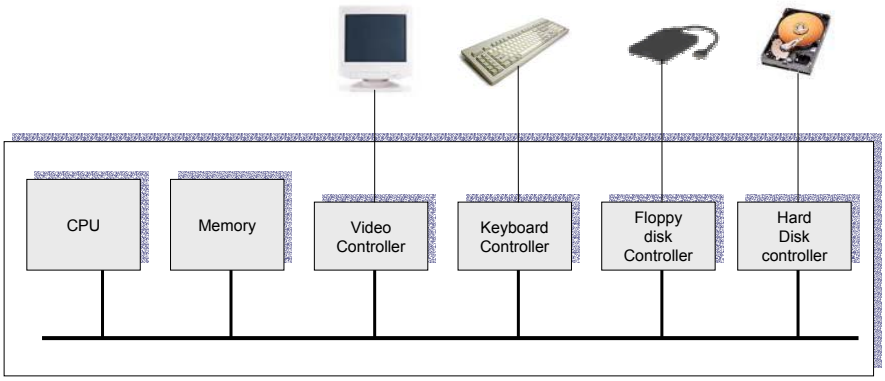
Computer Hardware Review

CS 342 – Operating Systems

Ibrahim Korpeoglu
Bilkent University
Computer Engineering Department

OS and Hardware



Hardware Components

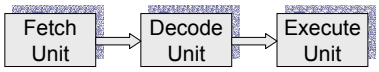
- Processors
- Memory
- I/O Devices
- Buses

Processor (CPU)

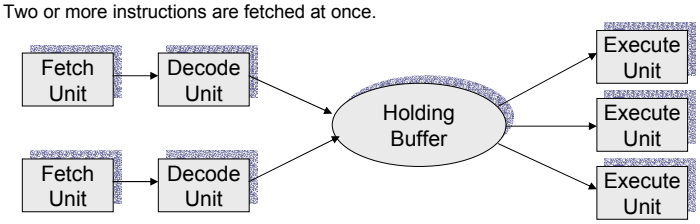
- It is the brain of the computer
  - Fetches instructions from main memory and executes them
    - Fetch, Decode, Execute cycles for each instruction.
- Each CPU has a specific set of instructions
  - Pentium and SPARC has different instruction set.
    - Therefore a program compiled for Pentium can not be run on a Sparc.
- Each CPU has also some set of registers.
  - There are
    - General purpose registers: hold varibles and temporary results
    - Special registers:
      - Program counter (PC)
      - Stack pointer (SP)
      - Program Status Word (PSW)
        - Condition code bits, CPU priority, mode (kernel or user), other control bits.

Processor (CPU)

- Mode advanced CPU use pipelining:
  - More than one instruction is executed in the CPU by use of a technique called pipelining
  - There are different type of units that processes instructions: fetch, decode and execute
  - While an instruction is in the fetch unit, an other instruction is in the decode unit, and an other one is in the execute unit.



- Some more advanced technique is superscalar CPI:
  - Multiple execution units are present: one for integer arithmetic, one for floating-point arithmetic, etc.
  - Two or more instructions are fetched at once.



Processor (CPU)

- Most CPUs have two modes: kernel mode, user mode.
  - A bit in PSW register control the mode of the program that is running in the CPU
  - When in kernel mode:
    - a CPU can execute every instruction in the instruction set.
    - A CPU can use very feature of the hardware,
  - When in user mode:
    - a CPU can execute a subset of the instructions
    - A CPU can use a subset of the features of the hardware.

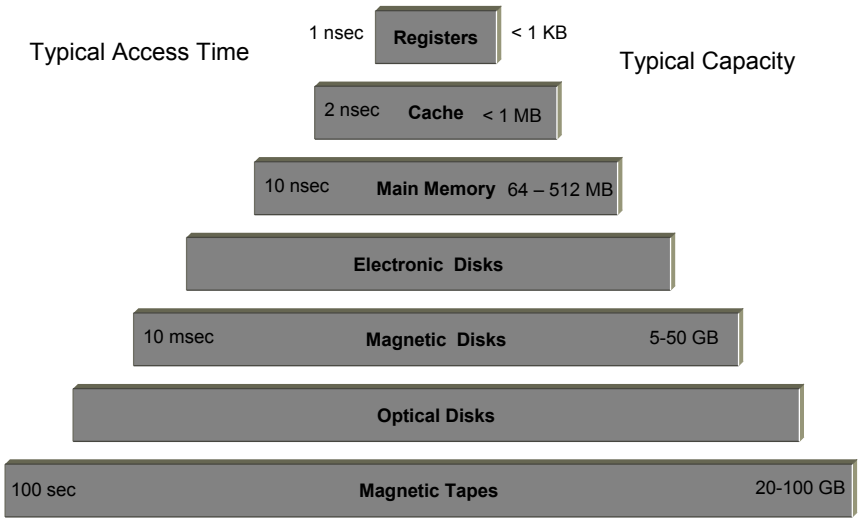
Processor (CPU) – System Call

- To obtain some service from OS, a user program makes a system call, which traps into the kernel and starts the OS.
  - The TRAP instruction switches from user mode to kernel mode and starts the operating system.
  - When the OS work is complete, the control is returned back to the running program to instruction following the TRAP instruction.

Memory

- Memory is used to store the instructions and data while a program is executing.
- Memory design objectives:
  - access speed (nanosecond)
  - Capacity (MB)
  - Cost ($)
- No memory technology satisfies all these objectives. Therefore there is a hierarchy of memory used by the system

Memory - Hierarchy



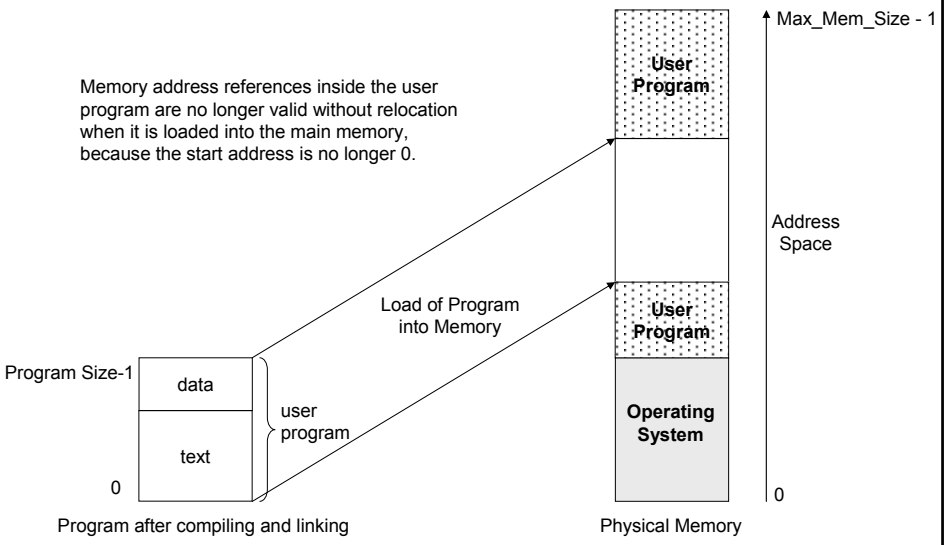
Memory – Hierarchy Member Features

Registers	Same material with CPU, located inside CPU, as fast as CPU, capacity= 32x32bits in 32-CPUs, 64x64bits in 64-bit CPUs. User program itself manages the registers, volatile storage
Cache	Mostly controlled by hardware, access time in the order of 2 clock cycles. Limited size due to high cost, volatile storage
Main Memory	Semiconductor material, volatile storage, random access, directly addressable by CPU
Electronic Disk	Can be designed to be either volatile or non-volatile. During normal operation, data is stored in a large DRAM array; a battery enables the data in DRAM to is copied to a hidden magnetic disk when external power is cut off.
Magnetic Disk	Two orders of magnitude cheaper than RAM, one or more metal platters rotating at 5400, 7200, or 10800 rpm. A platter consists of tracks and sectors. A sector is 512 bytes. Data transfer rate changes from 5MB/s to 160MB/s.
Magnetic Tape	Used for backup and large data-set storage. Sequential access. Very cheap.

