Q2 (a) What are the characteristics of LINUX?

Answer

1. Multitasking
2. Multi-user access
3. Multi-processing
4. Architecture independence
5. Demand load executables
6. Paging
7. Dynamic Cache for hard disk
8. Shared Libraries
9. Support for POSIX 1003.1
10. Various formats for executable files
11. Memory protected mode
12. Support for national keyboards and fonts
13. Different files systems
14. TCP/IP, SLIP and PPP support

Q3 (a) What is the main advantage and drawback of using micro kernel architecture?

Answer

Microkernel provides minimum functionality of IPC and memory management and can be implemented in a small form. Building on this microkernel, the remaining functions of the OS are relocated to autonomous processes, communicating with the microkernel via a well defined interface.

The main advantage of these structures is a system structure that is clearly less trouble to maintain. Individual components work independently of each other, cannot affect each other unintentionally, and are easy to replace. The development of new components is thus simplified.

The drawback to these architectures: The microkernel architectures force defined interfaces to be maintained between individual components and prevent sophisticated optimizations. In addition, in today's hardware architectures the IPC required inside the microkernel is more extensive than simple function calls. This makes the system slower than traditional monolithic kernels. This slight speed disadvantage is readily accepted since current hardware is generally fast enough and because the advantage of simpler system maintenance reduces development costs.

Q3 (b) Can the process be reactivated once it is interrupted?

Answer

The system call *pause* interrupts the execution of the program until the process is reactivated by a signal. This merely amounts to setting the status of the current process to TASK_INTERRUPTIBLE and then calling the scheduler. This results in another task becoming active.

The process can only be reactivated if the status of the process is returned to TASK_RUNNING. This occurs when a signal is received. The system call pause then returns with the fault ERESTARTNOHAND and carries out the necessary actions for handling of the signal.

```
asmlinkage int sys_pause(void)
{
        current->state = TASK_INTERRUPTIBLE;
        schedule( );
        return – ERESTARTNOHAND;
 }
```

Q4 (a) Describe in detail how the virtual Address Space for a Linux Process is used.

Answer

The virtual address space of any Linux process is divided into two subspaces: kernel space and user space. User space occupies the lower portion of the address space, starting from address 0 and extending up to the platform-specific task size limit (TASK_SIZE in file include/asm/processor.h).

The remainder is occupied by kernel space. Most platforms use a task size limit that is large enough so that at least half of the available address space is occupied by the user address space.

User space is private to the process, meaning that it is mapped by the process's own page table. In contrast, kernel space is shared across all processes.

There are two ways to think about kernel space: We can either think of it as being mapped into the top part of each process, or we can think of it as a single space that occupies the top part of the CPU's virtual address space. Interestingly, depending on the specifics of CPU on which Linux is running, kernel space can be implemented in one or the other way.

During execution at the user level, only user space is accessible. Attempting to read, write, or execute kernel space would cause a protection violation fault. This prevents a faulty or malicious user process from corrupting the kernel. In contrast, during execution in the kernel, both user and kernel spaces are accessible.

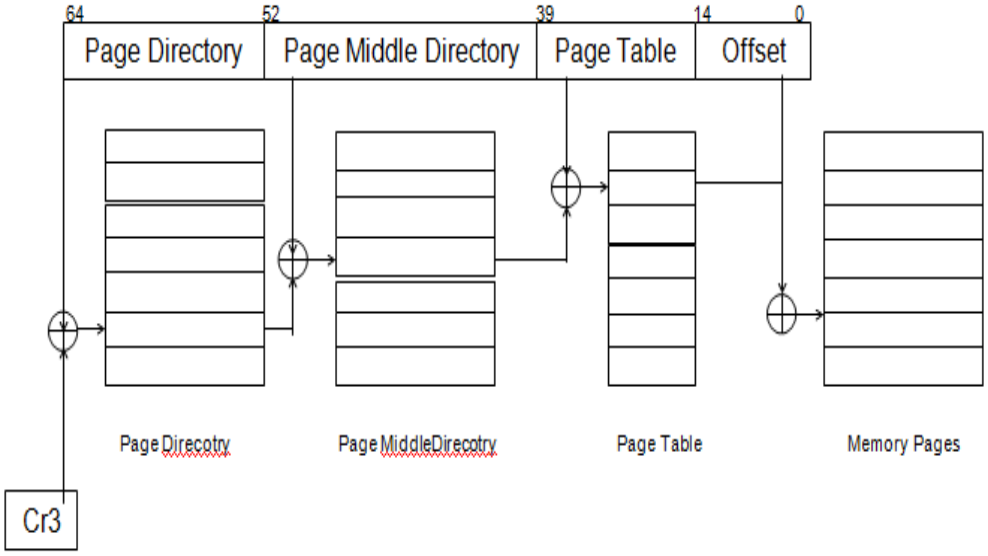
Before continuing our discussion, we need to say a few words about the page size used by the Linux kernel. Because different platforms have different constraints on what page sizes they can support, Linux never assumes a particular page size and instead uses the platform-specific page size constant (PAGE_SIZE in file include/asm/page.h) where necessary.

