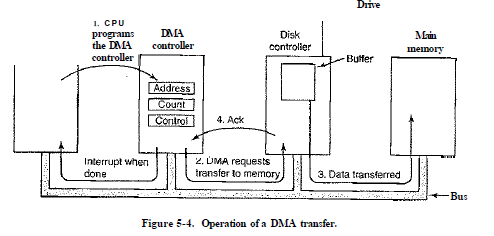
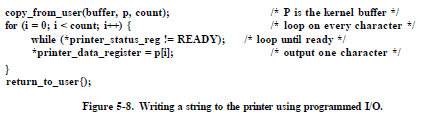
**Chapter 5 (I/O)**

* I/O devices can be roughly divided into two categories: block devices and character devices.
* Block device is one that stores information in fixed-size blocks each with its own address. (block sizes range from 512 bytes to 32,768 bytes), transfers are in units of one or more entire blocks. Property of a block device is to read or write each block independently. Examples, Hard disks, CD-ROMs, and USB sticks.
* Character device delivers or accepts a stream of characters, regardless to any block structure. It is not addressable and does not have any seek operation. Examples, Printers, network interfaces, mice.
* Clocks are not block addressable nor do they generate or accept character streams, they cause interrupts at well-defined intervals.
* Memory-mapped screens are not block addressable or character streams.
* Disk controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary.
  + The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum (Error-Correcting Code (ECC)) has been verified and the block has been declared to be error free, it can then be copied to main memory.
* Controller for a monitor also works as a bit serial device.
* Each controller has a few registers that are used for communicating with the CPU.
  + By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action.
  + By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command.
* many devices have a data buffer that the operating system can read and write.
* Alternatives help CPU communicates with the control registers and the device data buffers:
  + First approach, I/O port space contains all control registers’ I/O port numbers (8- or 16-bit integer), port number for each register, only the operating system can access it.
  + Second approach, map all the control registers into the memory space, and each control register is assigned a unique memory address to which no memory is assigned usually at the top of the address space. (this approach called memory-mapped I/O).
  + Third, a hybrid scheme, with memory-mapped I/O data buffers and separate I/O ports for the control registers. The **Pentium** uses it.
* When the CPU wants to read a word, either from memory or from an I/O port,
  + it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line.
  + A second signal line is used to tell whether I/O space or memory space is needed.
    - If it is memory space, the memory responds to the request.
    - If it is I/O space, the I/O device responds to the request.
  + If there is only memory space, every memory module and every I/O device compares the address lines to the range of addresses that it services.
  + If the address falls in its range, it responds to the request.
  + Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.
* Advantages of memory-mapped I/O:
  + With memory-mapped I/O, device control registers are just variables in memory and can be addressed in C like any other variables. Thus I/O device driver can be written entirely in C, and without memory-mapped I/O, some assembly code is needed and adds overhead to controlling I/O.
  + No special protection mechanism is needed to keep user processes from performing I/O.
  + Every instruction that can reference memory can also reference control registers.
* Disadvantages of memory-mapped I/O:
  + Caching a device control register would be disastrous.
  + If there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to.
* Special measures have to be taken to make memory-mapped I/O work on a system with multiple buses:
  + One possibility is to first send all memory references to the memory. If the memory fails to respond, then the CPU tries the other buses. (requires additional hardware complexity)
  + A second possible design is to put a snooping device on the memory bus to pass all addresses presented to potentially interested I/O devices. (problem: I/O devices may not be able to process requests at the speed the memory can)
  + third possible design is to filter addresses in the PCI bridge chip. This chip contains range registers that are pre-loaded at boot time. (needs to know at boot time which memory addresses are not really memory addresses).
* CPU needs to address the device controllers to exchange data with them. The operating system can only use DMA if the hardware has a DMA (Direct Memory Access) controller.
* A single DMA controller is available for regulating transfers to multiple devices.
* Disk reads occur if DMA is not used. First the disk controller reads the block from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.
* If DMA is used, the procedure is different.
  + **Step 1** First the CPU programs the DMA controller by setting its registers so it knows what to transfer where It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.
  + **Step 2** The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller.
  + **Step 3** the memory address to write to is on the bus' address lines so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle.
  + **Step 4** When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus
  + The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0.
  + the DMA controller interrupts the CPU to let it know that the transfer is now complete.



