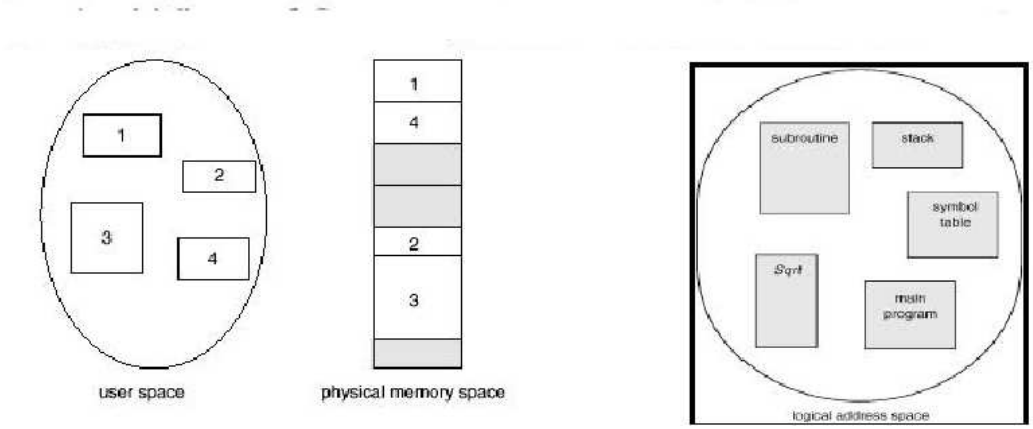

Figure 1: Logical View of Segmentation (left) , User's View of a Program (right).

0.1 Segmentation

- The most important problem with *base-and-limits* (see Fig. **??**) relocation is that there is only one segment for each process
- Segmentation generalizes the base-and-limits technique by allowing each process to be split over several segments (i.e., multiple base-limits pairs)
- *Segment table* maps 2-dimensional physical addresses (segment-number, offset); each table entry has:
  - base; contains starting physical address where segments reside in memory
  - *limit* specifies length of segment
- Table entries are filled as new segments are allocated for the process
- A segment is a region of contiguous memory. Although the segments may be scattered in memory, each segment is mapped to a contiguous region
- Memory-management scheme that supports user view of memory (see Fig. 1)
- Program is collection of segments. Segment a logical unit such as: main program, procedure, function, method, object, local variables,

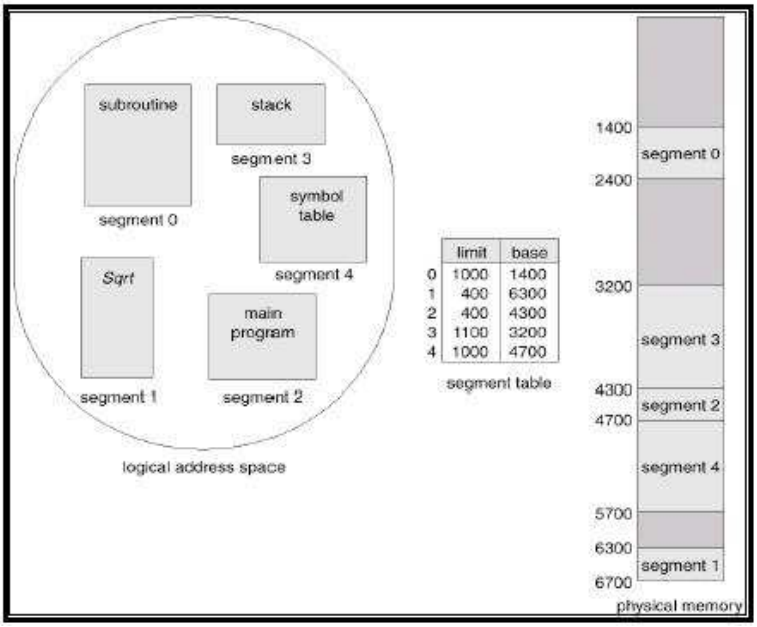

Figure 2: Example of Segmentation

global variables, common block, stack, symbol table, array (see Fig. 2)

