range that extends from 0 to TASK_SIZE.

- The area above (from TASK_SIZE to 232 or 264) is reserved exclusively for the kernel — and may not be accessed by user processes. TASK_SIZE is an architecture-specific constant that divides the address space in a given ratio — in IA-32 systems, for instance, the address space is divided at 3 GiB so that the virtual address space for each process is 3 GiB; 1 GiB is available to the kernel because the total size of the virtual address space is 4 GiB.

- Although actual figures differ according to architecture, the general concepts do not.

- This division does not depend on how much RAM is available. As a result of address space virtualization,each user process thinks it has 3 GiB of memory.

- The kernel space at the top end of the virtual address space is always the same, regardless of the process currently executing.

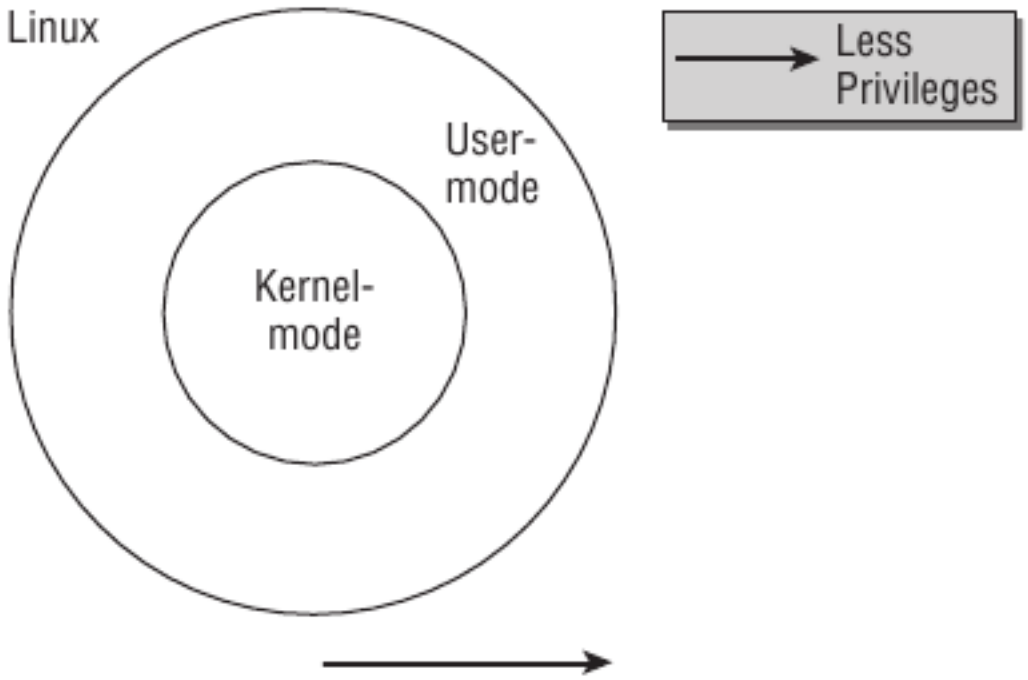
- The picture is more complex for a 64-bit system. Explore and find this – a good exercise !!!

Privilige levels

- Linux uses only two different modes - kernel mode and user mode. The key difference between the two is that access to the memory area above TASK_SIZE — that is, kernel space — is forbidden in user mode.

- User processes are not able to manipulate or read the data in kernel space. Neither can they execute code stored there. This is the sole domain of the kernel. This mechanism prevents processes from interfering with the core of the system – the key data-structures that manage the system-space, processes, threads, files, memory and so on.

Illustration of priviliged and less-priviliged modes



Executing kernel-space code/core-code

- the switch from user to kernel mode is made by means of special transitions known as system calls; these are executed differently depending on the system. If a normal process wants to carry out any kind of action affecting the entire system (e.g., manipulating I/O devices), it can do this only by issuing a request to the kernel with the help of a system call. The kernel checks whether the process is permitted to perform the desired action and then performs the action on its behalf. return is then made to user mode, using a special machine instruction.

