LESSON 1

Intro to systems programing

1.1. What is Systems Programming?

Computer programming can be categorized into two categories .i.e

1. Input
2. output

While designing software the programmer may determine the required inputs for that program, the wanted outputs and the processing the software would perform in order to give those wanted outputs. The implementation of the processing part is associated with application programming. Application programming facilitates the implementation of the required processing that software is supposed to perform; everything that is left now is facilitated by system programming. Systems programming is the study of techniques that facilitate the acquisition of data from input devices, these techniques also facilitates the output of data which may be the result of processing performed by an application.

1.2. Three Layered Approach

A system programmer may use a three layered approach for systems programming. As you can see in the figure the user may directly access the programmable hardware in order to perform I/O operations. The user may use the trivial BIOS (Basic Input Output System) routines in order to perform I/O in which case the programmer need not know the internal working of the hardware and need only the knowledge BIOS routines and their parameters. In this case the BIOS programs the hardware for required I/O operation which is hidden to the user. In the third case the programmer may invoke operating systems (DOS or whatever) routines in order to perform I/O operations. The operating system in turn will use BIOS routines or may program the hardware directly in order to perform the operation.

1.2.1. Methods of I/O

In the three layered approach if we are following the first approach we need to program the hardware. The hardware can be programmed to perform I/O in three ways i.e

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access

In case of programmed I/O the CPU continuously checks the I/O device if the I/O operation can be performed or not. If the I/O operations can be performed the CPU performs the computations required to complete the I/O operation and then again starts waiting for the I/O device to be able to perform next I/O operation. In this way the CPU remains tied up and is not doing anything else besides waiting for the I/O device to be idle and performing computations only for the slower I/O device.
In case of interrupt driven the flaws of programmed driven I/O are rectified. The processor does not check the I/O device for the capability of performing I/O operation rather the I/O device informs the CPU that it's idle and it can perform I/O operation, as a result the execution of CPU is interrupted and an Interrupt Service Routine (ISR) is invoked which performs the computations required for I/O operation. After the execution of ISR the CPU continues with whatever it was doing before the interruption for I/O operation. In this way the CPU does not remain tied up and can perform computations for other processes while the I/O devices are busy performing I/O and hence is more optimal. Usually it takes two bus cycles to transfer data from some I/O port to memory or vice versa if this is done via some processor register. This transfer time can be reduced bypassing the CPU as ports and memory device are also interconnected by system bus. This is done with the support of DMA controller. The DMA (direct memory access) controller can controller the buses and hence the CPU can be bypassed data item can be transferred from memory to ports or vice versa in a single bus cycle.

1.2.2. I/O controllers

No I/O device is directly connected to the CPU.
To provide control signals to the I/O device a I/O controller is required. I/O controller is located between the CPU and the I/O device. For example the monitor is not directly collected to the CPU rather the monitor is connected to a VGA card and this VGA card is in turn connected to the CPU through busses.
The keyboard is not directly connected to CPU rather its connected to a keyboard controller and the keyboard controller is connected to the CPU.

The function of this I/O controller is to provide

- I/O control signals
- Buffering
- Error Correction and Detection

We shall discuss various such I/O controllers interfaced with CPU and also the techniques and rules by which they can be programmed to perform the required I/O operation.
Some of such controllers are

•.1. • DMA controller

•.2. • Interrupt controller

•.3. • Programmable Peripheral Interface (PPI) • Interval Timer

•.4. • Universal Asynchronous Receiver Transmitter

•.5. We shall discuss all of them in detail and how they can be used to perform I/O operations.

1.2.3. Operating systems

Systems programming is not just the study of programmable hardware devices. To develop effective system software one needs to the internals of the operating system as well. Operating systems make use of some data structures or tables for management of computer resources. We will take up different functions of the operating systems and discuss how they are performed and how can the data structures used for these operations be accessed.
File management is an important function of the operating systems. DOS/Windows uses various data structures for this purpose. We will see how it performs I/O management and how the data structures used for this purpose can be directly accessed. The various data structures are popularly known as FAT which can be of 12, 16 and 32 bit wide, Other data structures include BPB(BIOS parameter block), DPB(

drive parameter block) and the FCBs(file control block) which collectively forms the directory structure.
To understand the file structure the basic requirement is the understanding of the disk architecture, the disk formatting process and how this process divides the disk into sectors and clusters.