- When a process is created, an empty segment table is inserted into the process control block (PCB)
- The segments are returned to the free segment pool when the process terminates
- Segmentation, as well as the base and limits approach, causes external fragmentation (because they require contiguous physical memory) and requires memory compaction
- An advantage of the approach is that only a segment, instead of a whole process, may be swapped to make room for the (new) process.
- Like paging, use virtual addresses and use disk to make memory look bigger that it really is
- Segmentation can be implemented with or without paging
- *Segment-table base register (STBR)* points to segment table's location in memory

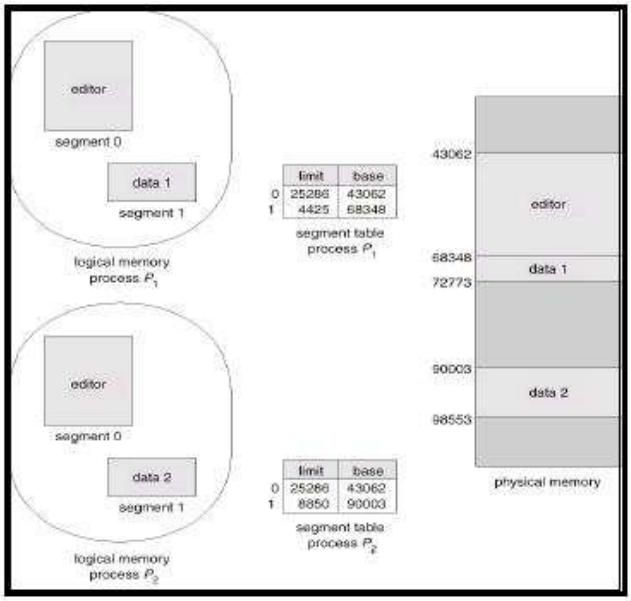

Figure 3: Sharing of Segmentation

- *Segment-table length register (STLR)* indicates number of segments used by a program , segment number s is legal if $s <$ STLR
- Segmentation Architecture
  - Relocation; dynamic, by segment table
  - Sharing; shared segments, same segment number
  - Allocation; first fit/best fit, external fragmentation
  - Protection: with each entry in segment table: illegal segment, read/write/execute privileges
  - Protection bits associated with segments; code sharing at segment level
  - Since segments vary in length, memory allocation a dynamic storage-allocation problem

0.1.1 Segmentation with Paging

- Advantages of Segmentation
  - Different protection for different segments read-only status for code

  - Enables sharing of selected segments (see Fig. 3)
  - Easier to relocate segments than entire address space
  - Enables sparse allocation of address space

- Disadvantages of Segmentation
  - Still expensive/difficult to allocate contiguous memory to segments
  - External fragmentation: Wasted memory
  - Paging; Allocation easier, Reduces fragmentation

- Advantages of Paging
  - Fast to allocate and free;
    * Alloc: Keep free list of free pages and grab first page in list, no searching by first-fit, best-fit
    * Free: Add page to free list, no inserting by address or size
  - Easy to swap-out memory to disk
    * Page size matches disk block size
    * Can swap-out only necessary pages
    * Easy to swap-in pages back from disk

- Disadvantages of Paging
  - Additional memory reference: Inefficient. Page table too large to store as registers in MMU. Page tables kept in main memory. MMU stores only base address of page table.
  - Storage for page tables may be substantial
    * Simple page table: Require entry for all pages in address space. Even if actual pages are not allocated
    * Partial solution: Base and bounds (limits) for page table. Only processes with large address spaces need large page tables. Does not help processes with stack at top and heap at bottom.
  - Internal fragmentation: Page size does not match allocation size
    * How much memory is wasted (on average) per process?
    * Wasted memory grows with larger pages