- besides executing code on behalf of a user program, the kernel can also be activated by asynchronous hardware interrupts, and is then said to run in interrupt context. The main difference to running in process context is that the user-space portion of the virtual address space must not be accessed. because interrupts occur at random times, a random user-land process is active when an interrupt occurs, and since the interrupt will most likely be unconnected with the cause of the interrupt, the kernel has no business with the contents of the current user-space. When operating in interrupt context, the kernel must be more cautious than normal; for instance, it must not go to sleep.

- besides normal processes, there can also be kernel threads running on the system. Kernel threads are also not associated with any particular user-space process, so they also have no business dealing with the user portion of the address space. In many other respects, kernel threads behave much more like regular user-land applications, though: In contrast to a kernel operating in interrupt context, they may go to sleep, and they are also tracked by the scheduler like every regular process in the system. The kernel uses them for various purposes that range from data synchronization of RAM and block devices to helping the scheduler distribute processes among CPUs, and we will frequently encounter them in upcoming discussions

Illustration of system-call / interrupt transitions

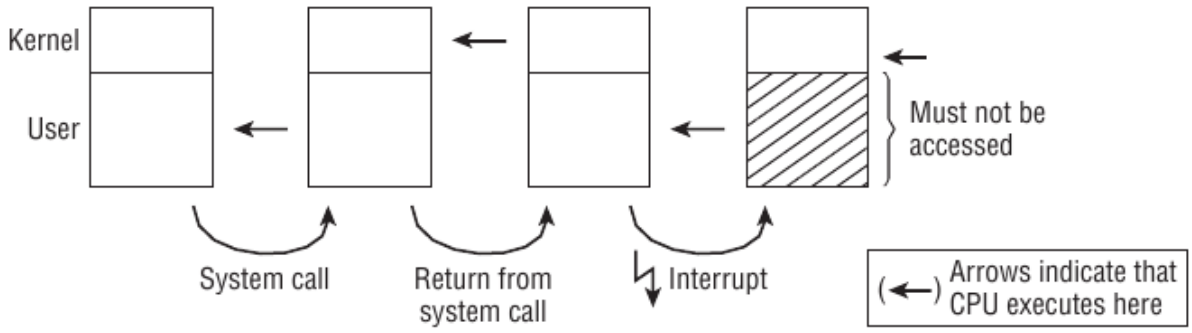
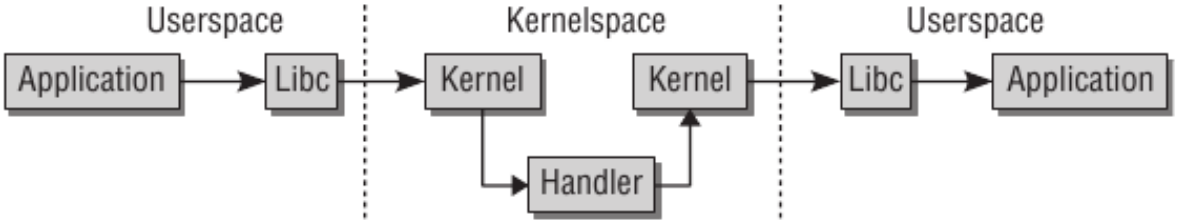


Illustration of control-flow during a system call



virtual address to physical address mapping

- in most cases, a single virtual address space is bigger than the physical RAM available to the system. and the situation does not improve when each process has its own virtual address space.

- the kernel and CPU must therefore consider how the physical memory actually available can be mapped onto virtual address areas.

- the preferred method is to use page tables to map virtual addresses to physical addresses. whereas virtual addresses relate to the combined user and kernel space of a process, physical addresses are used to address the RAM actually available.

- the virtual address spaces of both processes shown in the diagram below are divided into portions of equal size by the kernel. These portions are known as pages. Physical memory is also divided into pages of the same size.