Main Memory

- For unitask systems, the whole memory can be dedicated to a single program and OS.
- For multiprogramming and time-sharing systems, the memory is allocated to multiple processes (executing programs).
  - It increases CPU utilization
  - It decreases response time for interactive users.
- This brings two main problems:
  - How to protect programs from one an other and the kernel from them all
  - How to handle relocation

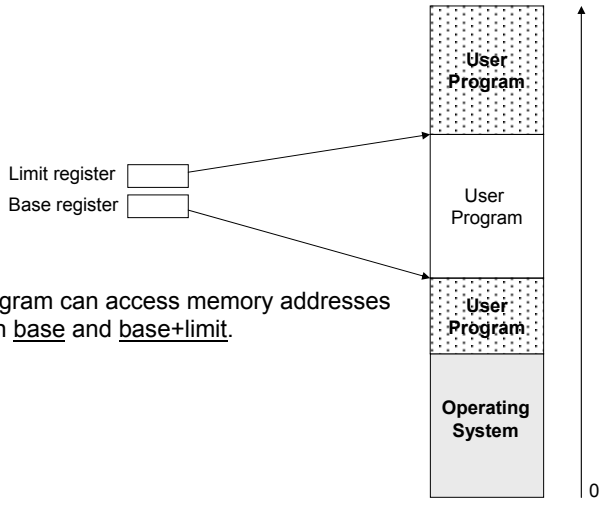
Main Memory – Relocation Problem



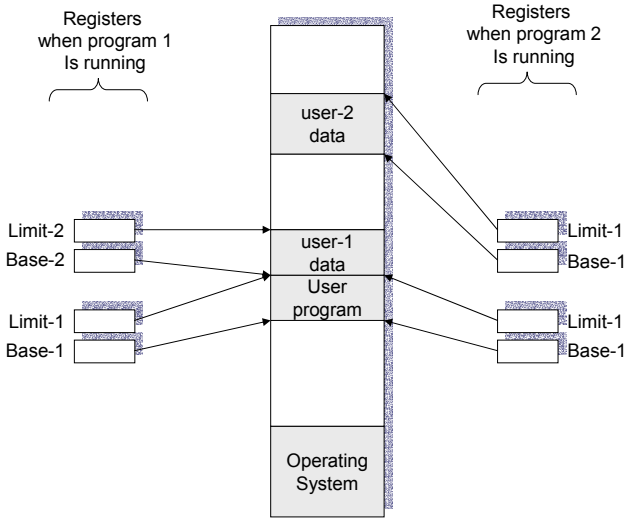
Solving Relocation Problem

- Two methods
  - 1) Before loading, after deciding where to load the user program, find all memory references inside the user program and update them with the new address.
  - 2) Use two registers for the running program: base and limit registers
    - Base register defines the start of the program in physical memory
    - Limit register defines the length (or size) of the program (program code –text- plus program data).

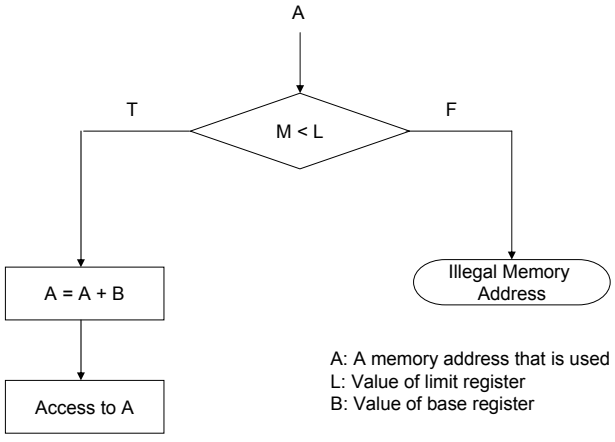
Solving Relocation Problem



The program can access memory addresses between base and base+limit.



Use of base and limit registers



A: A memory address that is used by the user program
L: Value of limit register
B: Value of base register

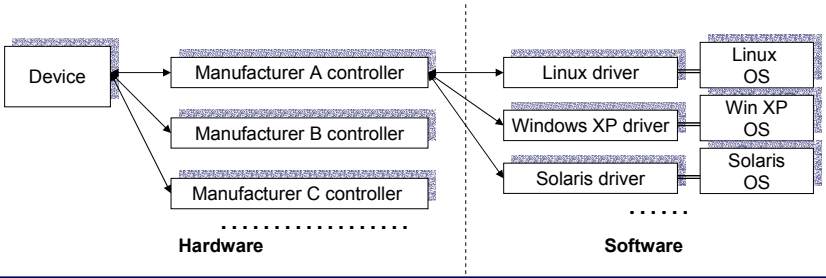
This operation is done by a device called Memory Management Unit (MMU). It sits logically between CPU and Memory. It can be physically located at the CPU chip.

I/O Devices

- OS must also manage I/O devices.
- User programs can not access I/O devices directly.
- An I/O device usually consists of two parts:
  - A device controller
    - A chip or set of chips that controls the device. Usually has a microcontroller in it that runs independent of CPU and that is programmed to control the device.
  - The device itself.
- Examples:
  - A graphics card and a monitor.
  - A hard disk controller and hard disk (drive) itself.
  - A floppy disk controller and a floppy disk drive itself.
  - A printer controller and printer itself.
  - A keyboard controller and keyboard itself.
  - …..
- A device controller is also called a card or an adapter.
- Some controllers may not have an associated mechanical device (a network card for example).

I/O Devices

- An OS deals (interacts) with device controller. A device controller deals with the actual device.
- An OS usually does not interact with the device itself directly.
- The part of OS that interacts with a device controller is called a device driver.
- Every different device controller has a different device driver.
- A device controller will have different device drivers for each operating system that device is supported.
- For same type of device, depending on the manufacturer, we may need a different device driver, although we are using the same operating system.



I/O Devices and drivers

- A driver is part of operating system that interacts with the device
  - Therefore a device deriver is hardware dependent
- A device deriver should be put (integrated) with the OS
  - It should run in kernel mode.
- There are 3 ways to put a driver into an OS
  - 1) relink the OS kernel with the driver and reboot system
    - Most Unix system work in this way.
  - 2) make an entry in an OS configuration file telling it needs the diver for talking to device. When OS boots up, it look to the entry and load the driver.
    - Windows uses this method.
  - 3) Load and accept new drivers while the kernel is running. Enables hot plugging. Called dynamic loading. No need for rebooting.
    - USB needs dynamic loading.

I/O Device Controller

- Every controller has
  - Some small number registers that are used to communicate with it.
    - A device driver for a controller can write and read these registers.
  - Some amount of buffer.
- There are two ways to access these registers and buffers in device controller.
  - 1) map the registers into the address space of OS
    - Consumes from OS address space
    - No need for special I/O instructions
  - 2) put the device registers into a special I/O port space.
    - Each register has a port address
    - Does not consume from OS address space
    - Needs special I/O instructions to read and write to these locations.
      - Intel machines use this scheme.