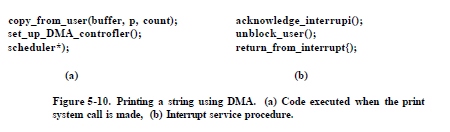
* After each word is transferred (steps 2 through 4), the DMA controller decides which device to service next. It may be set up to use a round-robin algorithm, or it may have a priority scheme design.
* Many buses can operate in two modes: word-at-a-time mode and block mode.
* In word-at-a-time mode, the DMA controller requests for the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. (cycle stealing)
* In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. (burst mode).
* Burst mode is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition.
* The down side to burst mode is that it can block the CPU and other devices for a substantial period of time if a long burst is being transferred.
* Most DMA controllers use physical memory addresses for their transfers. Operating system converts the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register.
  + An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead.
* The disk first reads data into its internal buffer before DMA can start. There are two reasons.
  + First, by doing internal buffering, the disk controller can verify the checksum before starting a transfer. If the checksum is incorrect, an error is signaled and no transfer is done.
  + The second reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate.
* An interrupt that leaves the machine in a well-defined state is called a precise interrupt, it has four properties:
  + The PC (Program Counter) is saved in a known place.
  + All instructions before the one pointed to by the PC have fully executed.
  + No instruction beyond the one pointed to by the PC has been executed.
  + The execution state of the instruction pointed to by the PC is known.
* In Imprecise interrupt, where different instructions near the program counter are in different stages of completion, with older ones not necessarily more complete than younger ones. Machines with imprecise interrupts usually vomit a large amount of internal state onto the stack to give the operating system the possibility of figuring out what was going on.
* Some computers are designed so that some kinds of interrupts and traps are precise and others are not.
  + For example, having I/O interrupts be precise but traps due to fatal programming errors be imprecise is not so bad since no attempt need be made to restart a running process after it has divided by zero.
  + Some machines have a bit that can be set to force all interrupts to be precise.
* Goals of the I/O Software:
  + **Device independence** means is that it should be possible to write programs that can access any I/O device without having to specify the device in advance. (Example, a program that reads a file as input should be able to read a file on a hard disk)
  + **Uniform naming** where name of a file or a device should simply be a string or an integer and not depend on the device in any way. (For example, a USB stick can be mounted on top of the directory /usr/ast/backup so that copying a file to /usr/astAbackup/monday copies the file to the USB stick.)
  + **Error handling** where errors should be handled as close to the hardware as possible.' If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it.
  + **Synchronous** (blocking) versus **asynchronous** (interrupt-driven) transfers, most physical I/O is asynchronous.
  + **Buffering**, means data that come off a device cannot be stored directly in its final destination. (digital audio devices)
  + **Sharable** versus **dedicated devices**, disks can be used by many users at the same time, tape drives have to be dedicated to a single user until that user is finished.
* Three fundamentally different ways that I/O can be performed programmed I/O (Simplest one), interrupt-driven I/O, and I/O using DMA. Programmed I/O is simple but has the disadvantage of tying up the CPU full time until all the I/O is done.
  + Operating system copies the buffer with the string to an array, say, p, in kernel space. (Example, Print string using Printer).



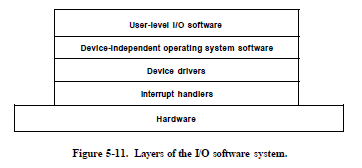
* In Interrupt-Driven I/O, printing is done on a printer that does not buffer characters but prints each one as it arrives.
  + The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts. When the printer has printed the character and is prepared to accept the next one, it generates an interrupt.
  + Disadvantage of interrupt-driven I/O is that an interrupt occurs on every character. Interrupts take time.