- Combine Paging and Segmentation

  - Structure
    - Segments correspond to logical units: code, data, stack. Segments vary in size and are often large
    - Each segment contains one or more (fixed-size) pages
  - Two levels of mapping to make tables manageable (2 look-ups!)
    - Page table for each segment
    - Base (real address) and bound (size) for each page table

- Segments + Pages Advantages
  - Advantages of Segments
    - Supports sparse address spaces. If segment is not used, no need for page table. Decreases memory required for page tables.
  - Advantages of Paging
    - Eliminate external fragmentation
    - Segments to grow without any reshuffling
  - Advantages of Both. Increases flexibility of sharing. Share at two levels: Page or segment (entire page table)

- Segments + Pages Disadvantages
  - Internal fragmentation increases. Last page of every segment in every process
  - Increases overhead of accessing memory
    - Translation tables in main memory
    - 1 or 2 overhead references for every real reference
  - Large page tables
    - Do not want to allocate page tables contiguously
    - More problematic with more logical address bits
    - Two potential solutions: Page the user page tables (multilevel page table), Inverted page table

0.1.2 Segmentation with Paging: MULTICS

- MULTICS solved problems of external fragmentation and lengthy search times by paging segments

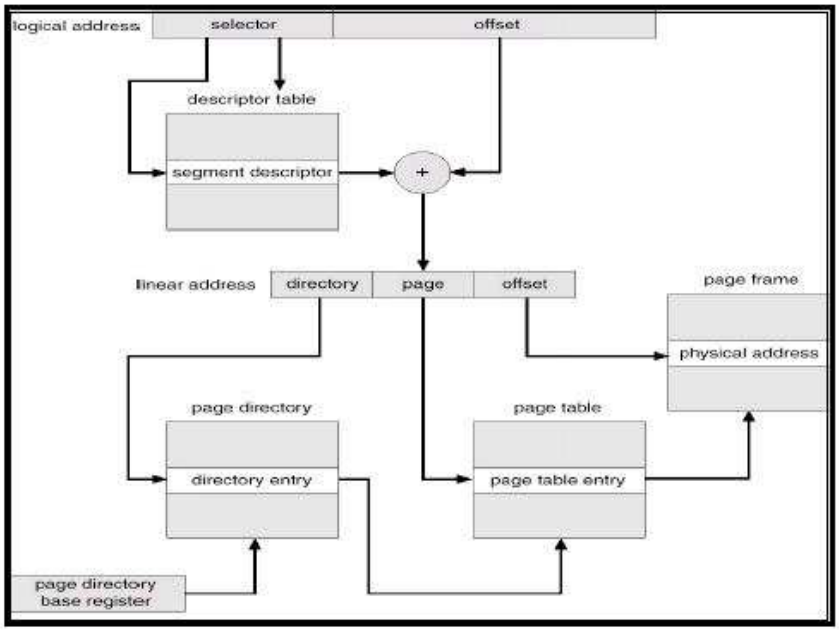

Figure 4: Intel 386 Address Translation

- Solution differs from pure segmentation: segment-table entry contains not base address of segment, but base address of *page table* for this segment

0.1.3 Segmentation with Paging: The Intel Pentium (see Fig. 4)

- Intel 386 and later use segmentation with paging
- OS/2 uses full scheme
- Other OSes mostly only use pages; Linux, Windows NT and successors

1 INPUT/OUTPUT

1.1 Principles of I/O Hardware

- There exists a large variety of I/O devices:
  - Many of the with different properties
  - They seem to require different interfaces to manipulate and manage them

Table 1: Device I/O Port Locations on PCs (Partial).