Example

- Intel IA-32 Architecture supports I/O port address space
  - There are two instructions to move data between controllers and CPU
    - IN: move data from port to register. The register in CPU is EAX (32 bit), AX (16 bit), or AL (8 bit)
      - INS: moves strings
    - OUT: move data from register to port
      - OUTS: moves strings
  - I/O port address could be:
    - An immediate operand, or
    - A value stored in DX register
- The I/O ports (registers on controllers) are accessed through a separate I/O address space
  - İt is distinct from physical memory address space
  - Address space consists of 2^{16} (64K) individually addressable 8-bit (1 byte) I/O ports: from 0 to 0xFFFF
- IA-32 also supports memory-mapped I/O

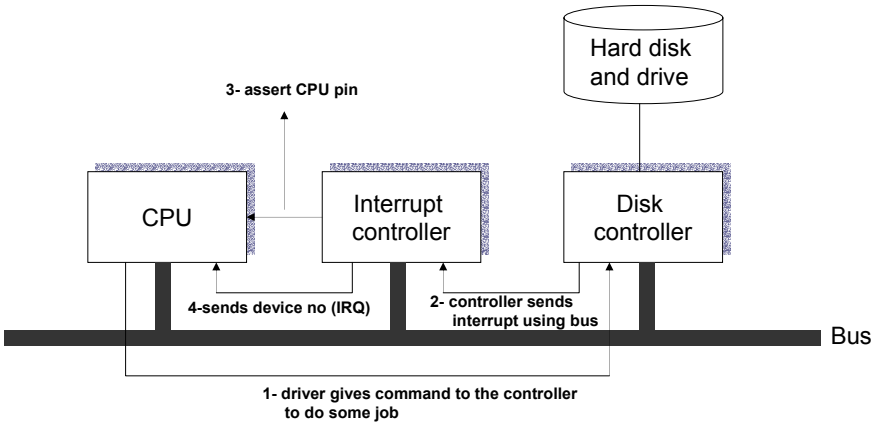
Seperate I/O port address space versus memory mapped I/O

- In memory mapped I/O, ınstructions to move data between memory and CPU can also be used to move data between controller registers and CPU
  - Such an instrunction is MOV instruction
    - MOV DST, SRC.
    - ....
- If a separate I/O port address space is used, then MOV instruction can not be used.
  - IN and OUT instructions are used instead.
    - IN AL, 0x03F8
    - IN AX, 0x03F8
    - IN EAX, 0x03F8
    - IN AL, DX
    - ....
    - OUT 0x03F8, AL
    - OUT 0x03F8, AX
    - OUT 0x03F8, EAX
    - OUT DX, AL,
    - .....

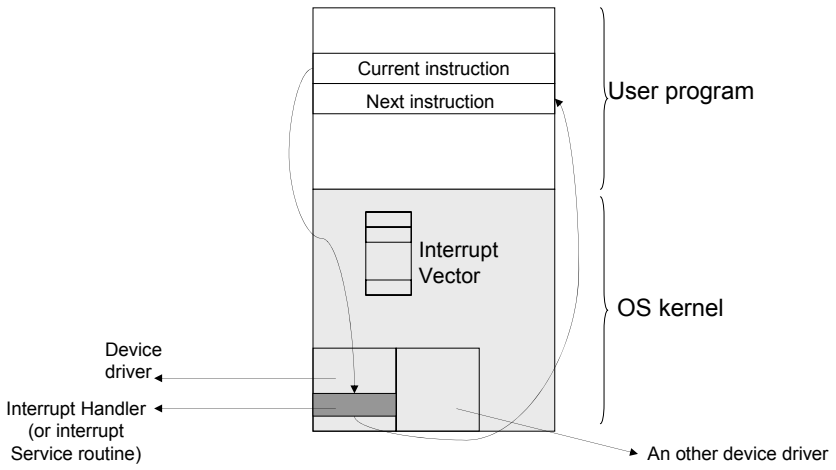
I/O methods

- Three ways
  - Polling (busy waiting)
    - Driver polls the controller until data is available or operation is complete
    - Waste of CPU with useless polling (busy waiting)
  - Interrupt driven
    - Driver starts I/O by giving commands to the controller
    - Process is blocked
    - CPU is given to an other process
    - Controller does the job independent of CPU and when finished, gives an interrupt to the CPU.
    - Interrupt service routine of driver takes the data from controller and copies it to the memort
    - The process that was blocked can now continue to run
  - Direct Memory Access
    - While a driver is copying data, CPU will be busy with the copy operation.
    - A special chip, called DMA controller (Direct memory access) does the copy of data from controller to memory (or vice versa) when the controller is finished accessing the device. DMA chip then gives and interrupt to indicate that copy operation is finished.
    - In this was CPU cycles are not wasted with copy operation. CPU can do some other useful work, while DMA is doing copying.

Interrupt driven I/O model



Interrupt driven I/O model

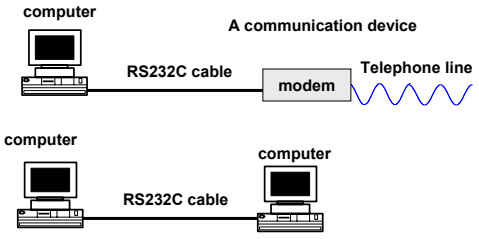


Example device and controller: serial port and UART

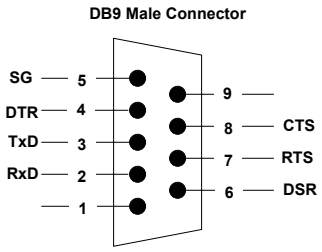
Serial ports on computers enables a computer to communicate with Other computers and device using serial cables.

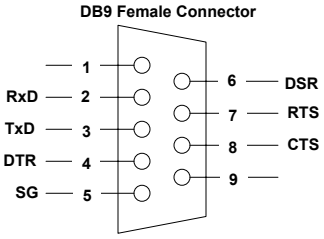
- a computer can be connected to an other computer using serial cable (also called RS-232 cable)
- a computer can be connected to an external modem
- a computer can be connected to a digital camera that also has serial port…..

The standard that is used for serial communication is called RS-232 standard



RS232 Connectors – 9 Pin

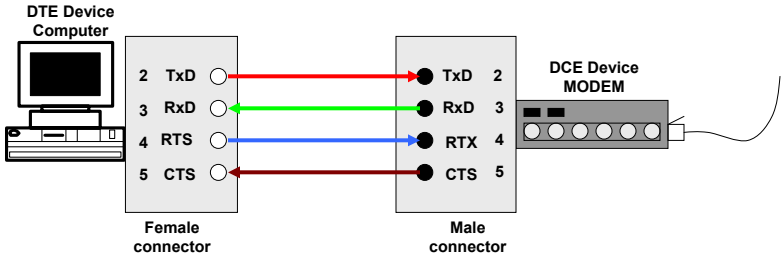


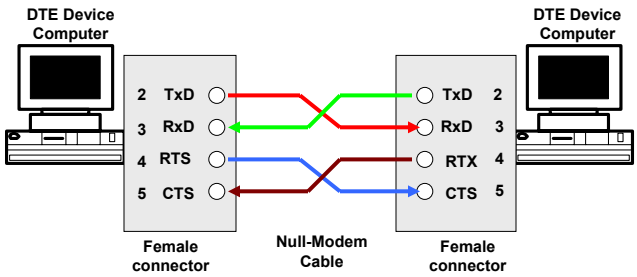


TxD: Transmite Data (from DTE to DCE)
RxD: Receive Data (from DCE to DTE)
RTS: Request to Send (from DTE to DCE)
CTS: Clear to Send (from DCE to DTE)

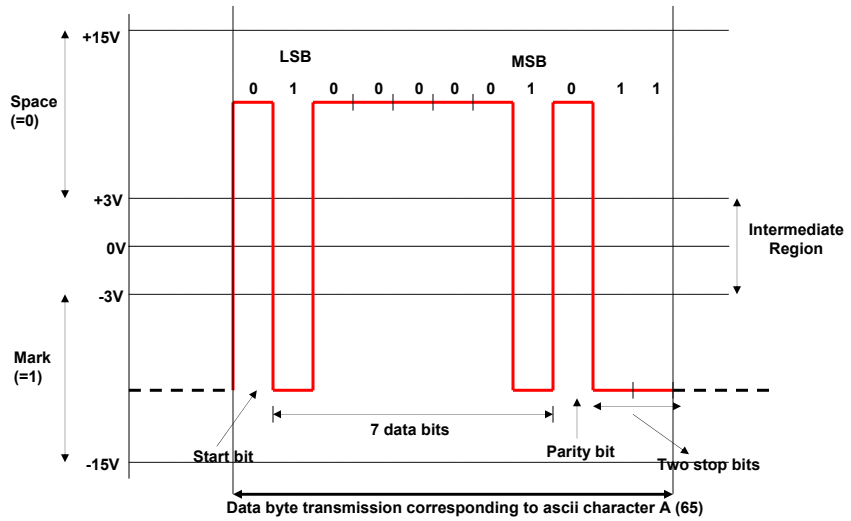
DSR: Data Set Ready (from DCE to DTE)
DTR: Data Terminal Ready (from DTE to DCE)
SD: Signal Ground
CD: Carriier Detect (from DCE to DTE)