1.2.4. Memory management

Memory management is another important aspect of operating systems.
Standard PC operate in two mode in terms of memory which are

- Real Mode
- Protected Mode

In real mode the processor can access only first one MB of memory to control the memory within this range the DOS operating system makes use of some data structures called

- FCB (File control block)
- PSP (Program segment prefix)

We shall discuss how these data structures can be directly accessed, what is the significance of data in these data structures. This information can be used to traverse through the memory occupied by the processes and also calculate the total amount of free memory available. Certain operating systems operate in protected mode. In protected mode all of the memory interfaced with the processor can be accessed.
Operating systems in this mode make use of various data structures for memory management which are

- Local Descriptor Table
- Global Descriptor Table
- Interrupt Descriptor Table

We will discuss the significance these data structures and the information stored in them. Also we will see how the logical addresses can be translated into physical addresses using the information these tables

1.2.5. Viruses and Vaccines

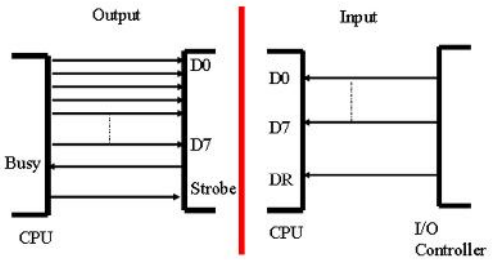
Once an understanding of the file system and the memory Management is developed it is possible to understand the working of viruses. Virus is a simple program which can embed itself within the computer resources and propagate itself. Mostly viruses when activated would perform something hazardous.
We will see where do they embed themselves and how can they be detected. Moreover we will discuss techniques of how they can be removed and mostly importantly prevented to perform any infections. There are various types of viruses but we will discuss those which embed themselves within the program or executable code which are

1. Executable file viruses
2. Partition Table or boot sector viruses

1.2.6. Device Drivers

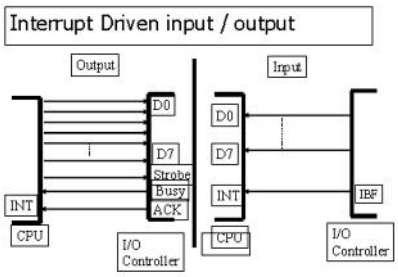
Just connecting a device to the PC will not make it work unless its device drivers are not installed. This is so important because a device driver contains the routines which perform I/O operations on the device. Unless these routines are provided no I/O operation on the I/O device can be performed by any application. We will discuss the integrated environment for the development of device drivers for DOS and Windows. We shall begin our discussion from means of I/O. On a well designed device it is possible to perform I/O operations from three different methods

- Programmed I/O
- Interrupt driven I/O
- DMA driven I/O



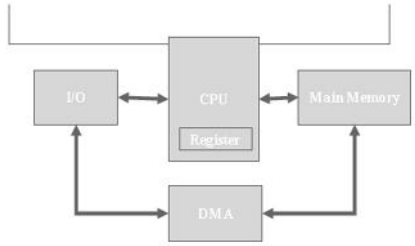
In case of programmed I/O the CPU is in a constant loop checking for an I/O opportunity and when its available it performs the computations operations required for the I/O operations. As the I/O devices are generally slower than the CPU, CPU has to wait for I/O operation to complete so that next data item can be sent to the device. The CPU sends data on the data lines. The device need to be signaled that the data has been sent this is done with the help of STROBE signal. An electrical pulse is sent to the device by turning this signal to 0 and then 1. The device on getting the strobe signal receives the data and starts its output. While the device is performing the output it's busy and cannot accept any further data on the other and CPU is a lot faster device and can process lot more bytes during the output of previously sent data so it should be synchronized with the slower I/O device.
This is usually done by another feed back signal of BUSY which is kept active as long as the device is busy. So the CPU is only waiting for the device to get idle by checking the BUSY signal as long as the device is busy and when the device gets idle the CPU will compute the next data item and send it to the device for I/O operation. Similar is the case of input, the CPU has to check the DR (data Ready) signal to see if data is available for input and when its not CPU is busy waiting for it.

