**Q2.2** What are the strengths and drawbacks of LINUX?

**ANSWER:**

STRENGTHS: The LINUX software is developed under open and distributed conditions. “Open” means that anyone can become involved if they are able to do so. This requires LINUX activists to be able to communicate quickly, efficiently, and above all, globally. The medium for this is the Internet. It is therefore no surprise that many of the developments are the product of gifted students with access to the Internet at their universities and colleges. The development systems available to these students tend to be relatively modest and therefore LINUX is still the 32-bit operating system that uses the least resources without sacrificing functionality. As LINUX is distributed under the conditions of the *GNU* Public License [GPL], the complete source code is available to users. This allows anyone to find out how the system works, and trace and remove any bugs.

DRAWBACKS: LINUX is a “programmer system” like UNIX. Cryptic commands, configurations that are difficult to follow, and documentation that is not always comprehensible make it far from easy to use – and not only for beginners.

**Q3.1** What is micro kernel? What is the main advantage and drawback of using micro kernel architecture?

**ANSWER:**

The micro kernel provides only the necessary minimum functionality of IPC and memory management and can be implemented in a small and compact 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.

This in itself results in a 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.

**Q3.2** Please explain the meaning of the system call *nice*.

**ANSWER:**

The system call *nice* is a little more complicated that the system call *getuid*. It expects a number by which the static priority of the current process is to be modified as its argument.

All system calls which process arguments must test the arguments for plausibility.

```
asmlinkage int sys_nice (int increment)
{
int newpriority;
```

Note that a larger argument for sys_nice( ) indicates a lower priority. This makes the name increment for the argument of nice a bit confusing.

```
if (increment  <  0  &&  !capable (CAP_SYS_NICE))
        return  –EPERM;
```

capable( ) checks whether the current process has the right to increase its priority. This is the case with the classical UNIX systems when the process has privileges. LINUX has a concept of subdividing these privileges in a finer way.

The new priority for the process can now be calculated. Among other things, a check is made at this point to ensure that the new priority for the process is within a reasonable range.

```
Newpriority = …

if      (newpriority  <  -20)
        newpriority  =  -20;
if      (newpriority  >  19 )
        newpriority  =  19;
current -> nice  = newpriority;

return 0;
} /* sys_nice */
```

**Q4.1** Describe the evolution of virtual memory in LINUX.

**ANSWER:**

As a shared library can be very large, it would not be a good idea if all its code were constantly being loaded into physical memory. We can be sure that the processes running at any one time will not be using all the functions in a library at the same time. Loading the code for unused functions squanders memory resources and is unnecessary. Even in larger programs there will certainly be sections of code which will never be touched by a process because, for example, certain program features are not used. Loading these parts of the program is just as wasteful as loading the unused sections of a library.

Virtual memory areas are frequently used by the hardware devices that map their memory in the address space. The communication between an application and the hardware is being carried out a lost faster via virtual memory areas than via system calls. Frame buffer devices are an example of this; using these, the memory of the graphics board can be mapped in a memory area.

If two processes are run by the same executable file, the program code does not need to be loaded into memory twice. Both processes can execute the same code in primary memory. It is also possible that large parts of the data segments of these processes will match. These also can be shared between the processes, provided neither process modifies this data. Only when a process modifies a page of memory does a copy-on-write need to be performed.

If a process reserves very large amounts of memory, the allocation of pages of physical memory would be extravagant. The process will only use these pages fully at a later stage, and possibly not even then. The way to get around this problem is to use copy-on-write, by which an empty page of memory is referenced more than once in the page tables for the process. It is only after a modification has been made at a specific address in the user segment that this page needs to be copied and mapped to the appropriate point in the linear address space.

It is clear from this that the separate areas of the user segment must have different attributes for the page table entries for the memory page, different handling routines for access errors, and different strategies to save to secondary memory. This justifies the evolution of virtual memory was introduced during the development of LINUX.