Q4 (b) Describe the process of "Converting the Linear address into a physical address" through a diagram.

Answer

Linux adopted a three – level paging model so paging is feasible on 64 bit architectures. The x86 processor only supports a two – level conversion of the linear address. While Alpha processor supports three-level conversion because the Alpha processor supports linear addresses with a width of 64 bits.

Three level paging model defines three types of paging table:
(i) Page (Global) directory
(ii) Page middle directory
(iii) Page Table



Q5 (a) Discuss how Shared Memory is used for inter process communication.

Answer

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

In the Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on the mmap() function and on the system's native virtual memory facility. Solaris 2.x also supports System V shared memory, which is another way to let multiple processes attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using shmget(). The original owner of a shared memory segment can assign ownership to another user with shmctl(). It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using shmctl(). Once created, a shared segment can be

attached to a process address space using shmat(). It can be detached using shmdt(). The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in <sys/shm.h>.

Q5 (b) Explain how ptrace is used by debuggers.

Answer

ptrace is used by debuggers (such as gdb and dbx), by tracing tools like strace and ltrace, and by code coverage tools. ptrace is also used by specialised programs to patch running programs, to avoid unfixed bugs or to overcome security features. It can further be used as a sandbox and as a runtime environment simulator (like emulating root access for non-root software).

By attaching to another process using the ptrace call, a tool has extensive control over the operation of its target. This includes manipulation of its file descriptors, memory, and registers. It can single-step through the target's code, can observe and intercept system calls and their results, and can manipulate the target's signal handlers and both receive and send signals on its behalf. The ability to write into the target's memory allows not only its data store to be changed, but also the applications own code segment, allowing the controller to install breakpoints and patch the running code of the target.

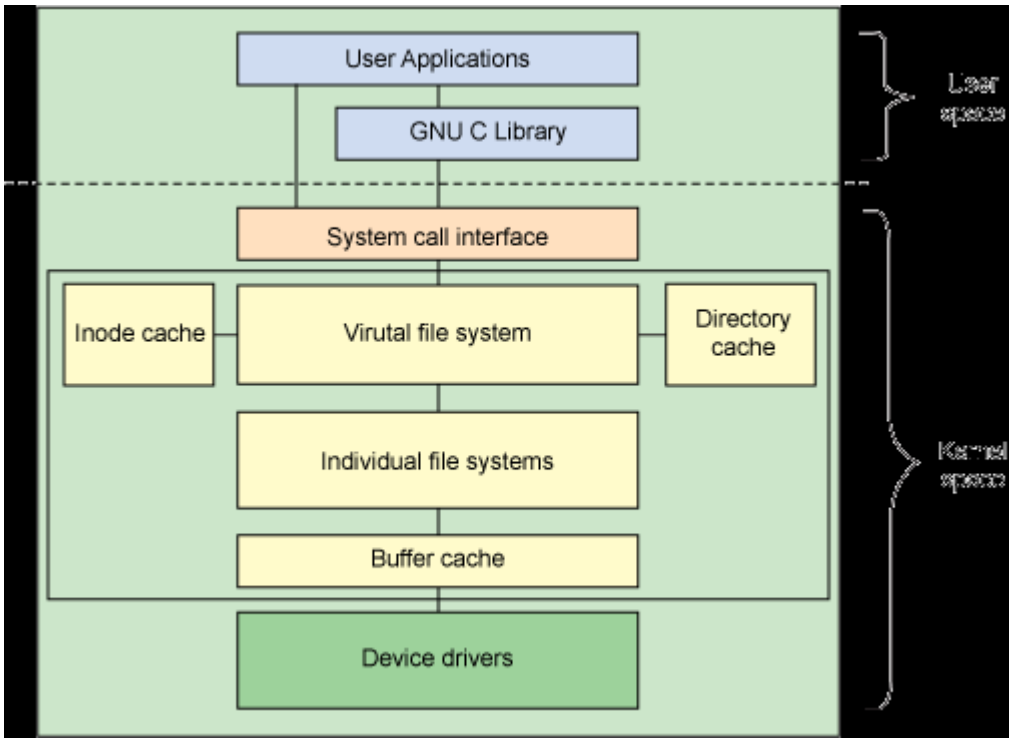
As the ability to inspect and alter another process is very powerful, ptrace can attach only to processes that the owner can send signals to (typically only their own processes); the superuser account can ptrace almost any process (except init). In Linux systems that feature capabilities based security, the ability to ptrace is further limited by the CAP_SYS_PTRACE capability. In FreeBSD, it's limited by FreeBSD jails and Mandatory Access Control policies.