Connnecting computer and devices using RS-232

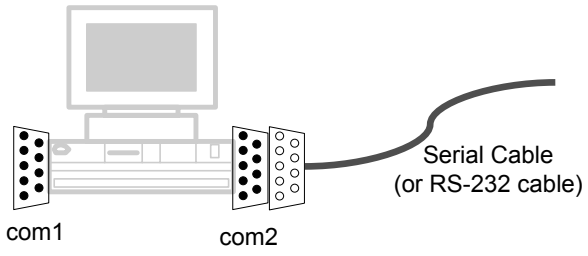




RS232C Data Transmission



- Each PC has two serial ports that are usually called COM1 and COM2 (or Serial A or Serial B)
- A serial port of a computer has a 9-pin male connector

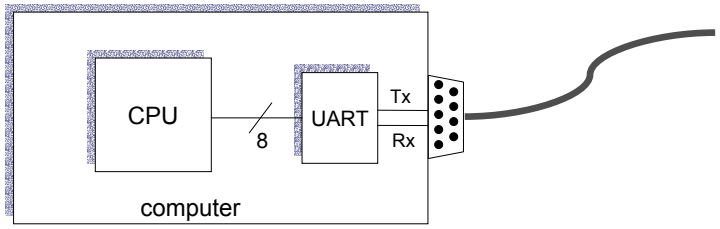


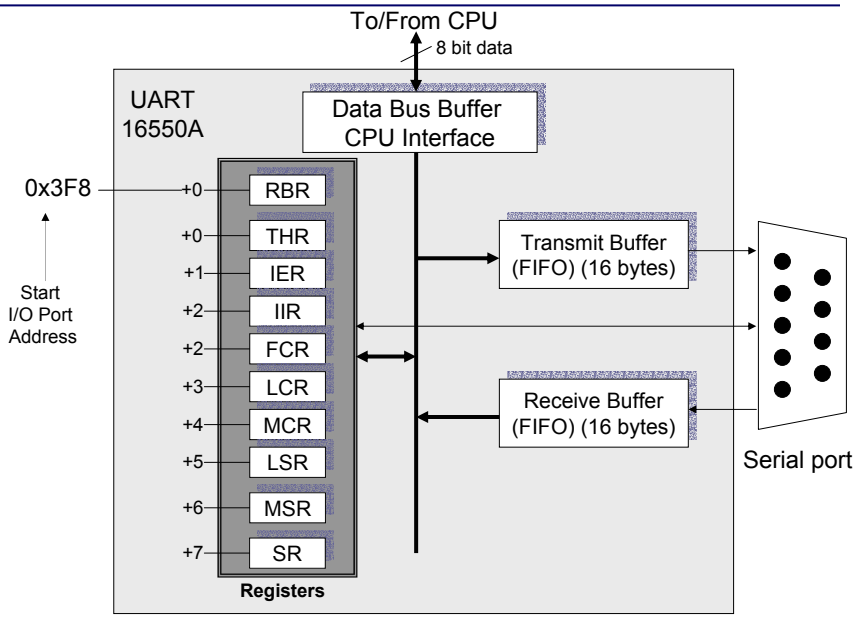
UART

The chip (or controller) that controls a serial port in a computer is Called UART: Universal Asynchronous Receiver Transmitter

UART interfaces with CPU using 8-bit parallel line
UART interface with serial port using 1 bit receive and transmit lines

Hence UART does the conversion between *parallel* and *serial* data.





UART registers

- Each Register is 8 bits
- Type of registers are:
  - RBR: Receive Buffer Registers
    - Contains the byte received if no FIFO is used
  - IER: Interrupt Enable Register
    - Is used to enable and disable interrupts that can be generated by UART.
  - LSR: Line Status Register
    - Shows the current state of the communication over Tx and Rx serial lines.
  - IIR: Interrupt Identification Register
    - Tells what kind of interrupt it is.
  - FCR: FIFO Control Register
    - Is used to control the FIFO queues.
  - LSR: Line Status Register
    - Used to set the characteristics (parameters) of RS-232 line.
  - MCR: Modem Control Register
    - Is used for handshaking operations with the attached modem device.
  - MSR: Modem Status Register
    - Contains info about the modem state.
  - THR: Transmit Holding Register
    - Contains the byte that is to be sent.

Interrupt Enable Register

Bit	Comment
0	**Received data available**
1	**Transmitter holding register empty**
2	Receiver line status register change
3	Modem status register change
4	Sleep mode (16750 only)
5	Low power mode (16750 only)
6	reserved
7	reserved

Receive interrupts will be generated
Send availability
Interrupt will be generated

Interrupt Identification Register

Bit	Value			Comment	Reset by
0	0			Interrupt pending	
	1			No interrupt pending	
1,2,3	Bit 3	bit2	bit1		
	0	0	0	Modem status change	
	0	0	1	THR empty	THR write, IIR read
	0	1	0	Received data available	RBR read
	0	1	1	Line status change	
	1	1	0		
4	0				
5	0				
	1				
6,7	Bit7	bit6			
	0	0			
	1	0			
	1	1			

Indeed, almost likely one

Line Control Register

Bit	Value			Comment
0,1	Bit1	bit0		*Data word length*
	0	0		5 bits
	0	1		6 bits
	1	0		7 bits
	1	1		**8 bits**
2	0			**1 stop bit**
	1			2 stop bits
3,4,5	Bit5	Bit4	Bit3	
	x	x	0	No parity
	0	0	1	Odd parity
	0	1	1	**Even parity**
	1	0	1	High parity
	1	1	1	
6	0			
	1			
7	0			Access to rx buffer, tx buffer and IER register
	1			Divisor latch access bit

Line Status Register (LSR)

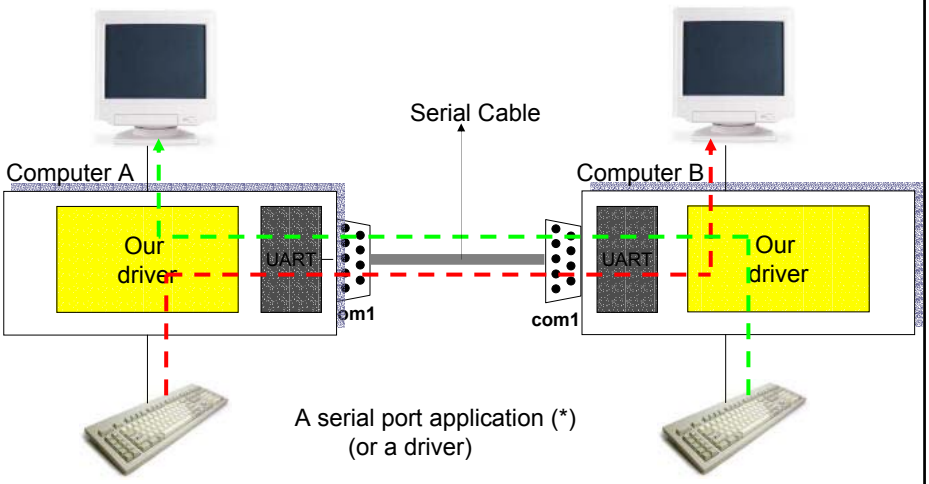
Bit	Notes
Bit 7	Error in Received FIFO
Bit 6	Empty Data Holding Registers
Bit 5	Empty Transmitter Holding Register
Bit 4	Break Interrupt
Bit 3	Framing Error
Bit 2	Parity Error
Bit 1	Overrun Error
Bit 0	**Data Ready**

Interrupt Vectors

INT (Hex)	IRQ	Common Uses
08	0	System Timer
09	1	Keyboard
0A	2	Redirected
0B	3	Serial Comms. COM2/COM4
0C	4	Serial Comms. COM1/COM3
0D	5	Reserved/Sound Card
0E	6	Floppy Disk Controller
0F	7	Parallel Comms.
70	8	Real Time Clock
71	9	Reserved
72	10	Reserved
73	11	Reserved
74	12	PS/2 Mouse
75	13	Maths Co-Processor
76	14	Hard Disk Drive
77	15	Reserved