**Q4.2** Please provide a complete list of memory page flags along with the respective descriptions.

© IETE

**ANSWER:**

| Flags | Description |
|---|---|
| PG_locked | The page is locked |
| PG_error | This flag indicates an error condition |
| PG_referenced | This page has been recently accessed |
| PG_uptodate | This page matches the hard disk contents |
| PG_free_after | This page should be released after an I/O operation |
| PG_decr_after | The counter *nr_async_pages* is decremented after reading this page |
| PG_swap_unlock_after | After reading from the swap space, the page should be unlocked by calling *swap_after_unlock_page* ( ) function |
| PG_reserved | The page is reserved |

**Q5.1** Describe how a debugger uses ptrace.

**ANSWER:**

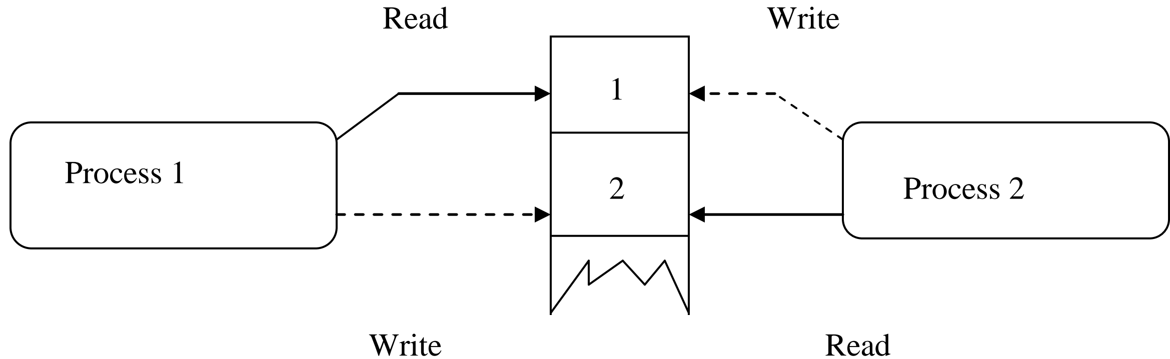
A debugger uses *ptrace* in the following way: it executes the system call fork and calls the function in the child process with PTRACE_TRACEME. The program to be inspected is started by execve. Since the PT_PTRACED flag is set, the *execve* call sends a SIGTRAP signal to itself. The system call will not allow *ptrace* to process programs for which S bit is set. It is not difficult to imagine the options that would otherwise be open to hackers. On return from *execve* the SIGTRAP signal is processed, the process is stopped, and the parent process is informed by being sent a SIGCHLD signal. The debugger will wait for this via the system call *wait*. It can then inspect the child process's memory, modify it and set breakpoints. The simplest way of doing this with x86 processor is to write an *int3* instruction at the appropriate address in the machine code. This instruction is only one byte long.

If the debugger calls *ptrace( )* with the request PTRACE_CONT, the child process will continue running until it processes the *int3* instruction, at which point the relevant interrupt handling routine sends a SIGTRAP signal to the child process, the child process is interrupted and the debugger is again informed. It could then, for example, simply abort the program to be inspected.

**Q5.2** Draw a diagram depicting a deadlock scenario while locking files. Explain briefly.

**ANSWER:**

Process 1 has locked the first byte in the file for read access and process 2 has locked the second byte. Process 1 then attempts to place a write lock on the second byte but is blocked by process 2. Process 2 in turn attempts to lock the first byte and is likewise blocked. Both processes would now wait for the other to release its lock, producing a deadlock situation.




**Q6.1** Describe the PROC file system. What are the disadvantages of using this file system?

**ANSWER:**

The PROC file system provides, in a portable way, information on the current status of the LINUX kernel and running process. It also allows modifications of kernel parameters in simple ways during runtime.

Each process in the system that is currently running is assigned a directory */proc/pid,* where *pid* is the process identification number of the relevant process.

.

Disadvantages: there is no interface for the individual files; every user has to find out where and how the information that is required is hidden in the file. Another disadvantage is that all information is output as strings, therefore conversion is always necessary for further processing.

**Q6.2** What entries are kept in the directory cache? Why?

**ANSWER:**

The directory cache comes originally from the *Ext2* file system. Since LINUX version 1.1.37 it has belonged to VFS and can be used by all file system implementations. In order to accelerate access via the reading of directories, directory entries are kept in this cache because they are needed to open files. It should provide a solution to the old problem that the user works with file names but the kernel works with *inodes*. The kernel must determine a name for the *inode*

and then again during the next access. In contrast to the *inodes* that exist permanently on the hard drive, the entries in the directory cache are purely RAM based.