Q6 (a) Describe the opening of a file operation of the textbook.

Answer

<u>High-level architecture</u>

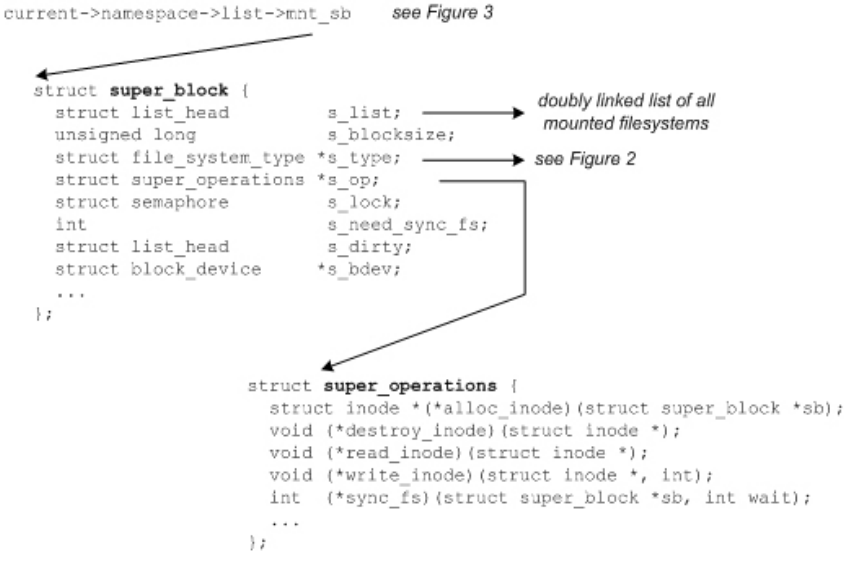
While the majority of the file system code exists in the kernel (except for user-space file systems), the architecture shown in Figure below shows the relationships between the major file system- related components in both user space and the kernel



Q6 (b) What is Superblock? What does it contain?

Answer

The superblock is a structure that represents a file system. It includes the necessary information to manage the file system during operation. It includes the file system name (such as ext2), the size of the file system and its state, a reference to the block device, and metadata information (such as free lists and so on). The superblock is typically stored on the storage medium but can be created in real time if one doesn't exist. You can find the superblock structure in the figure below.



One important element of the superblock is a definition of the superblock operations. This structure defines the set of functions for managing inodes within the file system. For example, inodes can be allocated with alloc_inode or deleted withdestroy_inode. You can read and write inodes with read_inode and write_inode or sync the file system with sync_fs. You can find the super_operations structure in ./linux/include/linux/fs.h. Each file system provides its own inode methods, which implement the operations and provide the common abstraction to the VFS layer.

Q7 (a) What is the difference between character and block devices?

Answer

There are two main types of devices under all systems, character and block devices. Character devices are those for which no buffering is performed, and block devices are those which are accessed through a cache. Block devices must be random access, but character devices are not required to be, though some are. Filesystems can only be mounted if they are on block devices.

Character devices are read from and written to with two function: foo_read() and foo_write(). The read() and write() calls do not return until the operation is complete. By contrast, block devices do not even implement the read() and write() functions, and instead have a function which has historically been called the ``strategy routine." Reads and writes are done through the buffer cache mechanism by the generic functions bread(), breada(), and bwrite(). These functions go through the buffer cache, and so may or may not actually call the strategy routine, depending on whether or not the block requested is in the buffer cache (for reads) or on whether or not the buffer cache is full (for writes). A request may be asyncronous: breada() can request the strategy routine to schedule reads that have not been asked for, and to do it asyncronously, in the background, in the hopes that they will be needed later.