Base Address	DLAB	Read/Write	Abr.	Register Name
+ 0	=0	Write	-	Transmitter Holding Buffer
	=0	Read	-	Receiver Buffer
	=1	Read/Write	-	Divisor Latch Low Byte
+ 1	=0	Read/Write	IER	Interrupt Enable Register
	=1	Read/Write	-	Divisor Latch High Byte
+ 2	-	Read	IIR	Interrupt Identification Register
	-	Write	FCR	FIFO Control Register
+ 3	-	Read/Write	LCR	Line Control Register
+ 4	-	Read/Write	MCR	Modem Control Register
+ 5	-	Read	LSR	Line Status Register
+ 6	-	Read	MSR	Modem Status Register
+ 7	-	Read/Write	-	Scratch Register

Polling based I/O example



Polling based I/O example-1

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8
 /* Defines Serial Ports Base Address */
 /* COM1 0x3F8          this I/O base address for serial port 1 */
 /* COM2 0x2F8                                  */
 /* COM3 0x3E8                                  */
 /* COM4 0x2E8                                  */

void main(void)
{
 int c;   /* content of  a register – a byte*/
 int ch; /* date byte */
 outportb(PORT1 + 1 , 0);   /* Turn off interrupts - Port1 (modifying IER)*/
                            /* continued on next page */
```

Continued on next page

Polling based I/O example-2

```
/*        PORT 1 - Communication Settings        */
 outportb(PORT1 + 3 , 0x80);  /* LCR access - SET DLAB ON */
 outportb(PORT1 + 0 , 0x03);  /* Set Baud rate-Divisor Latch Low Byte */
                              /* Default 0x03 =  38,400 BPS */
                              /*        0x01 = 115,200 BPS */
                              /*        0x02 =  57,600 BPS */
                              /*        0x06 =  19,200 BPS */
                              /*        0x0C =   9,600 BPS */
                              /*        0x18 =   4,800 BPS */
                              /*        0x30 =   2,400 BPS */
 outportb(PORT1 + 1 , 0x00);  /* Set Baud rate - Divisor Latch High Byte */
 outportb(PORT1 + 3 , 0x03);  /*LCR access:8 Bits, No Parity, 1 Stop Bit */
 outportb(PORT1 + 2 , 0xC7);  /*FCR access:FIFO Control Register */
 outportb(PORT1 + 4 , 0x0B);  /*MCR access:Turn on DTR,RTS,and OUT2 */

                          /* continued next page */
```

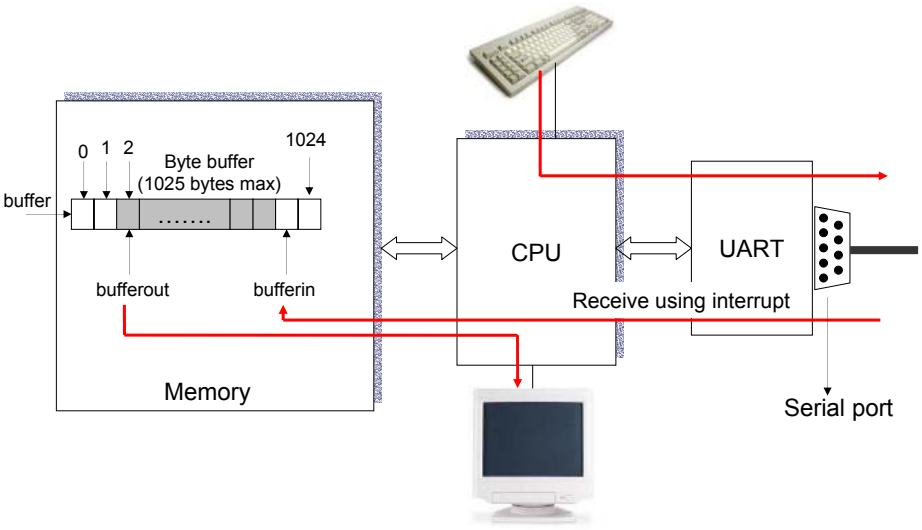
Continued on next page

Polling based I/O example-3

```
printf("\nSample Comm's Program. Press ESC to quit \n");
 do { c = inportb(PORT1 + 5);          /* Check to see if char has been */
                                       /* received.  Accessing    LSR register   */
     if (c & 1) { /* data is ready */
               ch = inportb(PORT1 + 0); /* access RBR-  get Char   */
               printf("%c",ch);}    /* Print Char to Screen          */

     if (kbhit()){
                ch = getch();        /* If key pressed, get Char */
                outportb(PORT1 + 0, ch);} /* access THR -  Send Char to
                                             Serial Port */
   } while (ch !=27); /* Quit when ESC (ASC 27) is pressed */
}


                    /* end of polling based I/O example for
                        serial ports */
```

Finished!

Interrupt driven I/O example



Interrupt driven I/O example - 1

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8  /* Port Address Goes Here */
#define INTVECT 0x0C /* Com Port's IRQ here
                        (Must also change PIC setting) */

 /* Defines Serial Ports Base Address */
 /* COM1 0x3F8                  */
 /* COM2 0x2F8                  */
 /* COM3 0x3E8                  */
 /* COM4 0x2E8                  */

int bufferin = 0;
int bufferout = 0;
char ch;
char buffer[1025];
                    /* continued on the next page */
```


Interrupt driven I/O example - 2

```
void interrupt (*oldport1isr)();

void interrupt PORT1INT()  /* Interrupt Service Routine (ISR) for PORT1 */
{
 int c;
 do { c = inportb(PORT1 + 5);   /* read LSR - 1 byte*/
     if (c & 1) /* check if data ready bit of LSR is set */
     {
          /* 1 byte of data is ready in RBR – receive buffer register */
          /* read 1 byte of data from RBR register into memory buffer */
          buffer[bufferin] = inportb(PORT1);
          bufferin++;          /* increment buffer read pointer */
          if (bufferin == 1024)
          {
               /* if end of buffer, then wrap around */
               bufferin = 0;
          }
     }
  } while (c & 1);  /* read all bytes available */
 outportb(0x20,0x20);  /* end of interrupt service routine, return to program */
}
                         /* continued on the next page */
```

Interrupt driven I/O example - 3

```
void main(void)
{
 int c;
 outportb(PORT1 + 1 , 0);       /* Turn off interrupts - Port1 */

 oldport1isr = getvect(INTVECT); /* Save old Interrupt Vector of later
                                    recovery */

 setvect(INTVECT, PORT1INT);     /* Set Interrupt Vector Entry */
                                    /* COM1 - 0x0C */
                                    /* COM2 - 0x0B */
                                    /* COM3 - 0x0C */
                                    /* COM4 - 0x0B */
          /* INTVECT is 0x0C */
          /* with this setting, the entry with index 0x0C in the interrupt vector has
             PORT1INT as interrupt service routine */
          /* ..... */
          /*   0x0C <----> PORT1INT     */  let say is this Interrupt Vector
          /* .....*/

                    /* continued on the next page */
```

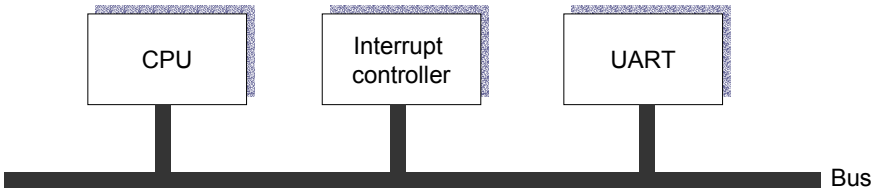

Interrupt driven I/O example - 4

```
/*        PORT 1 - Communication Settings        */
 outportb(PORT1 + 3 , 0x80);  /* SET DLAB ON */
 outportb(PORT1 + 0 , 0x0C);  /* Set Baud rate - Divisor Latch Low Byte */
                                 /* Default 0x03 =  38,400 BPS */
                                 /*       0x01 = 115,200 BPS */
                                 /*       0x02 =  57,600 BPS */
                                 /*       0x06 =  19,200 BPS */
                                 /*       0x0C =   9,600 BPS */
                                 /*       0x18 =   4,800 BPS */
                                 /*       0x30 =   2,400 BPS */
 outportb(PORT1 + 1 , 0x00);  /* Set Baud rate - Divisor Latch High Byte */
 outportb(PORT1 + 3 , 0x03);  /* LCR access: 8 Bits, No Parity, 1 Stop Bit */
 outportb(PORT1 + 2 , 0xC7);  /* FIFO Control Register access */
 outportb(PORT1 + 4 , 0x0B);  /* MCR access: Turn on DTR, RTS, and OUT2 */

 outportb(0x21,(inportb(0x21) & 0xEF)); /* Set Programmable Interrupt Controller,
                    so that we enable the interrupt that may come from serial port 1*/
           /* COM1 (IRQ4) - 0xEF  */ /* COM2 (IRQ3) - 0xF7  */
          /* COM3 (IRQ4) - 0xEF  */  /* COM4 (IRQ3) - 0xF7  */
 outportb(PORT1 + 1 , 0x01);  /* access IER - Interrupt when data received */
                                 /* continued on the next page */
```