**Q7.2** How many broad types of devices are allowed in LINUX? Describe them.

**ANSWER:**

There are two basic types of device: block-oriented devices and character-oriented devices.

Block devices are those to which there is random access, which means that any block can be read or written to at will. Under LINUX, these read and write accesses are handled transparently by the cache. Random access is an absolute necessity for file systems, which means that they can only be mounted on block devices.

Character devices, on the other hand, are devices which can usually only be processed sequentially and are therefore accessed without a buffer. This class includes the commonest hardware, such as sound cards, scanners, printers and so on, even where internal operation uses blocks. These blocks, however, are sequential in nature, and cannot be accessed randomly.

**Q8.1** Describe the layer model of the network implementation.

**ANSWER:**

As communication with network components presents a fairly complex task, it uses a layer structure like the file system. The individual layers correspond to levels of abstraction, with the level of abstraction increasing from layer to layer, starting with the hardware.

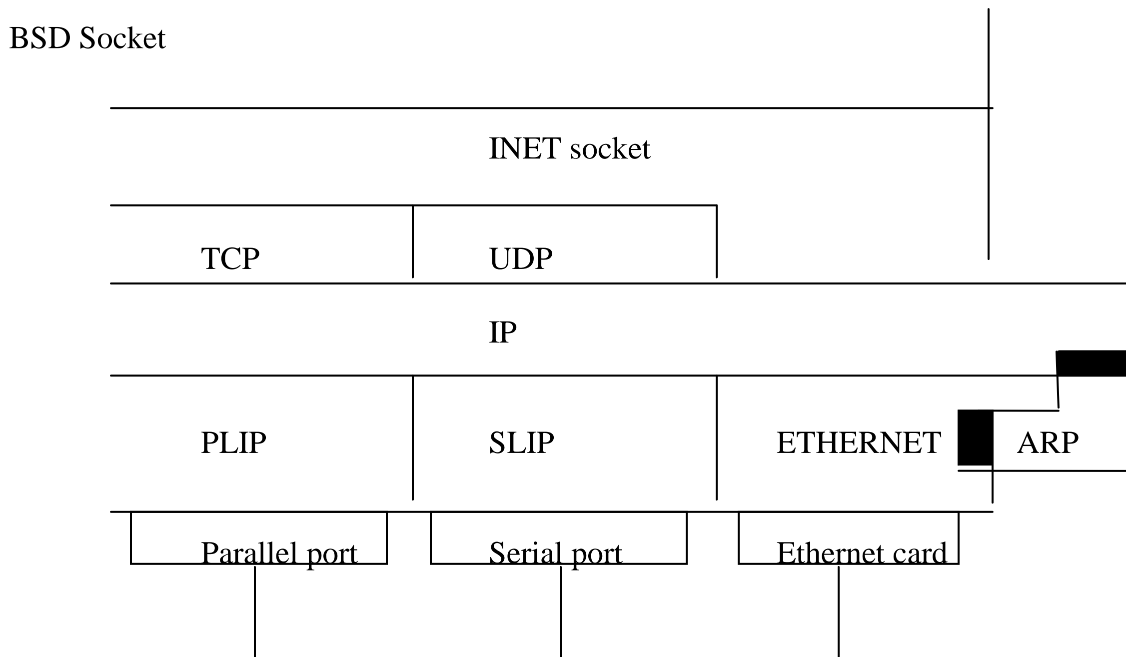
When a process communicates via the network, it uses the functions provided by the BSD socket layer. This takes care of a range of tasks similar to those handled by the virtual file system and administers a general data structure for sockets, which we shall call BSD sockets. The BSD socket interface has been selected by virtue of its widespread use, which simplifies the porting of network applications, most of which are already quite complex.

Below this layer is the INET socket layer. This manages the communication end points for the IP based protocols TCP and UDP. These are represented by the data structure sockets which we shall call INET sockets.

In the layer we have mentioned so far, no type distinction is as yet made between the sockets in the AF_INET address family. The layer that underlies the INET

socket layer, on the other hand, is determined by the type of socket, and may be the UDP layer, TCP layer or the IP layer directly. The UDP layer implements the user datagram protocol on the basis of IP, and the TCP layer similarly implements the Transmission Control Protocol for reliable communication links. The IP layer contains the code for the Internet Protocol version.4. This is where all the communication streams from the higher layers come together. Sockets of the other types are not included in this survey. Below the IP layer are the network devices, to which the IP passes the final packets. These then take care of the physical transportation of the information.