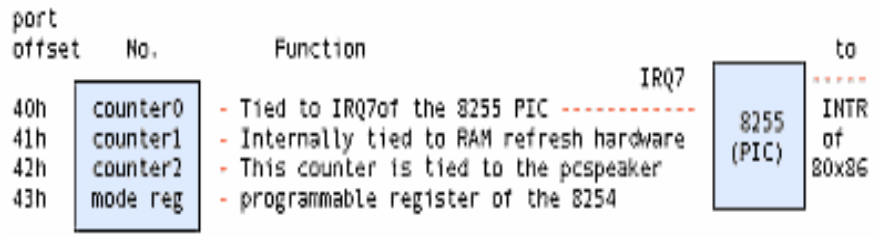
The sources for character devices are kept in .../kernel/chr_drv/, and the sources for block devices are kept in .../kernel/blk_drv/. They have similar interfaces, and are very much alike, except for reading and writing. Because of the difference in reading and writing, initialization is different, as block devices have to register a strategy routine, which is registered in a different way than the foo_read() and foo_write() routines of a character device drive.

Q7 (b) Discuss the method to create a Kernel Driver for the PC Speaker.

Answer

The internal speaker is tied to the buffered output of the 8254 timer chip on all PCs. The output of the 8254 timer is further latched through the

integrated system peripheral chip, through port 61h. A little chart is given below.

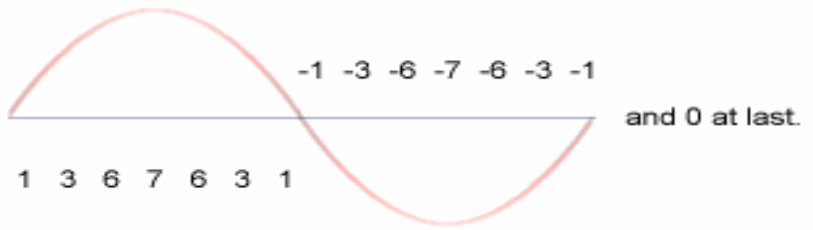


The base clock frequency of the 8254 is 1193180Hz which is 1/4 the standard NTSC frequency, incidentally. The counters have the values of the divisors, which, roughly speaking, are used to divide the base frequency. Thus the output of channel 0 will be at a frequency of 1193180Hz if counter0=1, 596590Hz if counter0=2 and so on. Therefore counter0=0 => a frequency of approximately 18.2 Hz, which is precisely the frequency at which the PIC is made to interrupt the processor.

Effectively this means that the value of counter0 will determine the frequency of the timer ISR is called. Changing counter 0 changes the rate at which the timer ISR is called. Therefore if the same person wrote both the code for the ISR, and that for programming counter 0 of the 8254 timer chip, then he could get his ISR called at a predetermined rate as required.

All this is leading to another aside.

When you hear sound, you know something near you is vibrating. If that something is a speaker cone, you know immediately that there is an electrical signal driving it. So we could always grab the signal generator by the scruff, if we want to snuff out the noise. If we want audio, we need a vibrating, or alternating, voltage. And we know that digital implies numbers, 1s and 0s. How do we put all of this stuff together and create digital audio?



If we run through the numbers in the diagram above, starting at 1 through 7 through 0 through -1 through -7 through -1 to 0, all in a second, we'd get a very approximate sine wave at 1Hz. Want a sine wave with a smoother curve? Just increase the number of samples you take per second. Here we've done 14. How about if it were 44000? That's the rate a CD player spews the numbers out to its DAC. DAC stands for Digital to Analog Converter, it's the little gadget that converts the 1s and 0s that make up the binary numbers that we are talking about into real analog time-varying voltage. Our little coding technique is called pulse code modulation. There are different ways to code the pulses, so we have PCM, ADPCM etc. The

waveform above could be termed "4bit, signed mono PCM at 14Hz" sampling rate.

We can develop a custom timer ISR to vibrate the speaker come at a pre-requisite frequency, so that all the ISR programmer has to do is to make the PC speaker cone move to the required amplitude (distance from the zero line) according to the sample value he gets from digital data, from a CDROM, for example. This means that we can set up a timer ISR for 44000Hz, and that is CD quality music staring at us! Perfect logic if you have a DAC to convert every sample into the corresponding analog voltage. In fact, the parallel port DAC driver does just that. Just rig a R - 2R ladder network of resistors and tie a capacitor across the output, feed it to any amplifier, even a microphone input will do, and voila, you have digital music!

All because the PC speaker is not at all tied to a DAC, but of all things, to a timer chip. Take look at the waveform output of a timer chip for, say, a sine wave:

We have two discrete values to play around with: One +5V, the other 0V and nothing in between. How do we get the Analog waveform? Oh man, why hast thou asked the impossible? Ask the designers at IBM who designed the first XT motherboards!

But we do have a very fragile, subtle solution. The techie terms are 1bit DAC, Chopping, and so on and so forth.

It's rather simple and easy to implement, and somewhere down the line, it was bound to happen.

