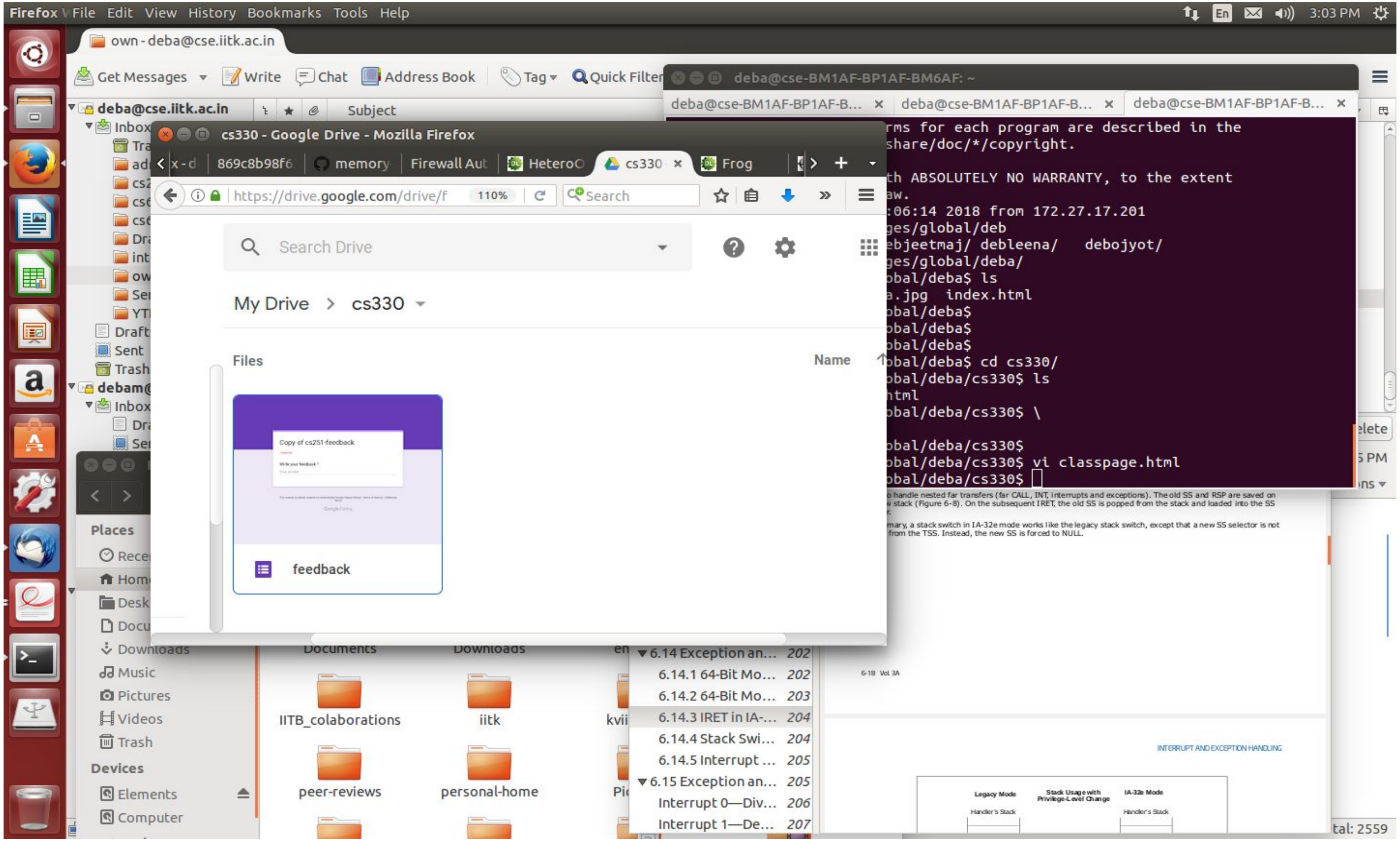
Operating Systems

Introduction

Debadatta Mishra, CSE, IITK

Computers: user view



Features of modern operating systems

