Q.2 a. Explain the main characteristics of Linux.
Answer: Page Number 2-4 of Text Book

b. What are additional configuration facilities?
Answer: Page Number 10-11 of Text Book

Q.3 a. Draw a diagram showing family relationships of processes.
Answer: Page Number 20 of Text Book

b. What are hardware interrupts? How they are used?
Answer: Page Number 32-33 of Text Book

Q.4 a. Explain how paging is implemented in Linux.
Answer: Page Number 80-82 of Text Book

b. Explain the four LRU lists.
Answer: Page Number 78 of Text Book

Q.5 a. What are the types of IPC supported by Linux?
Answer: Page Number 93 of Text Book

b. How is communication via files is performed?
Answer: Page Number 97-98 of Text Book

Q.6 a. Draw a diagram for the structure of a UNIX inode.
Answer: Page Number 2-4 of Text Book

Q.7 a. What is the concept of Polling and Interrupts?
Answer:

An Interrupt is an event external to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU Interrupts are needed due to the reason that People like connecting devices and a computer is much more than the CPU.it contains devices such as Keyboard, mouse, screen, disk drives Scanner, printer, sound card, camera, etc. These devices occasionally need CPU service ,but we can't predict when External events typically occur on a macroscopic timescale and we want to keep the CPU busy between events. An i**nterrupt** driven device driver is one where the hardware device being controlled will raise a hardware interrupt whenever it needs to be serviced. For example, an ethernet device driver would interrupt whenever it receives an ethernet packet from the network. The Linux kernel needs to be able to deliver the interrupt from the hardware device to the correct device driver. This is achieved by the device driver registering its usage of the interrupt with the kernel. It registers the address of an interrupt handling routine and the interrupt number that it wishes to own

Polling the device usually means reading its status register so often until the device's status changes to indicate that it has completed the request. As a device driver is part of the kernel it would be disasterous if a driver were to poll, since nothing else in the kernel

would run until the device had completed the request. Instead polling device drivers use system timers to have the kernel call a routine within the device driver at some later time. This timer routine would check the status of the command and this is exactly how Linux's floppy driver works. Polling by means of timers is at best approximate, a much more efficient method is to use interrupts

The difference between Interrupt and polling :

"Polling is like picking up your phone every few seconds to see if you have a call. Interrupts
are like waiting for the phone to ring."

- Interrupts win if processor has other work to do and event response time is not critical
- Polling can be better if processor has to respond to an event ASAP
  • May be used in device controller that contains dedicated secondary processor

b. What are the device drivers? What role does Operating System play for the same?

Answer:

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

A device driver simplifies programming by acting as translator between a hardware device and the applications or operating systems that use it. Programmers can write the higher-level application code independently of whatever specific h. Physical layers communicate with specific device instances.

In Linux environments, programmers can build device drivers either as parts of the kernel or separately as loadable modules

Q.8 a. Explain the layer structure of a network.

Answer: Page Number 233 of Text Book

b. Explain flags of the network devices.

Answer: Page Number 251 of Text Book

Q.9 a. What are modules? What can be implemented in 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. For example if we want to use a SCSI driver for an NCR

810 SCSI and if there is already not built it into the kernel. Then we would have to configure and then build a new kernel before we could use the NCR 810. There is an alternative, Linux allows us to dynamically load and unload components of the operating system as you need them. **Linux modules** are small 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.

We 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. 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.

Once a Linux module has been loaded it is as much a part of the kernel as any normal kernel code. It has the same rights and responsibilities as any kernel code; in other words, Linux kernel modules can crash the kernel just like all kernel code or device drivers can.

b. What is the need for coding Atomic Operations in the header file asm / atomic.h?

Answer:

The simplest means of synchronization in the Linux kernel are the atomic operations. *Atomic* means that the critical section is contained within the API function. No locking is necessary because it's inherent in the call. As C can't guarantee atomic operations, Linux relies on the underlying architecture to provide this. Since architectures differ greatly, you'll find varying implementations of the atomic functions. Some are provided almost entirely in assembly, while others resort to C and disabling interrupts using local _irq_save and local_irq_restore.

The atomic operators are ideal for situations where the data we need to protect is simple, such as a counter. While simple, the atomic API provides a number of operators for a variety of situations. Here's a sample use of the API.

To declare an atomic variable, you simply declare a variable of typeatomic_t. This structure contains a single int element. Next, you ensure that your atomic variable is initialized using the ATOMIC_INITsymbolic constant. In the case shown in Listing 1, the atomic counter is set to zero. It's also possible to initialize the atomic variable at runtime using the atomic_set function

Listing 1. Creating and initializing an atomic variable

```
atomic_t my_counter =
ATOMIC_INIT(0);

... or ...

atomic_set( &my_counter, 0 );
```

The atomic API supports a rich set of functions covering many use cases. You can read the contents of an atomic variable withatomic_read and also add a specific value to a variable with atomic_add. The most common operation is to simply increment the variable, which is provided with atomic_inc. The subtraction operators are also available, providing the converse of the add and increment operations. Listing 2 demonstrates these functions.

Listing 2. Simple arithmetic atomic functions

```
val = atomic_read( &my_counter );

atomic_add( 1, &my_counter );

atomic_inc( &my_counter );

atomic_sub( 1, &my_counter );

atomic_dec( &my_counter );
```

The API also supports a number of other common use cases, including the operate-and-test routines. These allow the atomic variable to be manipulated and then tested (all performed as one atomic operation). One special function called atomic_add_negative is used to add to the atomic variable and then return true if the resulting value is negative. This is used by some of the architecture-dependent semaphore functions in the kernel.

Text Book

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