I/O address range (hexadecimal)	Device
000-00F	DMA Controller
020-021	Interrupt Controller
040-043	Timer
200-20F	Game Controller
2F8-2FF	Serial port (secondary)
320-32F	Hard disk Controller
378-37F	Parallel port
3D0-3DF	Graphics Controller
3F0-3F7	Diskette drive Controller
3F8-3FF	Serial port (primary)

- We don't want a new interface for every device
- Diverse, but similar interfaces leads to code duplication
- Challenge: Uniform and efficient approach to I/O
- Common concepts
  - port
  - bus
  - controller (host adapter)
- each port is given a special address (see Table 1)
- **communication**, use an assembly instruction (high–level languages only work with main memory) to read/write a port; e.g., OUT *port, reg*: writes the value in CPU register *reg* to I/O port *port*
- **protection**, users should have access to some I/O devices but not to others
- I/O instructions control devices
- Devices have addresses, used by
  - direct I/O instructions
  - memory-mapped I/O

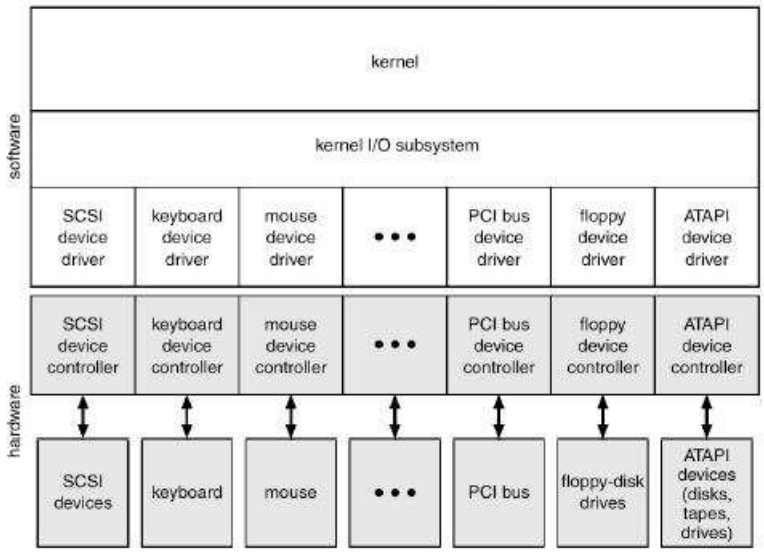

Figure 5: A kernel I/O structure.

1.1.1 Device Controllers (see Fig. 5)

- I/O devices have controllers; disk controller, monitor controller, etc.
- controller manipulates/interprets electrical signals to/from the device
- controller accepts commands from CPU or provides data to CPU
- controller and CPU communicate over I/O ports; control, status, input and output registers

1.1.2 I/O Devices

- Categories of I/O Devices (by usage)
  - Human readable
    - Used to communicate with the user
    - Printers, Video Display, Keyboard, Mouse
  - Machine readable
    - Used to communicate with electronic equipment
    - Disk and tape drives, Sensors, Controllers, Actuators
  - Communication

        * Used to communicate with remote devices
        * Ethernet, Modems, Wireless

- I/O system calls abstract device behaviors in generic classes (see Fig. 5)
- Device-driver layer hides I/O-controller differences from kernel
- Devices vary in many dimensions
    - character-stream or block
    - sequential or random-access
    - sharable or dedicated
    - speed of operation
    - read-write, read only, or write only
- Block and Character Devices; Block devices include disk drive
    - commands include read, write, seek
    - raw I/O or file-system access
    - file system maps location i onto block + offset
    - memory-mapped file access possible
- Character devices include keyboard, mouse, serial port
    - commands include get, put
    - libraries layered on top allow line editing

1.1.3 Characteristics (see Table 2) and Differences in I/O Devices

- Application
    - Disk used to store files requires file management software
    - Disk used to store virtual memory pages needs special hardware and software to support it
    - Terminal used by system administrator may have a higher priority
- Complexity of control;
    - Unit of transfer; Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk

Table 2: Characteristics of I/O Devices

aspect	variation	example
data transfer mode	character, block	terminal, disk
access method	sequential, random	modem, CD-ROM
transfer schedule	synchronous, asynchronous	tape, keyboard
sharing	dedicated, sharable	tape, keyboard
device speed	latency, seek time, transfer rate, delay between operations	
I/O direction	read only, write only, read-write	CD-ROM, graphics controller, disk