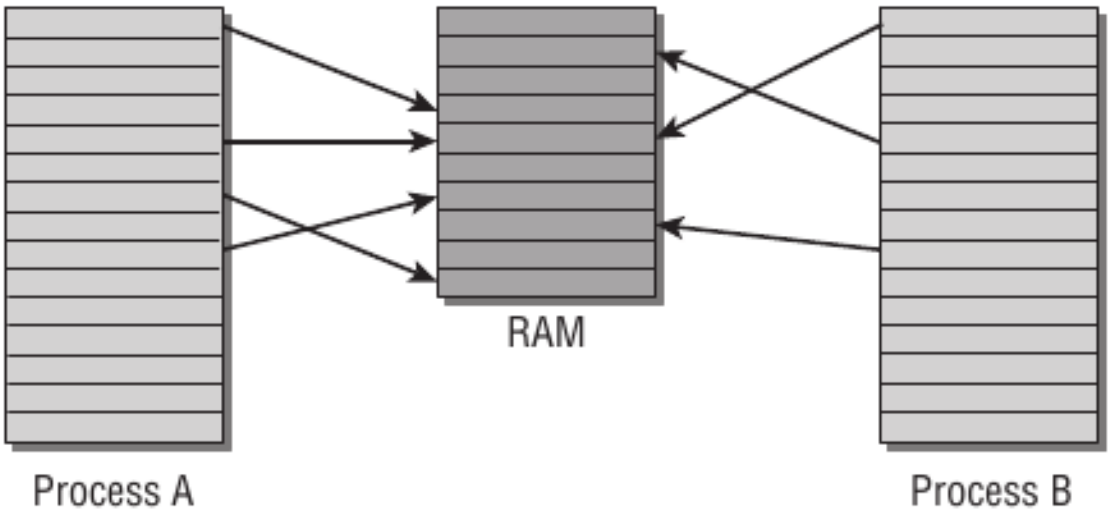
- physical pages are often called page frames. In contrast, the term page is reserved for pages in virtual address space.

- mapping between virtual address spaces and physical memory also enables the otherwise strict separation between processes to be lifted. our example includes a page frame explicitly shared by both processes. page 5 of A and page 1 of B both point to the physical page frame 5. this is possible because entries in both virtual address spaces (albeit at different positions) point to the same page.

- since the kernel is responsible for mapping virtual address space to physical address space, it is able to decide which memory areas are to be shared between processes and which are not.

- the diagram also shows that not all pages of the virtual address spaces are linked with a page frame. this may be because either the pages are not used or because data have not been loaded into memory because they are not yet needed. it may also be that the page has been paged-out onto hard disk and will be paged-back in main-memory when needed.

virtual to physical address mapping(pages to page-frames)



Page tables

- data structures known as page tables are used to map virtual address space to physical address space.

- easiest way of implementing the association between both would be to use an array containing an entry for each page in virtual address space. This entry would point to the associated page frame. but there is a problem. IA-32 architecture uses, for example, 4 KiB pages — given a virtual address space of 4 GiB, this would produce an array with a million entries.

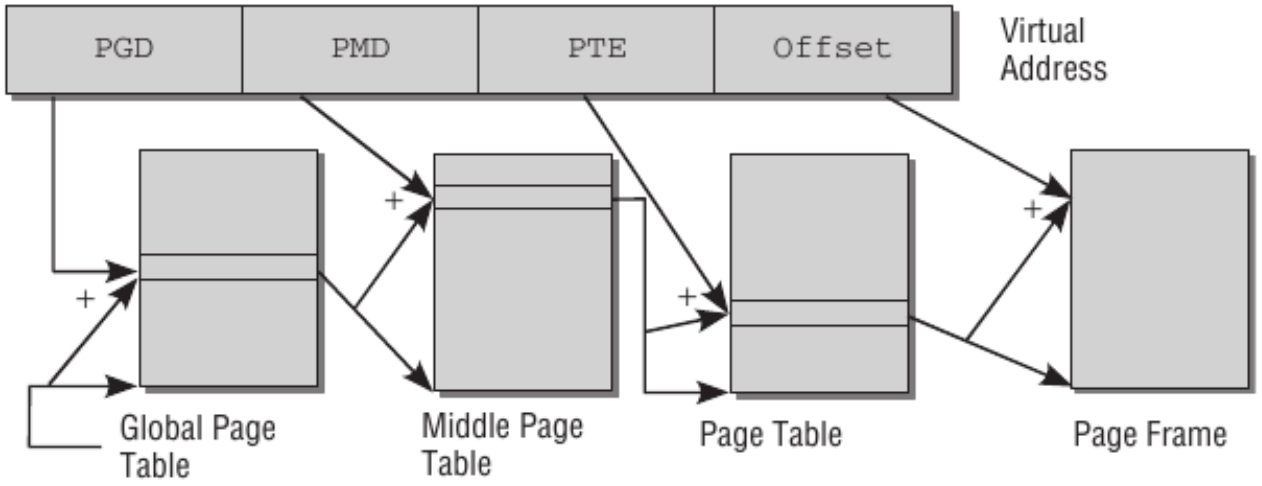
- on 64-bit architectures, the situation is much worse.