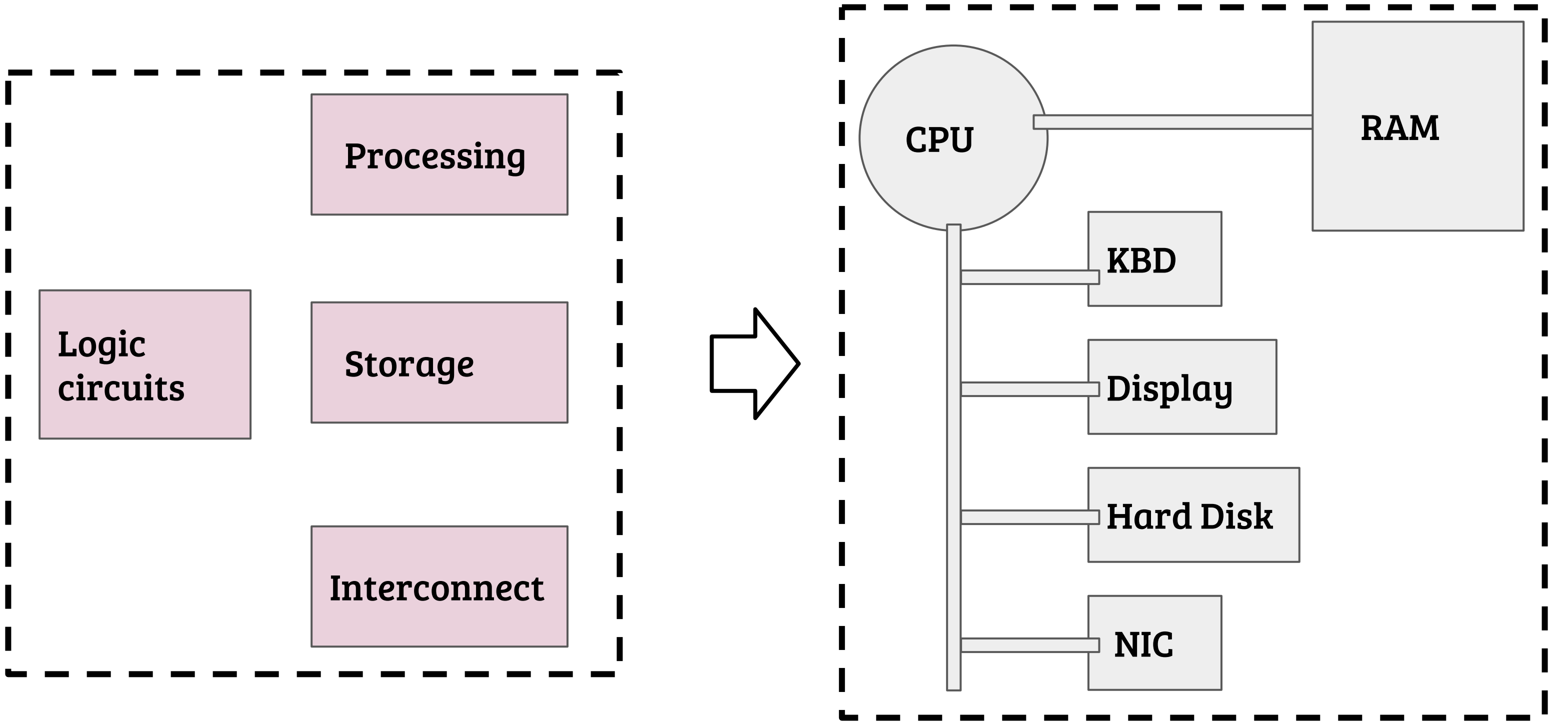
- ➔ Multiple applications active
- ➔ Simple to use
- ➔ Simple to add new applications
- ➔ Notion of a user, login
- ➔ GUI, network connectivity
- ➔ ….