  - Data representation; Encoding schemes
  - Error conditions; Devices respond to errors differently

- Blocking; process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- Nonblocking; I/O call returns as much as available
  - user interface, data copy (buffered I/O)
  - implemented via multi-threading code for I/O call
  - returns quickly with count of bytes transferred
- Asynchronous; process runs while I/O executes
  - difficult to use
  - I/O subsystem signals process when I/O completed, e.g., call-backs: pointer to completion code

1.1.4 Evolution of the I/O Function (see Fig. 6)

- Processor directly controls a peripheral device. Example: CPU controls a flip-flop to implement a serial line
- Controller or I/O module is added
  - Processor uses programmed I/O without interrupts

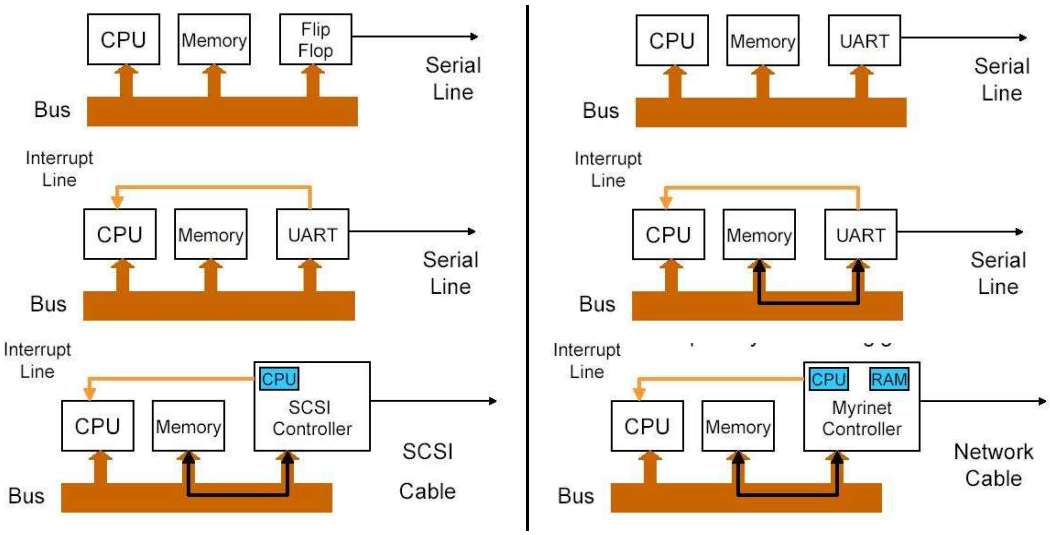

Figure 6: Evolution of the I/O Function

  - Processor does not need to handle details of external devices
  - Example: A Universal Asynchronous Receiver Transmitter
    * CPU simply reads and writes bytes to I/O controller
    * I/O controller responsible for managing the signalling

- Controller or I/O module with interrupts. Processor does not spend time waiting for an I/O operation to be performed

- Direct Memory Access

  - Blocks of data are moved into memory without involving the processor
  - Processor involved at beginning and end only

- I/O module has a separate processor. Example: SCSI controller, controller CPU executes SCSI program code out of main memory

- I/O processor

  - I/O module has its own local memory, internal bus, etc.
  - It is a computer in its own right.
  - Example: Myrinet Multi-gigabit Network Controller

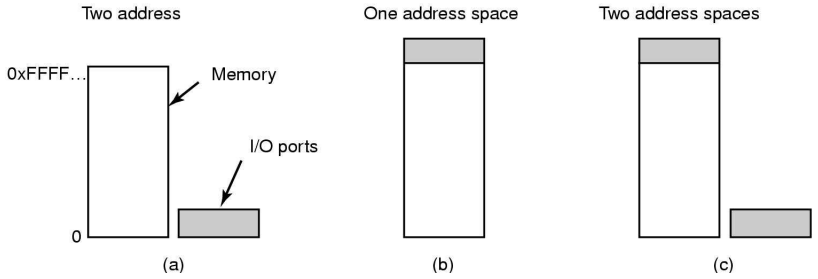

Figure 7: a) Separate I/O and memory space. b) Memory-mapped I/O. c) Hybrid.

