Q1. How are traps and exceptions related to interrupts? Explain the sequence of events when devices assert an active low digital signal to ask CPU attention to read 100 bytes of data.

i) An interrupt (execution of INT instruction or active low on a hardware pin) does a context switch, thus transferring control to interrupt service routine in the kernel. Exception occurs as a side-effect of instruction executions and can result in either aborts (divide by zero) or traps (need to switch user code to kernel). Exception transfer control to kernel code like interrupts.

ii) An interrupt signal when admitted results in a context switch and by using interrupt vector as an index into interrupt vector table which points to an interrupt service routine (ISR) in kernel code space. The ISR reads the data, saves it some buffer in kernel (user) memory space. Control returns to user program which was interrupted.

Q2. Draw a labelled diagram to explain various OS services. How are these services accessed by a user program?

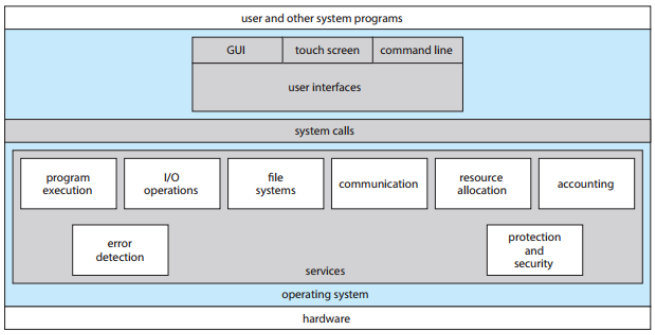

Figure 2.1 A view of operating system services.

A user program calls library functions which in turn makes the system calls. This indirection helps keep the language library function names to remain same across platforms (e.g. they are same on Windows and Unix/Linux, even though a "printf" library call might makes different system calls on these platforms.

Q3. Compare and contrast monolithic and microkernel architecture for OS. Draw both architectures first.

A monolithic kernel is one large executable that runs in one address space. This helps optimal execution times but an error result in OS crash. On the other hand, a microkernel approach only keeps the bare minimum functions in the kernel and implements all other OS services as separate sub-systems that run in the user space. Here, any errors in the sub-systems will only crash the sub-system and not the kernel. The key drawback of microkernel is the delay in processing system calls as the kernel now must interact with many sub-systems (each running in its own address space) before returning the call.

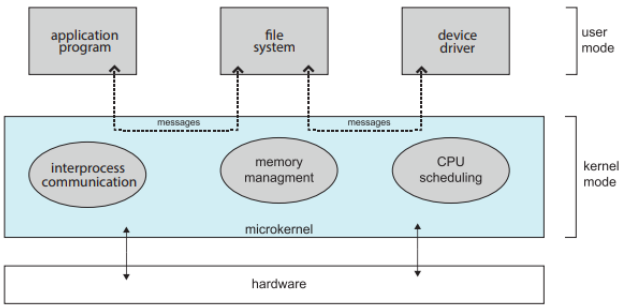

Figure 2.15 Architecture of a typical microkernel.

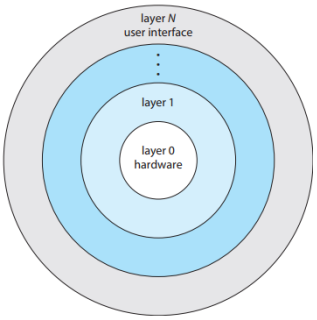

Figure 2.14 A layered operating system.

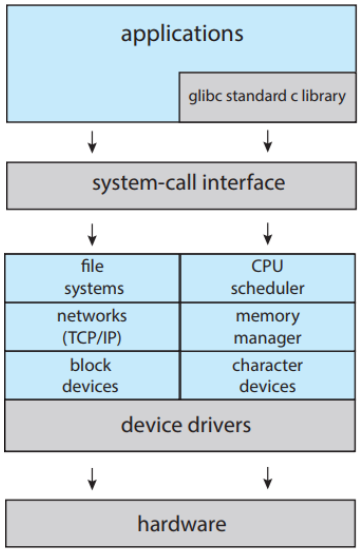

Figure 2.13 Linux system structure.