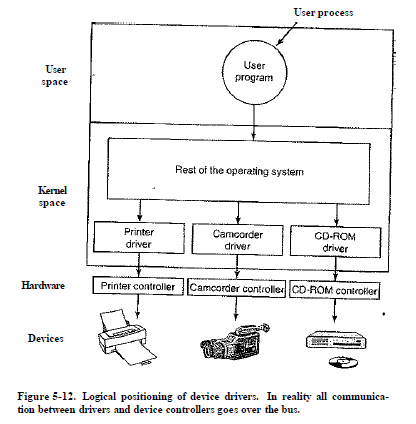
* DMA solves Interrupt-Driven I/O issue. DMA is programmed I/O, DMA controller feed the characters to the printer one at time, without the CPU being bothered.
  + This strategy requires special hardware.
  + The DMA controller is usually much slower than the main CPU.



* I/O software is typically organized in four layers:



* Interrupt Handler:
  + The driver starting an I/O operation block until the I/O has completed and the interrupt occurs. When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that started it. This model works best if drivers are structured as kernel processes, with their own states, stacks, and program counters. interrupt processing takes a considerable number of CPU instructions:
  + 1. Save any registers that have not already been saved by the interrupt hardware.
  + 2. Set up a context for the interrupt service procedure. Doing this may involve setting up the TLB, MMU and a page table.
  + 3. Set up a stack for the interrupt service procedure.
  + 4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
  + 5. Copy the registers from where they were saved to the process table.
  + 6. Run the interrupt service procedure. It will extract information from the interrupting device controller's registers.
  + 7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
  + 8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
  + 9. Load the new process' registers.
  + 10. Start running the new process.
* Device Drivers
  + Device driver is a specific code for controlling each I/O device attached to a computer. It’s written by the device's manufacturer and delivered along with the device.
  + Each device driver normally handles one device type, or at most, one class of closely related devices.
  + They need to be operating system **kernel**, but it is possible to construct drivers that run in **user space**, with system calls for reading and writing the device registers (Example, MINIX 3).
    - This design isolates the kernel from the drivers and the drivers from each other, eliminating a major source of system crashes or buggy drivers that interfere with the kernel in one way or another.
  + Drivers are classified into the block devices (disks), the character devices, (keyboards and printers).
  + One model is that most operating systems defined a standard interface that all block drivers must support and a second standard interface that all character drivers must support. These interfaces consist of a number of procedures that the rest of the operating system can call to get the driver to do work for it. (recompile Kernel)
  + Instead, drivers were dynamically loaded into the system during execution.



* + Device driver’s functions:
    - Accepts abstract read and write requests from the device-independent software above it.
    - Driver must initialize the device,
    - Manage its power requirements and log events.
  + Controlling the device means issuing a sequence of commands to it.
  + Drivers have to be reentrant, meaning that a running driver has to expect that it will be called a second time before the first call has completed.
* Device-Independent I/O Software
  + Basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software.
  + Functions can be done in the device-independent software:
    - Uniform interfacing for device drivers
    - Buffering
    - Error reporting
    - Allocating and releasing dedicated devices
    - Providing a device-independent block size
  + Uniform interfacing for device drivers:
    - If each device driver has a different interface, the driver functions available for the system to call differ from driver to driver, that means that interfacing each new driver requires a lot of new programming effort. Instead use the same interface. Examples for disks: read and write, turning the power on and off, and formatting.
    - Another aspect of having a uniform interface is how I/O devices are named (provides protection)
  + Buffering
    - Double buffering means using second buffer when the essential (first) buffer is full.
    - Circular buffer with two pointers. One points to the next free word, where new data can be placed and the other points to the first word of data in the buffer that has not been removed yet.
    - A a downside of buffers, if data get buffered too many times, performance suffers.
  + Error reporting:
    - Many errors are device-specific handled by the driver, but the framework for error handling is device independent.
    - Programming errors occur when a process asks for something impossible, such as writing to an input device, other errors are providing an invalid buffer address.
    - Actual I/O errors, example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off.
* User-Space I/O Software
  + Most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs. Examples,
    - Formatting of input and output is done by library procedures
    - The standard I/O library contains a number of procedures that involve I/O and all run as part of user programs.
  + Spooling is user-level I/O software doesn’t consist of library procedures and a way of dealing with dedicated I/O devices in a multiprogramming system. (Printer)
    - creates a special process, called a daemon, and a special directory, called a spooling directory.
    - another example, file transfer over a network often uses a network daemon.