Computers: application developer view

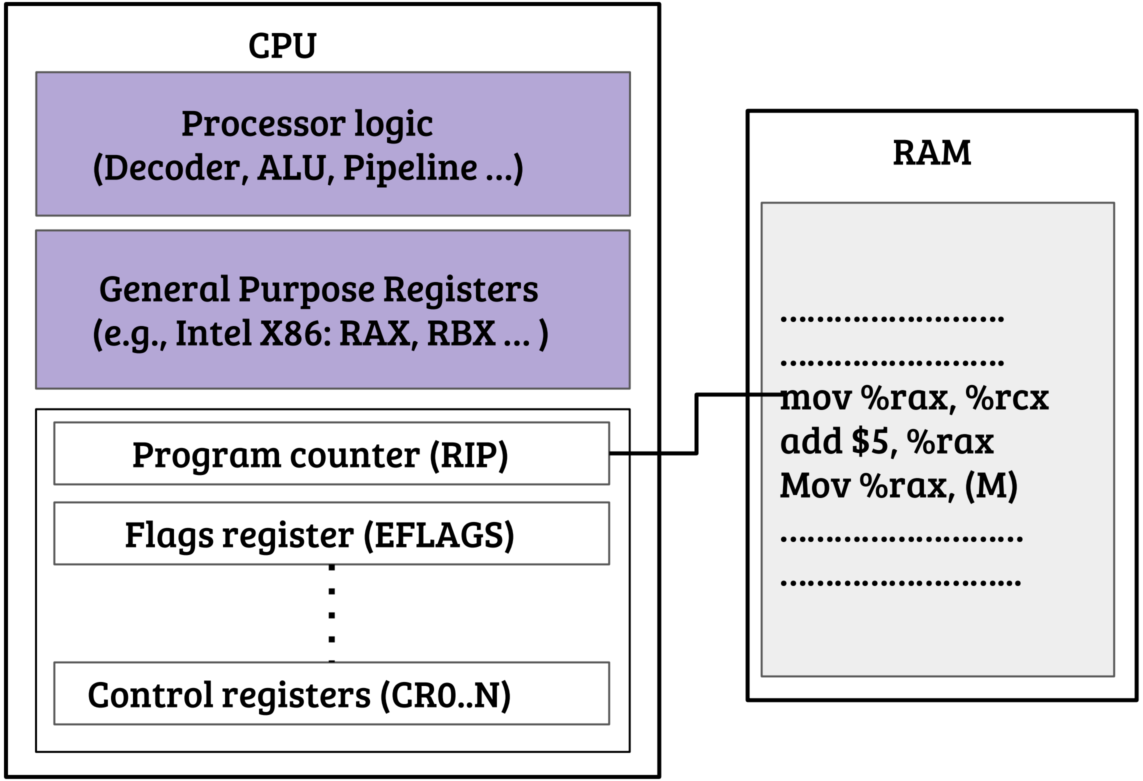
```
#include<stdio.h>
main()
{
      printf("Hello cs330!");
}

$ gcc hello.c
$ ./a.out
Hello cs330!
$
```

- ➔ We are already familiar with ESC 101, ESO 207, CS251
- ➔ What all we need to develop applications?
  - ◆ Data structures and Algorithms
  - ◆ Algorithms → Programming language
  - ◆ Library
  - ◆ Compiler/Interpreter
- ➔ How executed?

Computers: physical view (CS220)



Computing model and ISA



- ➔ PC points to the instruction (defined by ISA)
- ➔ Instruction execution
  - ◆ Fetch
  - ◆ Decode steps
  - ◆ Operand fetch
  - ◆ Execution
  - ◆ Result store
  - ◆ Commit
- ➔ Who sets PC to the first instruction?

Application program → Execution

```
int find_sum (int len, int *arr)
{
        int sum  = 0, i = 0;
        for(i=0; i < len; ++i)
        sum += arr[i];
        return sum;
}
………….
………….
………….
```

→

```
4006b6:     55              push   %rbp
4006b7:     48 89 e5        mov  %rsp,%rbp
4006ba:     89 7d ec        mov %edi,-0x14(%rbp)
4006bd:     48 89 75 e0     mov %rsi,-0x20(%rbp)
……………….
……………….
……………….
4006fd:     8b 45 f8        mov  -0x8(%rbp),%eax
400700:     5d              pop  %rbp
400701:     c3              retq
```

- ➔ High-level language → Assembly code → Machine code
- ➔ Who performs the translation?