Interrupt Driven I/O



The main disadvantage of programmed I/O as can be noticed is that the CPU is busy waiting for an I/O opportunity and as a result remain tied up for that I/O operation. This disadvantage can be overcome by means of interrupt driven I/O. In Programmed I/O CPU itself checks for an I/O opportunity but in case of interrupt driven I/O the I/O controller interrupts the execution of CPU when ever and I/O operation is required for the computation of the required I/O operation. This way the CPU can perform other computation and interrupted to perform and interrupt service routine only when an I/O operation is required, which is quite an optimal technique.

DMA driven I/O



In case data is needed to transferred from main memory to I/O port this can be done using CPU which will consume 2 bus cyclesfor a single word, one bus cycle from memory to CPU and other from CPU to I/O port in case of output and the vice versa in case of input. In case no computation on data is required CPU can be bypassed and another device DMA (direct memory access) controller can be used. Its possible to transfer a data word directly from memory to CPU and vice versa in a single bus cycle using the DMA, this technique is definitely faster.

Revision questions

Example ✎**.** Outline any three methods that you can use to perform I/O

Solution:

• Programmed I/O
• Interrupt driven I/O
• DMA driven I/O

□

EXERCISE 1. ✍ ...

References and Additional Reading Materials

1. Recommended: An Introduction to Programming and Object Oriented Design Using Java, Jaime
2. Nino and Frederick A. Hosch, John Wiley, New York.
3. Flanagan, D. (2005). Java in a nutshell : a desktop quick reference. O'Reilly (5th ed.).
4. Flanagan, D. (2004). Java examples in a nutshell : a tutorial companion to Java in a nutshell. O'Reilly (3rd ed.)

Course Journals

1. Acta Informatica ISSN 0001-5903
2. Advances in Computational Mathematics ISSN 1019-7168
3. Advances in data Analysis and Classification ISSN1 1862-5347
4. Annals Of software Engineering ISSN 1022-7091

Reference Journals

1. Journal of computer science and Technology ISSN 1000-9000
2. Journal of Science and Technology ISSN 1860-4749
3. Central European Journal Of Computer Science ISSN 1896-1533 Cluster computing ISSN 1386-7857

LESSON 2

Interupt mechanisms

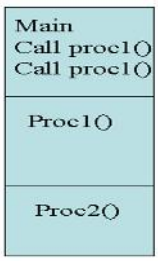
2.1. introduction

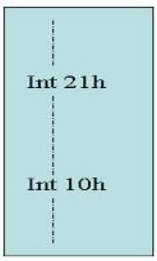
Interrupt follow a follow a certain mechanism for their invocation just like near or far procedures. To understand this mechanism we need to understand its differences with procedure calls. Difference between interrupt and procedure calls Procedures or functions of sub-routines in various different languages are called by different methods as can be seen in the examples.

- Call MyProc
- A= Addition(4,5);
- Printf("hello world");

The general concept for procedure call in most of the programming languages is that on invocation of the procedure the parameter list and the return address (which is the value if IP register in case of near or the value of CS and IP registers in case of far procedure) is pushed Moreover in various programming languages whenever a procedure is called its address need to be specified by some notation i.e. in C language the name of the procedure is specified to call a procedure which effectively can be used as its address.
However in case of interrupts the a number is used to specify the interrupt number in the call

- Int 21h
- Int 10h
- Int3



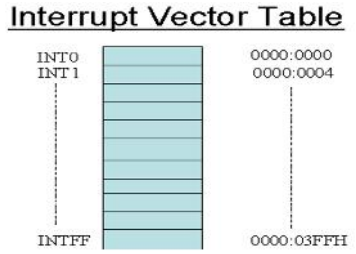


Moreover when an interrupt is invoked three registers are pushed as the return address i.e. the values of IP, CS and Flags in the described order which are restored on return. Also no parameters are pushed onto the stack on invocation parameters can only be passed through registers.