- because each process needs its own page tables, this approach is impractical because the entire RAM of the system would be needed to hold the page tables.

- as most areas of virtual address spaces are not used and are therefore not associated with page frames, a far less memory-intensive model that serves the same purpose can be used: multilevel paging.

- MMU does the run-time translation and TLB minimizes translations using caching

Illustration of a multi-level page-table



Memory mappings

- memory mappings are an important means of abstraction. They are used at many points in the kernel and are also available to user applications. Mapping is the method by which data from an arbitrary source are transferred into the virtual address space of a process.

- the address space areas in which mapping takes place can be processed using normal methods in the same way as regular memory. However, any changes made are transferred automatically to the original data source.

- for example, the contents of a file can be mapped into memory. a process then need only read the contents of memory to access the contents of the file, or write changes to memory in order to modify the contents of the file. The kernel automatically ensure that any changes made are implemented in the file.

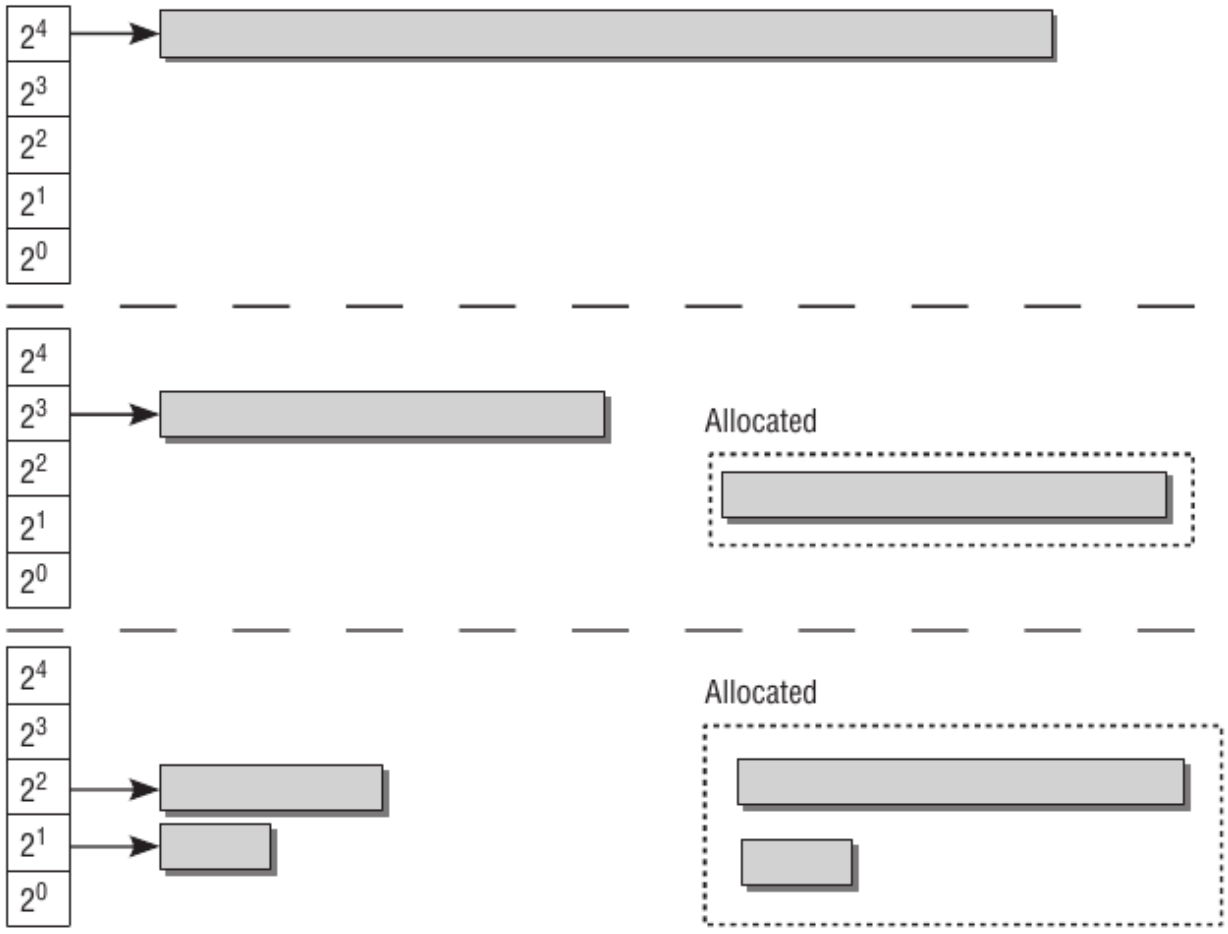
Physical memory management and KMA

- when it allocates RAM, the kernel must keep track of which pages have already been allocated and which are still free in order to prevent two processes from using the same areas in RAM.

- because memory allocation and release are very frequent tasks, the kernel must also ensure that they are completed as quickly as possible. The kernel can allocate only whole page frames. Dividing memory into smaller portions is delegated to the standard library in user-space.