Application program → Execution

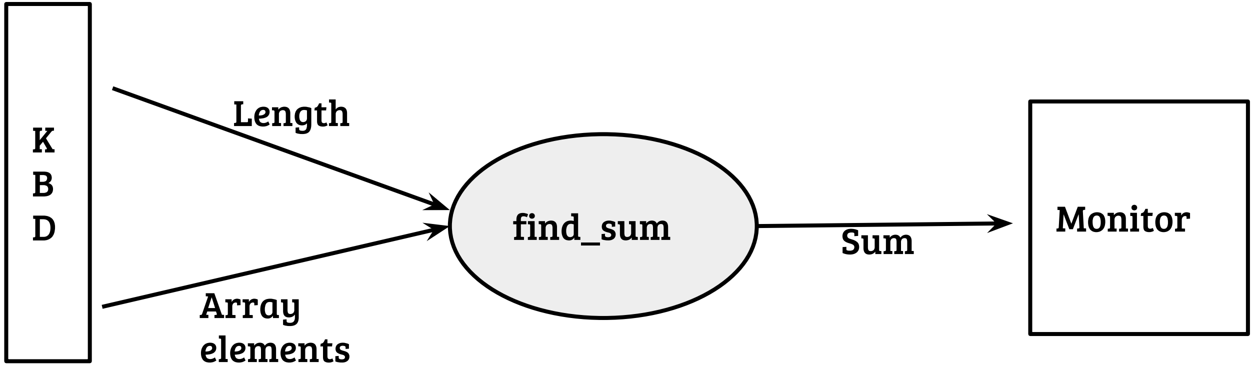
```
int find_sum (int len, int *arr)
{
        int sum  = 0, i = 0;
        for(i=0; i < len; ++i)
        sum += arr[i];
        return sum;
}
………….
………….
………….
```

Compiler
Assembler
Linker

```
4006b6:     55              push   %rbp
4006b7:     48 89 e5        mov  %rsp,%rbp
4006ba:     89 7d ec        mov %edi,-0x14(%rbp)
4006bd:     48 89 75 e0     mov %rsi,-0x20(%rbp)
……………….
……………….
……………….
4006fd:     8b 45 f8        mov  -0x8(%rbp),%eax
400700:     5d              pop  %rbp
400701:     c3              retq
```

- ➔ Where is executable file stored?
- ➔ Why to store it in the first place?
- ➔ Who loads the executable into RAM?

Application program (input/output)