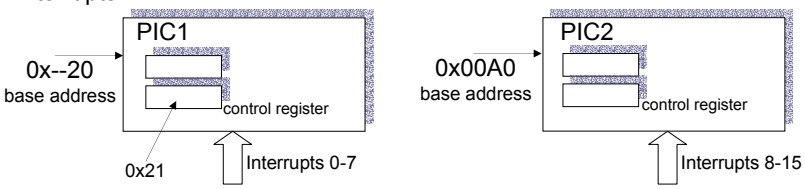
Interrupt driven I/O example - 5

```
printf("\nSample Comm's Program. Press ESC to quit \n");

 do {
    if (bufferin != bufferout)
    {   ch = buffer[bufferout];
        bufferout++;
        if (bufferout == 1024) {bufferout = 0;}
             printf("%c",ch);}
         if (kbhit()){
             c = getch();                  /* read a byte from keyboard */
             outportb(PORT1, c);           /* write the byte to the serial port */
                    }
   } while (c !=27);
 outportb(PORT1 + 1 , 0);       /* Turn off interrupts - Port1 */
 outportb(0x21,(inportb(0x21) | 0x10));  /* MASK IRQ using PIC */
                                                  /* COM1 (IRQ4) - 0x10  */
                                                  /* COM2 (IRQ3) - 0x08  */
                                                  /* COM3 (IRQ4) - 0x10  */
                                                  /* COM4 (IRQ3) - 0x08  */
 setvect(INTVECT, oldport1isr); /* Restore old interrupt vector */
}
                        /* end of program */
```


Interrupt Controller



In PC, Interrupt controller is called Programmable Interrupt Controller (PIC). It is used to enable and disable hardware interrupts and to coordinate the interrupts.



PIC1

PIC1 control register

Bit	Disable IRQ	Function
7	IRQ7	Parallel Port
6	IRQ6	Floppy Disk Controller
5	IRQ5	Reserved/Sound Card
4	IRQ4	Serial Port
3	IRQ3	Serial Port
2	IRQ2	PIC2
1	IRQ1	Keyboard
0	IRQ0	System Timer

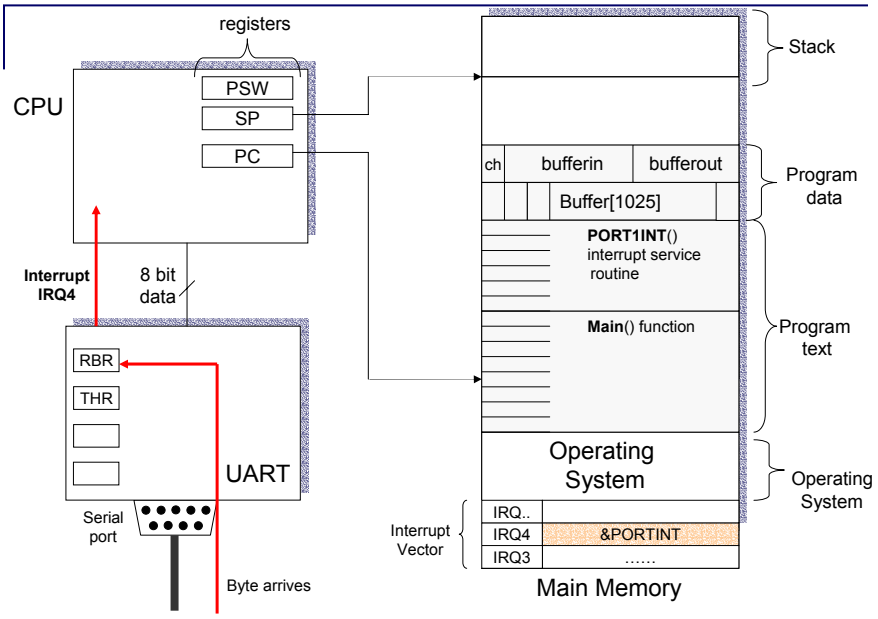
PIC2

PIC2 control register

Bit	Disable IRQ	Function
7	IRQ15	Reserved
6	IRQ14	Hard Disk Drive
5	IRQ13	Maths Co-Processor
4	IRQ12	PS/2 Mouse
3	IRQ11	Reserved
2	IRQ10	Reserved
1	IRQ9	IRQ2
0	IRQ8	Real Time Clock

Interrupt driven I/O example comments

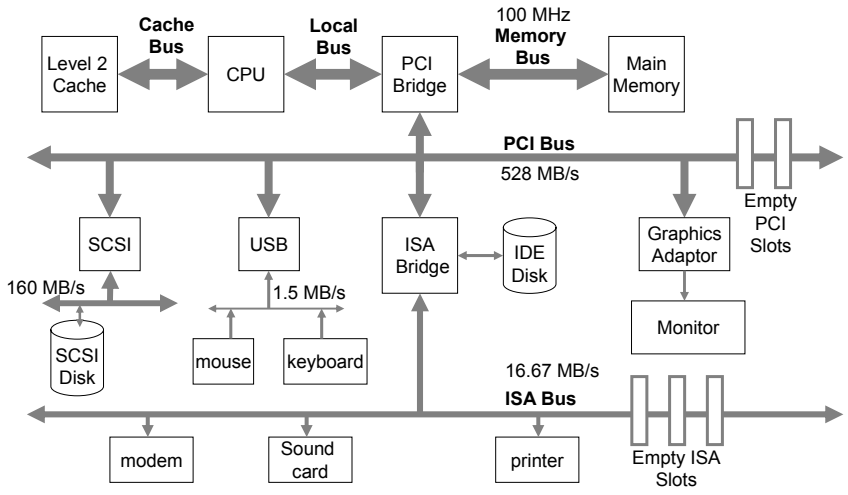
- In this example:
  - Receive from serial port is interrupt driven
  - Send to serial port is not interrupt driven
    - The data byte received from keyboard is sent to the serial port. Since a user can not type too fast, there is no risk of overflowing the output buffer of serial port UART.
    - If the data rate that has to go to the serial port would be very high, then we would also need send operation (write operation) to the serial port to be interrupt driven.



Busses

- All traffic between CPU, memory, I/O devices go over shared busses.
  - Initially one bus was enough.
  - In today's modern computers there are several busses.
- Example system: a PC has 8 busses
  - Local bus
  - Cache bus
  - Memory bus
  - PCI bus
  - SCSI
  - USB
  - IDE
  - ISA bus

A Pentium System



Different Buses

- Memory Bus
  - To talk to the memory
- Cache Bus
  - To talk to off-chip cache (level 2 cache)
- Local Bus
  - Connects to PCI bridge
- PCI bus
  - High-speed bus to connect fast devices
    - monitors, network cards. etc.
- ISA bus
  - Slow bus to connect slow devices
    - Printer, sound card, modem
- IDE Bus
  - To connect disks and CD-ROMs.
- SCSI Bus
  - High performance bus
  - Can connect fast hard disks, scanners.
- IEEE 1395
  - 50 MB/s speed
  - Connects multimedia devices, video camcorders, etc.
- USB
  - To connect slow I/O devices
  - Keyboard, mouse, digital camera

Installing I/O Cards

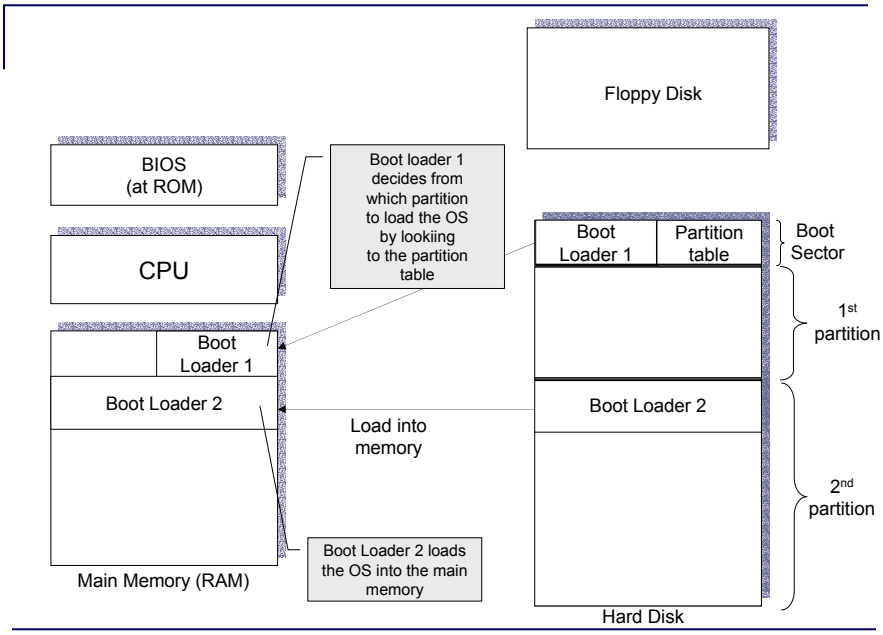
- Each I/O card has
  - An Interrupt Request Level (IRQ number)
  - An I/O port base address
    - Base address used to address the card registers.
- Example
  - A serial port com1 has:
    - IRQ = 4
    - I/O port base = 0x3F8
- Every device should have different IRQ number and different I/O port range (base, size)

Installing I/O Cards

- When we install a new card to our computer, We have to make sure that:
  - Card's IRQ number and I/O port base address do not conflict with some other card installed in the system.
- To enable this:
  - Old cards had switches (jumpers) on them to change and set IRQ level and base address.
  - New card do not need that: system automatically sets them
    - System read info about each card
    - System assigns IRQ levels and base addresses to each card
    - System makes sure that there are no conflicts.

Pentium Boot Up

- When a PC is book, the BOIS is started.
  - BOIS: Basic Input Output System
  - BIOS is a program that resides in a ROM or (flash RAM) on motherboard. It is installed at factory.
  - BOIS contains:
    - Low level I/O software
    - Procedures to read from keyboard, to write to screen, to do disk I/O, …
- BOIS jobs at startup
  - Checks abut how much RAM is there
  - Checks if keyboard and other basic devices are working
  - Scans ISA and PCI busses to detect all the devices
    - Some device are legacy (invented before plug-and-play)
      - They have fixed I/O base address and IRQ number.
    - Some devices are plug-and-play
      - Their IRQ and base address can be set by system
  - Configures the devices: assigns IRQ and base addresses to devices that need these info.
  - It then determines the book device, by trying a list of devices: floppy, cd-rom, hard-disk.
  - After determining the boot device:
    - The first sector on the boot device (**boot sector**) contains the a program that examines the partition table.
    - Partition table is located at the end of the boot sector. Determines which is the primary partition.
  - A secondary boot loader is read from that partition.
  - This boot loader loads the OS from that partition.
  - OS is started executing.



Operating System Concepts

- Each OS has some basic concepts
  - Processes
  - Deadlocks
  - Memory and Virtual Memory
  - Files
  - Input/Output
  - Protection and Security

Processes

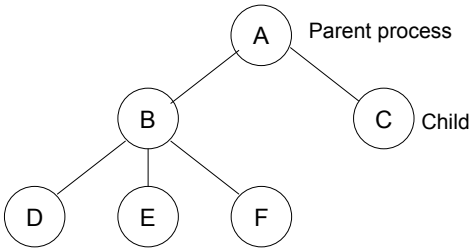
- A process is a program in execution.
- Each process has associated with it:
  - An address space that process can read from and write to
    - Address space contains:
      - Program's text (or executable code)
      - Programs data (global variables)
      - Program's stack
        - Used to store procedure parameters, local variables, etc, return addresses, etc.
  - A set of registers
    - Program counter
    - Stack pointer, ….
  - Some other information
    - Open file descriptors, etc.

Processes

- CPU is shared by multiple processes
- A running process may be suspended temporarily, so that an other process may be run
- The state of a suspended process must be saved into memory (OS space), so that when that process is run again, the state can be restored and process can continue to run without any problem.
- All info about each process (except its memory image) is saved in a table inside kernel called **process table.**

Processes

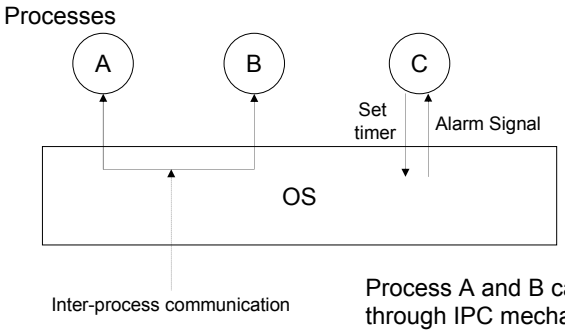
- A process can create other processes
  - They are called child processes.
- In this way a tree of processes can be obtained.



There are system calls that are used to create, terminate, and wait for processes.

A system call is an OS function.

Processes



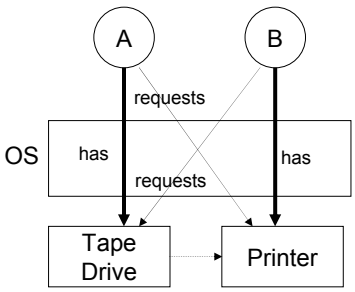
Process A and B can communicate through IPC mechanisms provided by OS

OS can send signals (software interrupts) to Processes

A process can also send a signal to an other Process.

Deadlocks

- When two or more processes are interacting (share resource) they may fall into deadlock situations, where no further work can be done.



Each of processes A and B want to read Some data from a tape in tape driver and print that out.

Process A has tape drive and requests printer.

Process B has printer and requests tape drive.

This is a deadlock situation where both processes wait for each other and can not proceed.

Memory Management

- Need to share memory by multiple processes
  - Protection and relocation problems.
  - Hardware provides mechanism for protection and relocation
    - By providing limit and base registers, etc.
  - OS used the mechanisms that hardware provide to manage and enable protection.
  - OS is the policy generated that are used over the mechanism that hardware provides.
- Virtual Memory.
  - A processed address space (virtual address space) can be larger that the maximum physical memory size.
  - OS has to keep part of process' address space that is used by CPU in memory and the rest in hard-disk.
  - OS swaps back and forth the used parts of virtual address space between memory and hard-disk.

Input/Output

- Every OS has and I/O subsystem that is used to manage and access the I/O devices connected to the computer.
- Some of the I/O software is device dependent (like device drivers) and some other part is device independent: applied to many I/O devices equally well.

Files

- OS hides the details of disks and I/O devices from application.
- It provides a nice interface of files to store data.
  - Users do not deal with disk blocks, sectors, etc.
  - They just deal with files.
- File System in OSs is usually organized as tree.

Files

- In Unix there are two kind of files:
  - Regular file that are stored on the hard-disk and that used to store info and records.
  - Special files that are used to make I/O device to look like a file.
    - Example: /dev/ttya corresponds to serial port.
    - In this way, file I/O functions such as read, write() can also be used to access the I/O devices. This provides a uniform I/O interface for application programmers.
  - There two kinds of special files:
    - Block special files: can be used to model a devices that are block addressable (random access), such as disks.
    - Character special files: can be used to model printers, modems, that accepts or outputs a character stream.
  - Special files are usually kept in /dev directory.