this library splits the page frames received from the kernel into smaller areas and allocates memory to the processes.

- in addition to the above, the kernel must also allocate memory for the various sub-systems in the system-space – for all these requests, kernel memory allocator or physical memory manager is responsible – it is further divided into sub sub - systems

buddy allocator



slab cache / slab allocator

- often the kernel itself needs memory blocks much smaller than a whole page frame. Because it cannot use the functions of the standard library, it must define its own, additional layer of memory management that builds on the buddy system and divides the pages supplied by the buddy system into smaller portions.

- the method used not only performs allocation but also implements a generic cache for frequently used small objects; this cache is known as a slab cache. It can be used to allocate memory in two ways:

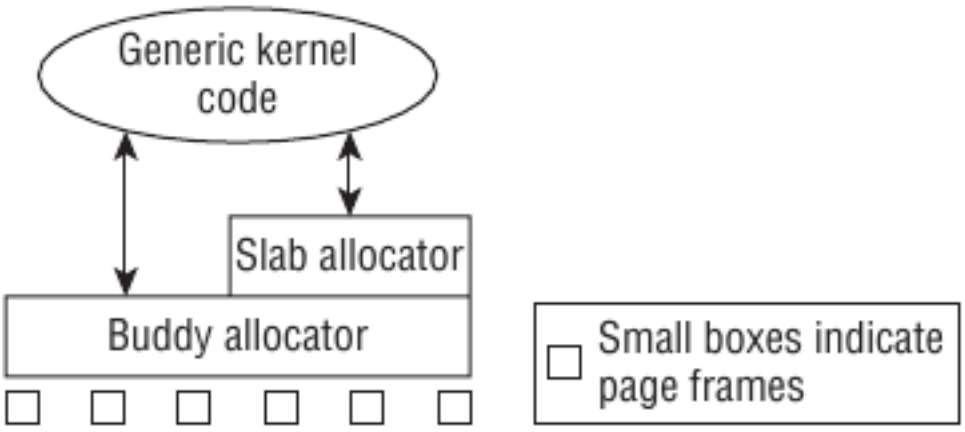
  - for frequently used objects, the kernel defines its own cache that contains only instances of the desired type. Each time one of the objects is required, it can be quickly removed from the cache (and returned there after use); the slab cache automatically takes care of interaction with the buddy system and requests new page frames when the existing caches are full.

  - for the general allocation of smaller memory blocks, the kernel defines a set of slab caches for various object sizes that it can access using the same functions with which we are familiar from user-space programming; a prefixed k indicates that these functions are associated with the kernel: kmalloc and kfree.