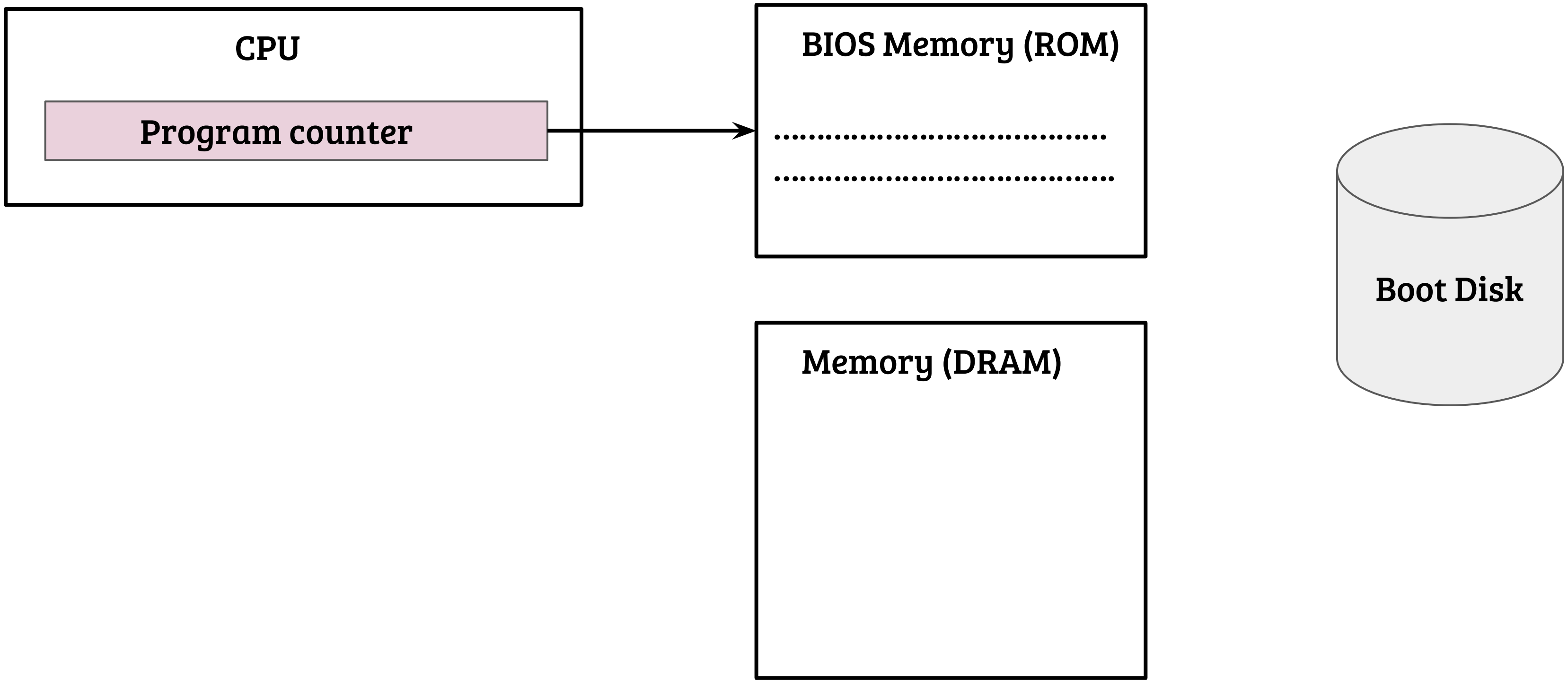


- ➔ Read inputs from keyboard
- ➔ Allocate memory for the array
- ➔ Write output into monitor

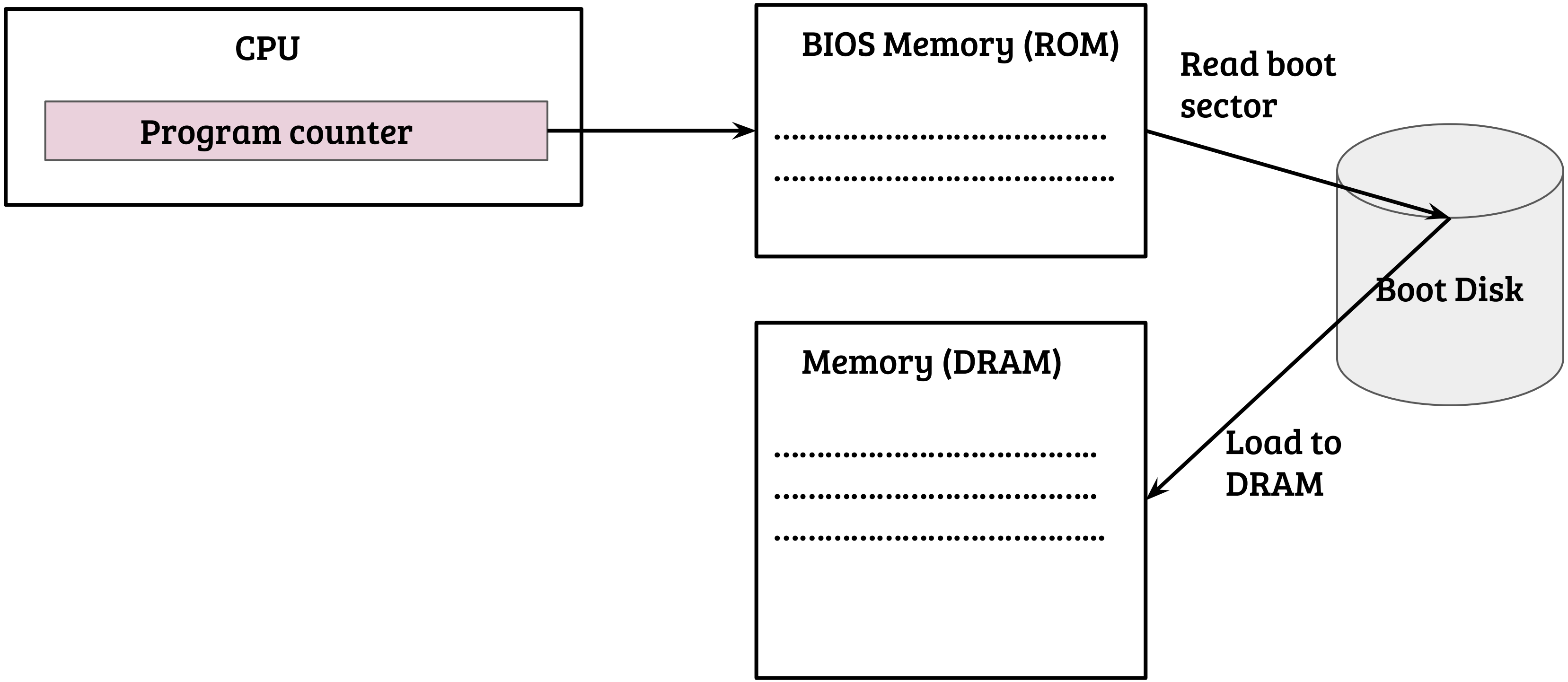
Multitasking

- ➔ Concurrent execution of multiple applications
- ➔ Resource multiplexing: How do they share resources?
- ➔ Isolation: How are they isolated?
- ➔ Resource management: How efficiently are the resources utilized?