2.1.1. The interrupt vector table

The interrupt number specified in the interrupt call is used as an index into the interrupt vector table. Interrupt vector table is a global table situated at the address 0000:0000H. The size of interrupt vector table is 1024 bytes or 1 KB. Each entry in the IVT is sized 4 bytes hence 256 interrupt vectors are possible numbered (0-FFH). Each entry in the table contains a far address of an interrupt handlers hence there is a maximum of 256 handlers however each handlers can have a number of services within itself. So the number operations that can be performed by calling an interrupt service routine (ISR) is indefinite depending upon the nature of the operating system. Each vector contains a far address of an interrupt handler. The address of the vector and not the address of interrupt handler can be easily calculated if the interrupt number is known. The segment address of the whole IVT is 0000H the offset address for a particular interrupt handler can be determined by multiplying its number with 4 eg. The offset address of the vector of INT 21H will be 21H * 4 = 84H and the segment for all vectors is 0 hence its far address is 0000:0084H,(this is the far address of the interrupt vector and not the interrupt service routine or interrupt handler). The vector in turn contains the address of the interrupt service routine which is an arbitrary value depending upon the location of the ISR residing in memory.

Fig 2 (Interrupt Vector Table)



Moreover it is important to understand the meaning of the four bytes within the interrupt vector. Each entry within the IVT contain a far address the first two bytes (lower word) of which is the offset and the next two bytes (higher word) is the segment address.

Generally there are three kind of ISR within a system depending upon the entity which implements it

- BIOS (Basic I/O services) ISRs
- DOS ISRs
- ISRs provided by third party device drivers

When the system has booted up and the applications can be run all these kind of ISRs maybe provided by the system. Those provided by the ROM-BIOS would be typically resident at any location after the address F000:0000H because this the address within memory from where the ROM-BIOS starts, the ISRs provided by DOS would be resident in the DOS kernel (mainly IO.SYS and MSDOS.SYS loaded in memory) and the ISR provided by third party device drivers will be resident in the memory occupied by the device drivers.

2.1.2. Interrupt Invocation

Although hardware and software interrupts are invoked differently i.e hardware interrupts are invoked by means of some hardware whereas software interrupts are invoked by means of software instruction or statement but no matter how an interrupt has been invoked processor follows a certain set steps after invocation of interrupts in exactly same way in both the cases.
These steps are listed as below

- Push Flags, CS, IP Registers, Clear Interrupt Flag
- Use (INT#)*4 as Offset and Zero as Segment

This is the address of interrupt Vector and not the ISR

- Use lower two bytes of interrupt Vector as offset and move into IP
- Use the higher two bytes of Vector as Segment Address and move it into CS=0:[offset+2]
- Branch to ISR and Perform I/O Operation
- Return to Point of Interruption by Popping the 6 bytes i.e. Flags CS, IP.

Moreover the instruction INT 21H can be assembled and executed in the debug program, on doing exactly so the instruction is traced through and the result is monitored. It can be seen that on execution of this instruction the value of IP is changed to 107CH and the value of CS is changed to 00A7H which cause the execution to branch to the Interrupt # 21H in memory and the previous values of flags, CS and IP registers are temporarily saved onto the stack as the value of SP is reduced by 6 and the dump at the location SS:SP will show these saved values as well.

2.1.3. Parameter passing into Software interrupts

In case of procedures or function in various programming languages parameters are passed through stack. Interrupts are also kind of function provided by the operating system but they do not accept parameters by stack rather they need to passed parameters through registers.

2.1.4. Software interrupts invocation

Now let's see how various interrupts can be invoked by means of software statements. First there should be way to pass parameters into a software interrupt before invoking the interrupt; there are several methods for doing this. One of the methods is the use of pseudo variables. A variable can be defined a space within the memory whose value can be changed during the execution of a program but a pseudo variable acts very much like a variable as its value can be changed anywhere in the program but is not a true variable as it is not stored in memory. C programming language provides the use of pseudo variables to access various registers within the processor.

The are various registers like AX, BX, CX and DX within the processor they can be directly accessed in a program by using their respective pseudo variable by just attaching a "_" (underscore) before the register's name eg. _AX = 5; A = _BX .

After passing the appropriate parameters the interrupt can be directly invoked by calling the *geninterrupt ()* function. The interrupt number needs to be passed as parameter into the *geninterrupt()* function.