True communication always takes place between two sides, producing a two-way flow of information. For this reason, the various layers are also connected together in the opposite direction. This means that when IP packets are received, they are passed to the IP layer by the network devices and processes. The interaction between different layers is illustrated below:




**Q8.2** What are the differences between SLIP and PLIP?

**ANSWER:**

The most significant difference between SLIP and PLIP is that one protocol uses the computer's serial interface for data transfer while the other transfers data via the parallel port. When we speak of the parallel interface here, we do not mean Ethernet pocket adapters but the "bare" interface.

While PLIP enables a very powerful link to be set up between two computers, SLIP is the simplest way of connecting a computer or a local network to the internet via a serial link (a modern connection to a telephone network). SLIP and PLIP differ from Ethernet in that they can only transfer IP packets. For simplicity, SLIP does not even use a hardware header, nor does PLIP make great demands. It simply sets the hardware address to "fd:fd" plus the IP address and then uses the Ethernet functions for the protocol header.

**Q9.1** What are the problems with multiprocessing systems? How LINUX kernel handles these problems?

**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 microprocessors 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.

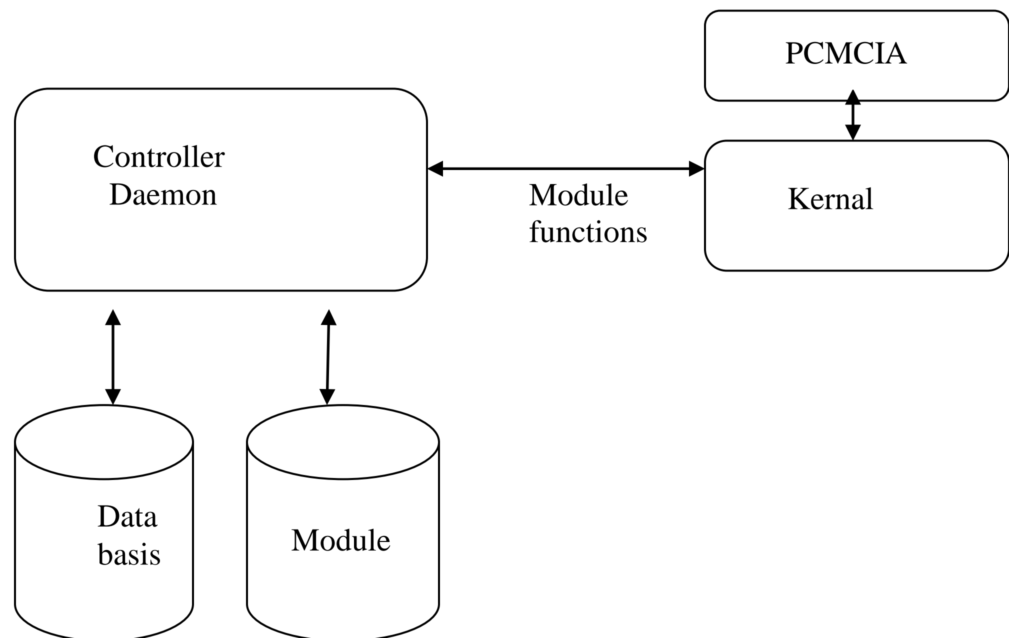
In the LINUX kernel implementation, various rules have been established. One of them is that no process running in kernel mode is interrupted by another process running in kernel mode, except when it releases control and sleeps. This rule ensures that large areas of the kernel are atomic with respect to other processes and thus simplifies many functions in the LINUX kernel.

A further rule establishes that interrupt handling cannot be interrupted by a process running in the kernel mode, but that in the end control is returned to this same process. A process can block interrupts and thus make sure that it will not be interrupted.

The last rule that is important for us states that interrupt handling cannot be interrupted by process running in the kernel mode. This means that interrupt handling will be processed completely, or at most be interrupted by another interrupt of higher priority.

**Q9.2** Draw a diagram depicting the Daemon for dynamic loading and unloading of modules.

**ANSWER:**




## TEXTBOOK

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

© IETE 