1.1.5 Memory-Mapped I/O (see Fig. 7)

- Separate I/O and memory space
  - I/O controller registers appear as I/O ports
  - Accessed with special I/O instructions
- Memory-mapped I/O
  - Controller registers appear as memory
  - Use normal load/store instructions to access
- Hybrid; x86 has both ports and memory mapped I/O
- Bus Architectures (see Fig. 8)

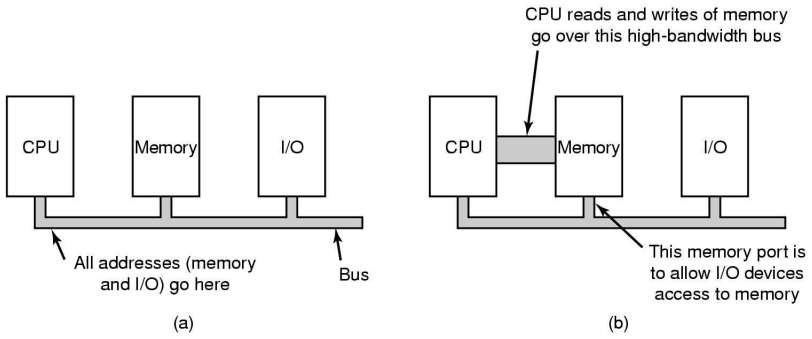

Figure 8: a) A single-bus architecture. b) A dual-bus memory architecture.

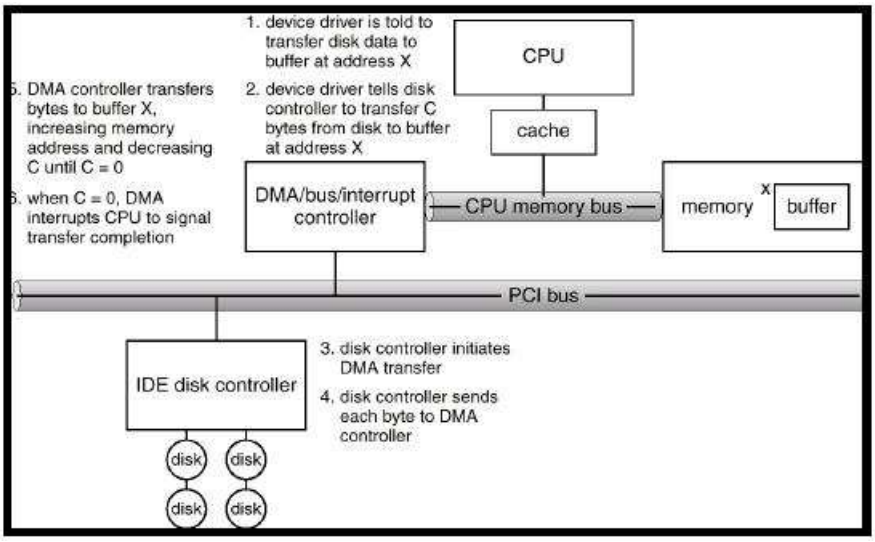

Figure 9: The Process to Perform DMA Transfer.