Interrupt # 21H, Service # 09 description

Now lets learn by means of an example how this can be accomplished. Before invoking the interrupt the programmer needs to know how the interrupt behaves and what parameters it requires. Lets take the example of interrupt # 21H and

service # 09 written as 21H/09H in short. It is used to print a string ending by a '$' character and other parameters describing the string are as below
Inputs AH = 0x09
DS = Segment Address of string
DX = Offset Address of string
Output
The '$' terminated string at the address DS:DX is displayed
One thing is note worthy that the service # is placed in AH which is common with almost all the interrupts and its service. Also this service is not returning any significant data, if some service needs to return some data it too is received in registers depending upon the particular interrupt.
Example:

```
#include<stdio.h>
#include<BIOS.H>
#include<DOS.H>
#include<conio.h>
char st[80] ={"Hello World$"};
void main()
{
clrscr(); //to clear the screen contents
_DX = (unsigned int) st;
_AH = 0x09;
geninterrupt(0x21);
getch(); //waits for the user to press any key
}
```

this is a simple example in which the parameters of int 21H/09H are loaded and then int 21H is invoked. DX and AH registers are accessed through pseudo variables and then *geninterrupt()*is called to invoke the ISR. Also note that _DS is not loaded. This is the case as the string to be loaded is of global scope and the C language compiler automatically loads the segment address of the global data into the DS register.
Another Method for invoking software interrupts
This method makes use of a Union. This union is formed by two structure which correspond to general purpose registers AX, BX, CX and DX. And also the half

register AH, AL, BH, BL, CH, CL, DH, DL. These structures are combined such that through this structure the field ax can be accessed to load a value and also its half components al and ah can be accessed individually. The declaration of this structure goes as below. If this union is to be used a programmer need not declare the following declaration rather declaration already available through its header file "dos.h"

```
struct full
{
unsigned int ax;
unsigned int bx;
unsigned int cx;
unsigned int dx;
};
struct half
{
unsigned char al;
unsigned char ah;
unsigned char bl;
unsigned char bh;
unsigned char cl;
unsigned char ch;
unsigned char dl; unsigned char dh;
};
typedef union tagREGS
{
struct full x;
struct half h;
}REGS;
```

This union can be used to signify any of the full or half general purpose register shows if the field ax in x struct is to be accessed then accessing the fields al and ah in h will also have the same effect as show in the example below.

Example:

```
#include<DOS.H>
union REGS regs;
```

```
void main (void )
{
regs.h.al = 0x55;
regs.h.ah = 0x99;
printf ("%x",regs.x.ax); }
```

The int86() function

The significance of this REGS union can only be understood after understanding the int86() function. The int86() has three parameters. The first parameter is the interrupt number to be invoked, the second parameter is the reference to a REGS type union which contains the value of parameters that should be passed as inputs, and third parameter is a reference to a REGS union which will contain the value of registers returned by this function. All the required parameters for an ISR are placed in REGS type of union and its reference is passed to an int86() function. This function will put the value in this union into the respective register and then invoke the interrupt. As the ISR returns it might leave some meaningful value in the register (ISR will return values), these values can be retrieved from the REGS union whose reference was passed into the function as the third parameter.

Example using interrupt # 21H service # 42H

To make it more meaningful we can again elaborate it by means of an example. Here we make use of ISR 21H/42H which is used to move the file pointer. Its detail is as follows

Int # 21 Service # 42H

Inputs

AL = Move Technique

BX = File Handle

CX-DX = No of Bytes File to be moved

AH = Service # = 42H

Output

DX-AX = No of Bytes File pointer actually moved

This service is used to move the file pointer to a certain position relative to a certain point. The value in AL specify the point relative to which the pointer is moved. If the value of AL = 0 then file pointer is moved relative to the BOF (begin of File) if AL=1 then its moved relative to current position and if AL = 2 then its moved relative to the EOF (end of file).

CX-DX specify the number of bytes to move a double word is needed to specify this value as the size of file in DOS can be up to 2 GB.
On return of the service DX-AX will contain the number of bytes the file pointer is actually moved eg. If the file pointer is moved relative to the EOF zero bytes the DX-AX on return will contain the size of file if the file pointer was at BOF before calling the service.

Revision questions

Example . Outline any three ISR that can be used in systems

Solution:

BIOS (Basic I/O services) ISRs
DOS ISRs
ISRs provided by third party device drivers

□

EXERCISE 2. ....