  - in addition, you can create your own custom-caches as well

Note : we will be practically touching upon the above facilities during system call writing and adding procfs entries ; you may also find extensive use of them in device-driver writing

Illustration of slab-allocator architecture



Time management

- the kernel must be capable of measuring time and time differences at various points — when scheduling processes, for example.

- jiffies are one possible time base. a global variable named jiffies_64 and its 32-bit counterpart jiffies are incremented periodically at constant time intervals.

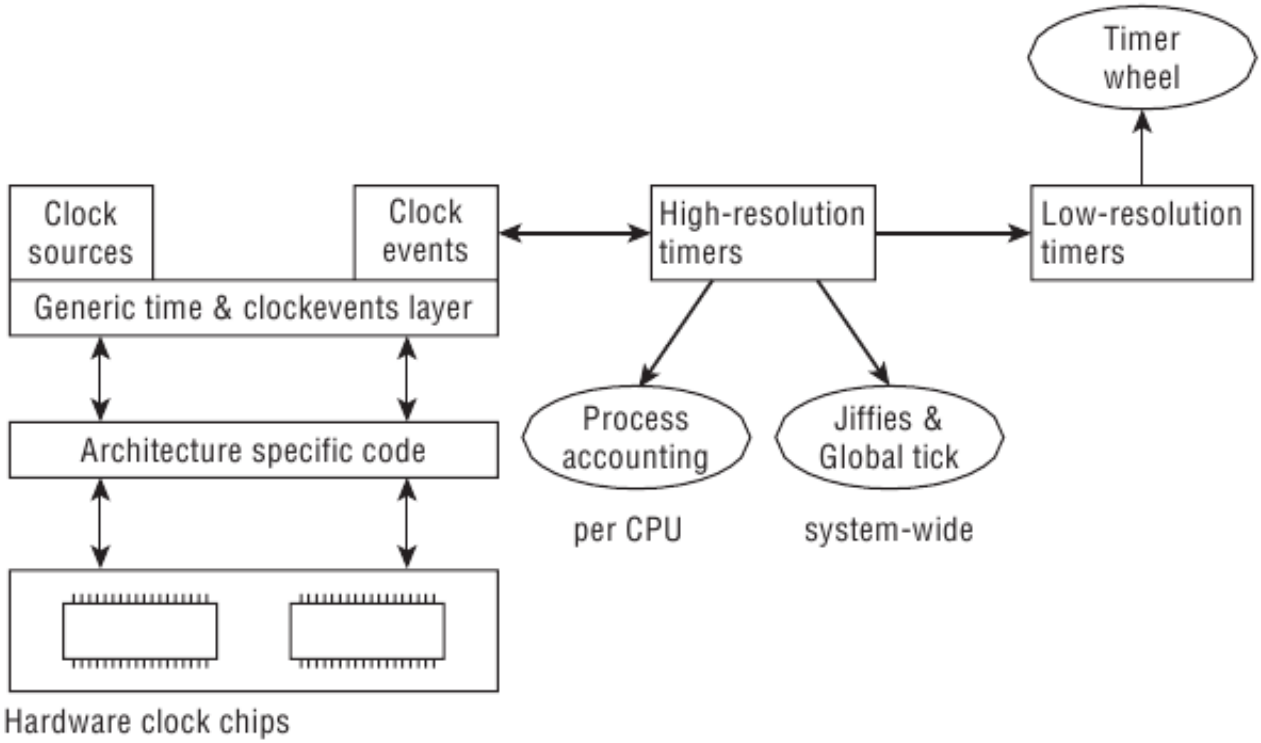
- the various timer mechanisms of the underlying architectures are used to perform these updates — each computer architecture provides some means of executing periodic actions, usually in the form of timer interrupts.

- depending on architecture, jiffies is incremented with a frequency determined by the central constant HZ of the kernel. This is usually on the range between 1,000 and 100; in other words, the value of jiffies is incremented between 1,000 and 100 times per second.