- A single-bus architecture; if the computer has a single bus, having everyone look at every address is straightforward
- A dual-bus memory architecture; the trend in modern personal computers is to have a dedicated high–speed memory bus. This bus is tailored for optimize memory performance, with no compromises for the sake of slow I/O devices. Pentium systems even have three external buses (memory, PCI, ISA)

1.1.6 Direct Memory Access (DMA)

- Takes control of the bus from the CPU to transfer data to and from memory over the system bus
- Cycle stealing is used to transfer data on the system bus
- The instruction cycle is suspended so data can be transferred
- The CPU pauses one bus cycle, CPU Cache can hopefully avoid such pauses
- Reduced number of interrupts occur, No expensive context switches
- Cycle stealing causes the CPU to execute more slowly; Still more efficient than CPU doing transfer itself

- The CPU cache can hide some bus transactions
- Number of required busy cycles can be cut by
  - integrating the DMA and I/O functions
  - Path between DMA module and I/O module that does not include the system bus
- The Process to Perform DMA Transfer (see Fig. 9)
  1. device driver is told to transfer disk data to buffer at address X
  2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
  3. disk controller initiates DMA transfer
  4. disk controller sends each byte to DMA controller
  5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C=0
  6. when C=0, DMA interrupts CPU to signal transfer completion

1.1.7 Interrupts Revisited (see Fig. 10)

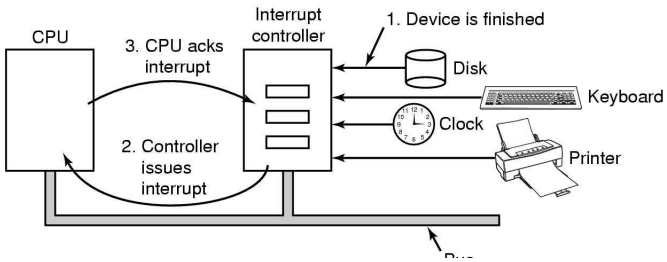

Figure 10: How interrupts happen. The connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

- CPU interrupt request line triggered by I/O device
- Interrupt handler receives interrupts
- Maskable to ignore or delay some interrupts

- Interrupt vector to dispatch interrupt to correct handler based on priority some unmaskable
- Interrupt mechanism also used for exceptions, traps

1.2 Principles of I/O Software

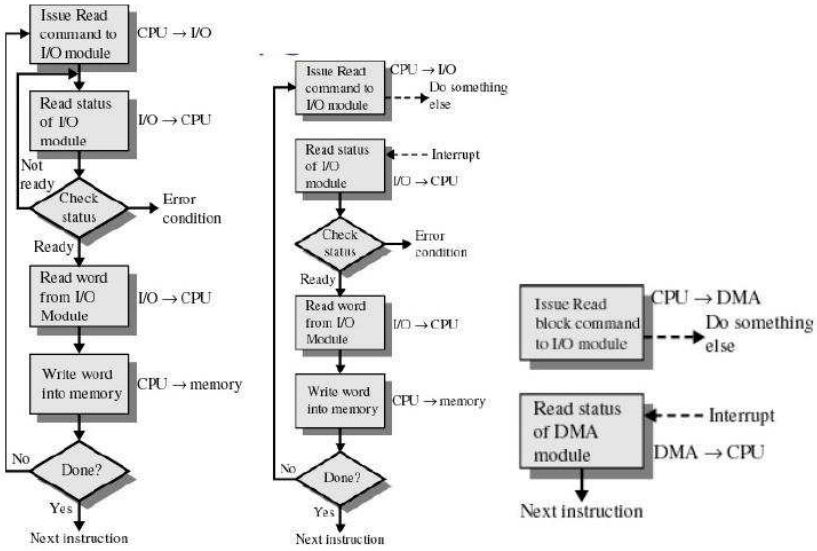

Figure 11: a) Programmed I/O. b) Interrupt-Driven I/O. c) Direct Memory Access.

1.2.1 Programmed I/O (see Fig. 11a)

- Also called *polling*, or *busy waiting*
- I/O module (controller) performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete; Wastes CPU cycles

1.2.2 Interrupt-Driven I/O (see Fig. 11b)

- Processor is interrupted when I/O module (controller) ready to exchange data
- Processor is free to do other work
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor

1.2.3 Direct Memory Access (see Fig. 11c)

- Transfers a block of data directly to or from memory
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer