# What is an OS?

- OS is a software that sits between hardware and applications and provides interfaces to applications for accessing hardware

# What is an application?

example.c

```
#include <stdio.h>
main() {
  printf("Hello world\n");
}
```

```
gcc example.c
./a.out
```

a.out is an application.

# What is an application?

- Firefox
- Terminal
- Power point
- etc.

# Hardware

- Disk
- Network device
- RAM
- CPU
- Monitor
- Keyboard
- etc.

## Disk interface


write_to_sector(0)
write_to_sector(7)
read_from_sector(9)

The disk interface is the sequence of sectors. In this example, the sector size is 512 bytes. The disk interface allows programmers to read/write from/to sectors.

# OS interface

fd = open(filename, “w”);

write(fd, buf, 32);

Even though the entire file is not stored in consecutive sectors, the OS interfaces allow you to access the file sequentially

OS interface is the file system. A file is divided into the contiguous blocks, where the block size is equal to sector size. Blocks can be stored in random sectors. The OS hides the complexity of blocks to sector translation from the programmers.

# OS interface

fd = open(filename, “w”);

write(fd, buf, 32);

Even though the entire file is not stored in consecutive sectors, the OS interfaces allow you to access the file sequentially

| BLOCK | SECTOR |
|---|---|
| 0 | 15 |
| 1 | 23 |
| 2 | 43 |
| 3 | 29 |
| … | … |

The OS keeps the mapping from blocks to sectors.

# What does an OS do?

- OS is a library
  - write_to_console, write, read, etc.

# write_to_console

- An API that writes a character to the display device
- Applications are not allowed to use their own implementation of write_to_console
  - Different from the standard library where you can use a custom implementation of library functions, e.g., strcpy, strcmp, etc.

## What does an OS do?

- OS is a library
  - write_to_console
- Enforce use of OS library
  - Access to hardware recourses is only permitted through the OS library APIs

We can not let the applications access the console directly, because in that case, an application can hijack the console and prevent other applications from using them.

# Isolation

- An application can't see the data of other applications
- Applications are untrusted
  - A gaming application can't see your password in a banking application
- OS is trusted
  - If our OS is malicious, then it can allow malicious applications to steal our passwords
  - We all trust our OS to protect us from malicious software

# What does an OS do?

- OS is a library
  - write_to_console
- Enforce use of OS library
  - Access to hardware recourses is only permitted through the OS library APIs
- Isolation

## Sharing of hardware resources

- E.g., OS decides which application is going to use the display device at a given time

## What does an OS do?

- OS is a library
  - write_to_console
- Enforce use of OS library
  - Access to hardware recourses is only permitted through the OS library APIs
- Isolation
- Share hardware resources among applications

## Tentative weekly lecture plan

1. Introduction to OS and x86 architecture
2. Threads
3. Synchronization
4. Processes, System calls

5, 6. Segmentation

7, 8. Paging

9, 10. File system

11, 12. Concurrency

13. Other OS designs

## Books

- xv6 book
- xv6 code listing
- Operating System Concepts, 9th Edition, Wiley by Silberschatz, Galvin, Gagne
- Intel software developers manual (for x86 architecture)
- Pintos documentation

# Exams

- You have to bring the xv6 book and xv6 code listing during the exams

# Prerequisite

- Good C programming skills

## Evaluation

- Programming : 20
- Homework : 10
- Mid semester : 25
- End semester : 40
- Refresher module : 5

## Passing criteria

- Meet the following conditions
  - At least 15 marks in midsem + endsem (65)
  - At least 10 marks in other components (35)

# Office hours

- Mon, Wed : You can meet me just after class
- Tue, Thurs : 3pm -3:30pm   (A505, New academic)

# Plagiarism

https://www.iiitd.ac.in/academics/resources/academic-dishonesty

## What is inside a.out?

- gcc -m32 -fno-pic example.c
- objdump -dx a.out |less

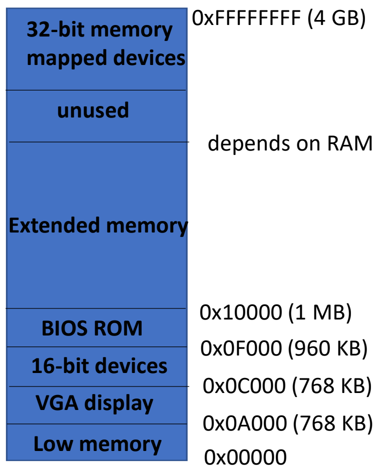
a.out contains CPU (x86) instructions

Instructions are stored in memory

## Physical address space

- Each byte in the main memory (RAM) has an address
- Using 32 bits, we can generate $2^{32}$ unique values
- In x86 32-bit architecture, the physical address space contains $2^{32}$ addresses

# Physical address space



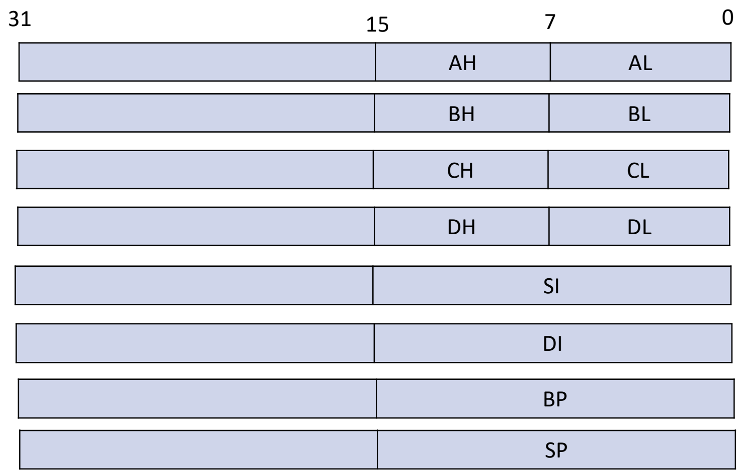

For larger RAM (> 4GB), PAE support is available in x86. PAE can only be used with page-tables (will discuss later). PAE allows a 48-bit physical address on 32-bit x86 architecture.

# Registers

- A general-purpose register can hold a 32-bit value (data/address)

- X86 has eight general-purpose registers
  - EAX, ECX, EDX, EBX, ESI, EDI, EBP, ESP

# Registers




| 16 bit | 32 bit |
|---|---|
| AX | EAX |
| BX | EBX |
| CX | ECX |
| DX | EDX |
| | ESI |
| | EDI |
| | EBP |
| | ESP |

# What is inside a.out?

- gcc -m32 -fno-pic example.c
- objdump -dx a.out |less

a.out contains CPU (x86) instructions

Instructions are stored in memory

CPU reads the instructions from memory and executes them

# x86 CPU interface

- The instruction pointer (EIP register) contains the memory address where the next instruction is stored

```
while (1) {
   instruction = fetch(EIP);
   execute instruction;
   if (instruction doesn’t modify EIP) {
      EIP = EIP + length(instruction);
   }
}
```

# x86 instructions

- Branch instructions can modify EIP
  - call, ret, jmp, je, jne, etc. are branch instructions
  - Transfer the control to other parts of the code
- Instructions can read/write from/to physical address space
- Instructions can perform arithmetic or logical operations on registers and physical addresses

# x86 instructions

add $1, %eax

An instruction consists of an opcode followed by one or more operands

Operands are constants, registers, direct memory addresses, and indirect memory addresses

# x86 instructions

add $1, %eax

The instructions are the sequence of bytes

The above notation is used by the programmers to write an assembly program that can be converted into the bytecode using an assembler

# Register mode

movl %eax, %edx

%edx = %eax

b,w,l denotes the size of operands

AT&T (gcc) syntax: opcode src, dst

## Register mode

movb %al, %bl

%bl = %al

b,w,l denotes the size of operands
AT&T (gcc) syntax: opcode src, dst

## Register mode

movw %ax, %bx

%bx = %ax

b,w,l denotes the size of operands

AT&T (gcc) syntax: opcode src, dst

# Immediate

movl $0x123, %eax

%eax = 0x123;

## Direct memory access

movl 0x123, %edx

%edx = *((int32*)0x123);

(memory read)

# Direct memory access

movl %edx, 0x123

*((int32*)0x123) = %edx;

(write to memory)

# Indirect memory operand

disp(base, index, scale)

disp: 32 bit signed integer
base, index : register
scale: 1,2,4,8

address: disp + base + (index * scale)

# Indirect memory access

movl (%ebx), %edx

%edx = *((int32*)%ebx)

(memory read)

# Indirect memory access

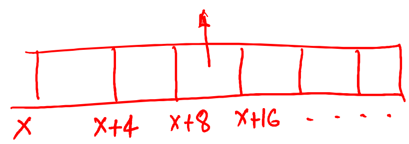
```
movl 4(%ebx), %edx

%edx = *((int32*)(%ebx+4))
```

# Indirect memory access

movl (%ebx,%ecx,4), %edx

%edx = ((int32*)(%ebx))[%ecx]




int a[10];

a[2] = 100;

a + (2×4)

ebx[ecx]

ebx + (ecx*4)

## Indirect memory access

movl (%ebx,%ecx,1), %edx

ebx + ecx

%edx = *((int32*)(%ebx + %ecx))

# Indirect memory access

movl %edx, (%ebx,%ecx,1)

*(int32*)(%ebx + %ecx) = %edx

(write to memory)

# Indirect memory access

movl $0x0, (%ebx,%ecx,1)

*(int32*)(%ebx + %ecx) = 0x0

do we need to specify the “l” suffix

## Indirect memory access

mov $0x0, (%ebx,%ecx,1)

is ambiguous

It is not evident from this instruction that, whether we want to store 1,2, or 4-byte value (0) into the destination memory address. A “b”, “w”, or “l” suffix is required to eliminate this ambiguity.

## Indirect memory access

mov %eax, (%ebx,%ecx,1)

is not ambiguous

Here, we know that %eax contains a 4-byte value.