- timing based on jiffies is relatively coarse-grained because 1,000 Hz is not an excessively large frequency nowadays. with high-resolution timers, the kernel provides additional means that allows for keeping time in the regime of nanosecond precision and resolution, depending on the capabilities of the underlying hardware.

- it is possible to make the periodic tick dynamic. When there is little to do and no need for frequent periodic actions, it does not make sense to periodically generate timer interrupts that prevent the processor from powering down into deep sleep states. This is helpful in systems where power is scarce, for instance, laptops and embedded systems.

Illustration of the timer/clock management

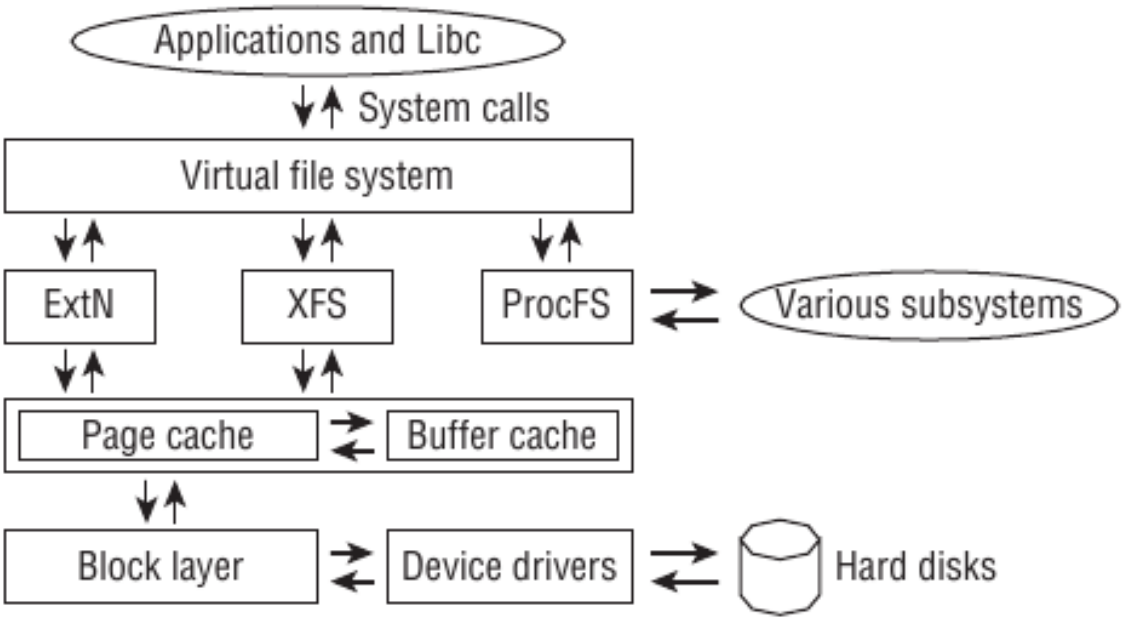


File systems

- the kernel must provide an additional software layer to abstract the special features of the various low-level file-systems from the application layer (and also from the kernel itself).

- this layer is referred to as the VFS (virtual filesystem or virtual filesystem switch).
  it acts as an interface downward (this interface must be implemented by all filesystems) and upward (for system calls via which user processes are ultimately able to access filesystem functions). This is illustrated in diagram below

Illustration of VFS layer and different file-systems in the kernel



Modules and hot-plugging

- modules are used to dynamically add functionality to the kernel at run time — device drivers, filesystems, network protocols, practically any subsystem of the kernel can be modularized.

- this removes one of the significant disadvantages of monolithic kernels as compared with microkernel variants.

- modules can also be unloaded from the kernel at run time, a useful aspect when developing new kernel components.

- modules are an essential requisite to support for hotplugging. Some buses (e.g., USB and FireWire) allow devices to be connected while the system is running without requiring a system reboot. when the system detects a new device, the requisite driver can be automatically added to the kernel by loading the corresponding module.

- modules also enable kernels to be built to support all kinds of devices that the kernel can address without unnecessarily bloating kernel size. Once attached hardware has been detected, only the requisite modules are loaded, and the kernel remains free of unnecessary drivers.