The idea is to drive the PC speaker cone in bursts, when we can't quite push it up to the mark smoothly. Mind you, at 22Khz the cone is a mighty lazy bloke, it reluctantly moves up to the mark. Halfway through, take a rest so that if it's overdone and the cone has overshot, it gets time to come back down. Something like anti-lock brakes in automobiles. When you press the brake pedal half way down, the mechanism starts alternately pushing the brakes on and off. When you're standing on the pedal, the brake shoes are not quite stuck to the wheel drum, they're hammering at a furious pace. So you don't get a locked wheel. Similarly the more frequently you hammer the speaker cone with a +5V pulse, the farther it moves from the centerline. Bingo! Vary the frequency of pulsing according to the required amplitude. I named the DOS version fm.com just to remind myself that the idea was indeed ingenuous.

Now go back to the first figure and look at counter 2 of the 8254. Where does it lead to ? To the PC speaker, of course. Now all we have to do to get REAL sound, is to dump a scaled (remember 1 < countervalue < 65535) that is proportional to sample value (value => amplitude in PCM).

Q8 (a) Describe the socket structure with the help of a diagram, draw the socket and the relationship to its substructure.

Answer Page Number 236 of Textbook

Q8 (b) Discuss the network devices: PLIP and the dummy device in Linux.

Answer

A PLIP link used to connect two machines is a little different from an Ethernet. PLIP links are an example of what are called point-to-point links, meaning that there is a single host at each end of the link. Networks like Ethernet are called broadcast networks. Configuration of point-to-point links is different because unlike broadcast networks, point-to-point links don't support a network of their own. PLIP provides very cheap and portable links between computers.

The dummy device is a little exotic, but rather useful nevertheless. Its main benefit is with standalone hosts and machines whose only IP network connection is a dialup link. In fact, the latter are standalone hosts most of the time, too.

The dilemma with standalone hosts is that they only have a single network device active, the loopback device, which is usually assigned the address 127.0.0.1. On some occasions, however, you must send data to the "official" IP address of the local host.

Q9 (a) What are modules? Describe how data mapping takes place between modules.

Answer

Linux is a monolithic kernel; that is, it is one, single, large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative is to have a micro-kernel structure where the functional pieces of the kernel are broken out into separate units with strict communication mechanisms between them. This makes adding new components into the kernel via the configuration process rather time consuming. Say you wanted to use a SCSI driver for an NCR 810 SCSI and you had not built it into the kernel. You would have to configure and then build a new kernel before you could use the NCR 810. There is an alternative, Linux allows you to dynamically load and unload components of the operating system as you need them. Linux modules are lumps of code that can be dynamically linked into the kernel at any point after the system has booted. They can be unlinked from the kernel and removed when they are no longer needed. Mostly Linux kernel modules are device drivers, pseudo-device drivers such as network drivers, or file-systems.

You can either load and unload Linux kernel modules explicitly using the insmod and rmmod commands or the kernel itself can demand that the kernel daemon (kerneld) loads and unloads the modules as they are needed.

Dynamically loading code as it is needed is attractive as it keeps the kernel size to a minimum and makes the kernel very flexible. My current Intel kernel uses modules extensively and is only 406Kbytes long. I only occasionally use VFAT file systems and so I build my Linux kernel to automatically load the VFAT file system module as I mount a VFAT partition. When I have un mounted the VFAT partition the system detects that I no longer need the VFAT file system module and removes it from the system. Modules can also be useful for trying out new kernel code without having to rebuild and reboot the kernel every time you try it out. Nothing, though, is for free and there is a slight performance and memory penalty associated with kernel modules. There is a little more code that a loadable module must provide and this and the extra data structures take a little more memory. There is also a level of indirection introduced that makes accesses of kernel resources slightly less efficient for modules.

Q9 (b) What are the problems with multiprocessor systems? How are they overcome in UNIX-like systems?

Answer

For the correct functioning of a multitasking system, it is important that data in the kernel can only be changed by one processor so that identical resources cannot be allocated twice. In the Unix-like systems, there are two approaches to the solution of this problem. Traditional UNIX systems use a relatively coarse-grained locking; sometimes even the whole kernel is locked so that only one process can be present in the kernel. Some more advanced systems implement a finer-grained locking which, however, entails high additional expenditure and is normally used only for multiprocessor and real-time operating systems. In the latter, fine-grained locking reduces the time that a lock must be kept, thus allowing a reduction of the particularly critical latency time.

Text Book

Linux Kernel Internals, M. Beck, H. Bome, et al, Pearson Education, Second Edition, 2001