Security

- We need to protect files.
  - File access modes in Unix.
  - rwx bits
    - R: can read
    - W: can write
    - X: can execute
  - These 3 bits are provided:
    - For the user
    - For the group
    - For all others
  - Therefore, a total of 9 bits are used to access control a files.
- We need to protect system
  - Username/password
- …..

System Calls

What is a system call?

- The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides.
- The set of system calls varied from OS to OS, but the semantics of most calls are very similar.
- An assembly programmer uses the **instruction set** of machine to program, use system services and access hardware
- An application programmer uses the set of OS **system calls** to program and access hardware and to use services of OS.

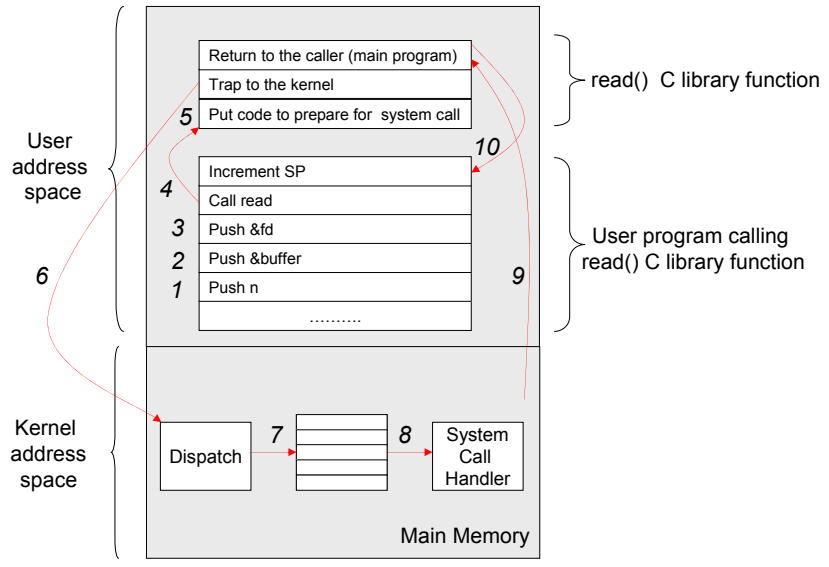
What happens when a system call is called

- A system call *n* is called from an application program.
- Upon calling a system call *n,* a function inside OS is called to handle the call.
  - When a system call is issued, the control of CPU is given from application program to Operating System.
    - Need context switch.
    - Process state has to be saved, before giving control to OS.
    - Usually this initial setup is done using assembly language.
    - For an application programmer it is difficult to program in assembly language to issue a system call.
    - Therefore, for *system call*, a simple to use *library function* with the same name is provided in the language the user programs. The library function is responsible from starting the system call by use of assembly language.

System call steps

- Example: we want to read *n* bytes from file *myfile* into a buffer (character array) *buf*.

```
void main(void)
{
    int fd;      /* file descriptor corresponding to myfile. */
    char buf[1024];   /* buffer where we want to put the data that is read
*/
    int n;        /* number of bytes that we want to read */
    int count;   / * number of bytes read */
    ……..
    fd = open("myfile", R_ONLY);
    ……..
    n = read(fd, buffer, n); /* we are calling a C library function which in
                                turn will invoke a system_call);
    if (n <= 0)
         printf ("could not read data properly);
    ……..
}
```

System Call Steps

System Calls

- POSIX standard
  - Defines the set of OS functions that an POSIX-compliant OS needs to support
  - These OS *functions* are in reality library functions, which call their corresponding system calls.
  - More than one *function* can be implemented with one system call, although usually there is one-to-one mapping
  - Some *functions* may be implemented in user-space without requiring system call.
  - The functions can be categorized depending on the service they provide:
    - Process management functions
    - File management functions
    - Directory and file system functions
    - Miscellaneous functions.
  - We will also refer to these *POSIX functions* as *system calls*.

Some important POSIX system calls

Call	Description
Process Management	
pid = fork()	Ceate a child process identical to parent
pid = waitpid(pid, &statloc, oprions)	Wait for a child toW terminate
s = execve(name, argv, environp)	Replace a process' memory image
exit(status)	Terminate process execution and return status
File Management	
fd = open(file, how, …)	Open a file for reading, writing, or both
s = close(fd);	Close and open file
n = read(fd, buffer, nbytes);	Read data from a file into a buffer
n = write(fd, buffer, nbytes):	Write data from a buffer into a file.
position = lseek(fd, offset, whence);	Move the file pointer
s = stat(name, &buf)	Get a file's status information
Directory and File System Management	
S= mkdir(name, mode)	Create a new directory
S = remove directory	Remove an emoptry directory
S = link(name1, name2)	Create a new entry, name2, pointing to name1
S = unlink(name)	Remove a directory entry

Call	Description
Directory and File System Management	
s = mkdir(name, mode)	Create a new directory
s = remove directory	Remove an emoptry directory
S = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system
Miscellanous	
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Win32 API

Unix	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess does the job
Exit	ExitProcess	Terminate Execution
open	CreateFile	Create a file or open an existing file
Close	CloseHandle	Close a file
read	ReadFile	Read Data from a file
Write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Descroy and existing file
mount	(none)	Wind32 does not support mout

Unix	Win32	Description
umount	(none)	WIn32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (NT does)
Kill	(none)	Win32 does not signals
time	GetLocalTime	Get the current time

Operating System Structure

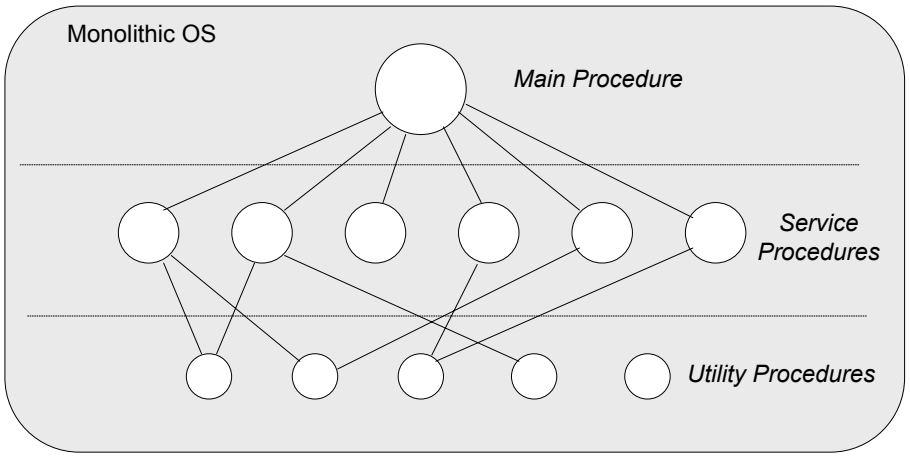
Structuring OS design

- Lets to to the inside of OSs: how they are structured
- There are 5 different major structures that are tried
  - Monolithic systems
  - Layered systems
  - Virtual Machines
  - Exokernels
  - Client-Server Model

Monolithic Systems

- We can say that there is minum amount of structure
- Everything is put together into a big chunk of code.
- Each procedure has a well defined interface
  - But every procedure can call every other procedure
    - Every procedure is visible to every other procedure: no information hiding.
- All files (and procedures inside them) are compiled and linked to a single OS executable file
- There is a actually some structure

Monolithic Systems - Simple Structure Model



A simple structuring model for monolithic Operating Systems

Layered Systems

Layer	Function
5	The Operator
4	User programs
3	Input-Output Communication
2	Operator-process communication
1	Memory disk management
0	Processor Allocation and Multiprogramming

Example Systems: THE operating system, MULTICS

Virtual Machines

- A *Virtual machine OS kernel* provides several *virtual machines* to the next layer up
- *Virtual machine OS kernel* does
  - Multiprogramming
- *Virtual machines* are
  - Exact copies of the bare hardware
  - They include kernel and user mode hardware emulation
  - They have I/O, Interrupts, and everything else a real machine has.
- A *virtual machine* can run any OS on top of it.

Virtual Machines - Example