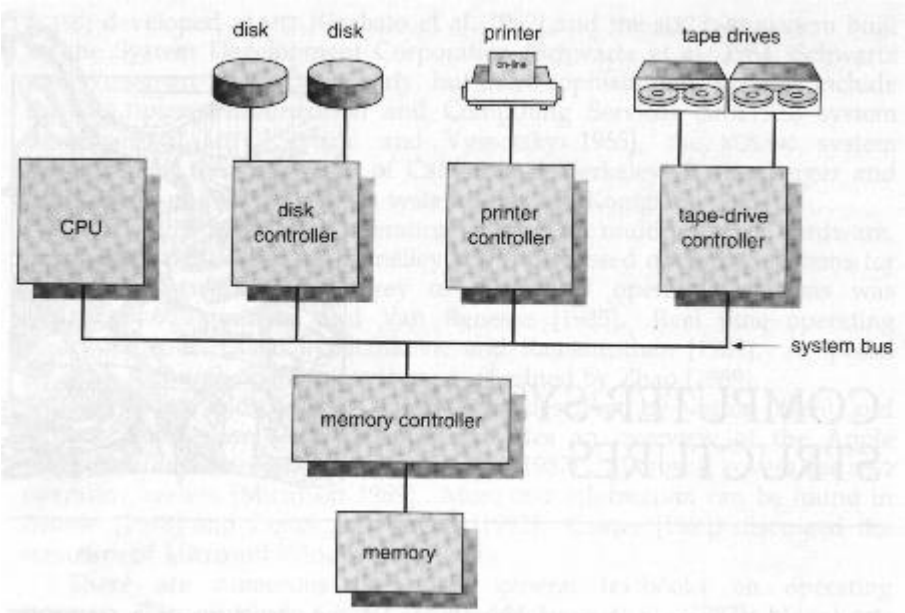
Computer-System Operation

A modern, general-purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus that provides access to shared memory (see figure below). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently (at the same time), competing for memory cycles. To ensure orderly access to the shared memory (RAM), a memory controller is provided whose function is to synchronize access to the memory.

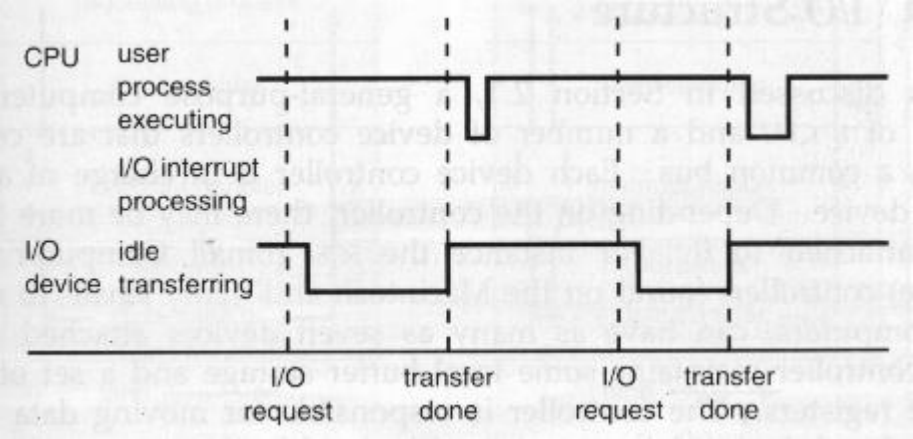


A modern computer system.

For a computer to start running - for instance, when it is powered up or rebooted it needs to have an initial program to run. This initial program, or **bootstrap program,** tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and to start it executing. To accomplish this goal, the bootstrap program must locate the operating system **kernel** (core part of the operating system) and load it into memory. The operating system then starts executing the first process, such as "init", and waits for some event to occur. The occurrence of an event is usually signaled by an **interrupt** (signal to the CPU requesting attention) from either the hardware or software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

There are many different types of events that may trigger an interrupt, for example, the completion of an I/O operation, division by zero, invalid memory access, and a request for some operating system service. For each such interrupt, a service routine is provided that is responsible for dealing

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes, and upon completion, the CPU resumes the interrupted computation. A time line of this operation is shown in the figure below.



Interrupt time line for a single process doing output.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information, and it, in turn, would call the interrupt-specific handler. However, interrupts must be handled very quickly, and given that there are a predefined number of possible interrupts, a table of pointers to interrupt routines may be used instead. The interrupt routine is then called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first 100 or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector,** of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as MS-DOS and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state, for instance, by modifying register values, it must explicitly save the current state and then restore it before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation will resume as though the interrupt had not occurred.

Usually, interrupts are **disabled** while an interrupt is being processed, delaying any incoming interrupts until the operating system is done with the current one, after which interrupts are **enabled**. If they were not thus disabled, the processing of the second interrupt while the first was being serviced would overwrite the first's data, and the first would be a **lost interrupt**. Sophisticated interrupt architectures allow for one interrupt to be processed during another. They often use a

and interrupt processing information is stored separately for each priority. A higher-priority interrupt will be taken even if a lower-priority interrupt is active, but interrupts at the same or lower levels are **masked,** or selectively disabled, to prevent lost interrupts or unnecessary ones.

Modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt, or a trap. A **trap** (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access), or by a specific request from a user program that an operating-system service be performed.

The interrupt-driven nature of an operating system defines that system's general structure. When an interrupt (or trap) occurs, the hardware transfers control to the operating system. First, the operating system preserves the state of the CPU by storing registers and the program counter. Then, it determines which type of interrupt has occurred. This determination may require **polling,** the querying of all I/O devices to detect which requested service, or it may be a natural result of a vectored interrupt system. For each type of interrupt, separate segments of code in the operating system determine what action should be taken.

Last Updated Jul.28/99