- Linux core is known as modular, monolithic kernel

Software caching

- the kernel uses caches to improve system performance.

- data read from slow block devices are held in RAM for a while, even if they are no longer needed at the time. When an application next accesses the data, they can be read from fast RAM, thus bypassing the slow block device.

- because the kernel implements access to block devices by means of page memory mappings, caches are also organized into pages, that is, whole pages are cached, thus giving rise to the name page cache.

- buffer cache is used to cache data that are not organized into pages

- refer back to the VFS diagram above

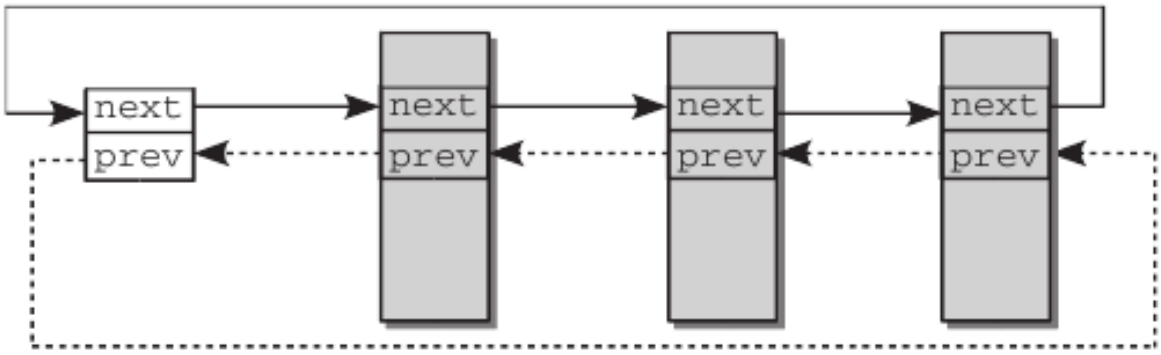
List handling

```
<linux/list.h>
   struct list_head {
            struct list_head *next, *prev;
   };
```

This element could be placed in a data structure as follows:

```
   struct task_struct {
   ...
      struct list_head run_list;
   ...
   };
```

Illustration of list usage



- very popular in the Linux kernel space data-structures

- useful when we look into the data-structures

Typedefs in kernel space data

- the kernel uses typedef to define various data types in order to make itself independent of architecture-specific features because of the different bit lengths for standard data types on individual processors.

- the definitions have names such as sector_t (to specify a sector number on a block device), pid_t (to indicate a process identifier), and so on, and are defined by the kernel in architecture-specific code in such a way as to ensure that they represent the applicable value range.

- many times, system provides helper functions to work with the typedefs

per-CPU variables / localization to avoid extensive locking

- a particularity that does not occur in normal userspace programming is per-CPU variables. They are declared with DEFINE_PER_CPU(name, type), where name is the variable name and type is the data type (e.g., int[3], struct hash, etc.).

- on single-processor systems, this is not different from regular variable declaration.

- on SMP systems with several CPUs, an instance of the variable is created for each CPU.
the instance for a particular CPU is selected with get_cpu(name, cpu), where smp_processor_id(), which returns the identifier of the active processor, is usually used as the argument for cpu.

- employing per-CPU variables has the advantage that the data required are more likely to be present in the cache of a processor and can therefore be accessed faster. This concept also skirts round several communication problems that would arise when using variables that can be accessed by all CPUs of a multiprocessor system.

Size , scope and growth ....

- although a wide range of topics are covered in our discussions, they inevitably just represent a portion of what Linux is capable of:

- it is simply impossible to discuss all aspects of the kernel in detail. Still, what we will discuss will give enough foundation to move forward

- kernel development is a highly dynamic process, and the speed at which the kernel acquires new features and continues to improve is sometimes nothing short of miraculous. as a study by the Linux foundation has shown , roughly 10,000 patches go into each kernel release, and this massive

amount of code is created by nearly 1000 developers per release

- Unix is simple and coherent, but it takes a genius (or at any rate a programmer) to understand and appreciate the simplicity.

— Dennis Ritchie

- that was about Unix and a very longtime ago; Linux is a much more complex and bigger beast, now

---