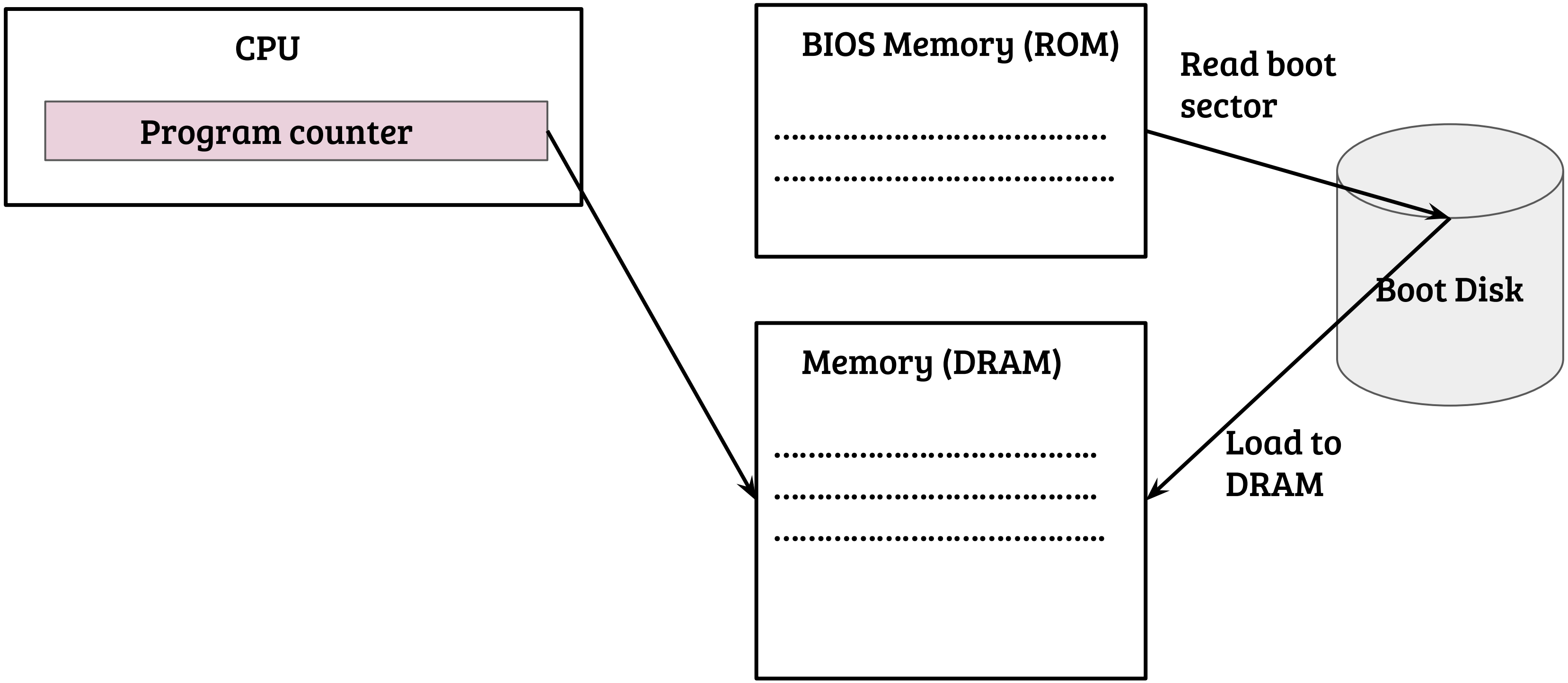
How it all starts? A simplified view



How it all starts? A simplified view



How it all starts? A simplified view.



Application load and execution

- ➔ Alternate 1: BIOS loads the applications
- ➔ Alternate 2: Let the BIOS load a standard application (lets call it K), which loads other applications.
  - What are the features of K?
  - How will it locate and load other applications?

Input/output

- ➔ Alternate 1: All applications implement their own I/O operations
- ➔ Alternate 2: Applications use a library that implements all I/O operations
- ➔ Alternate 3: Let application K provide interfaces for I/O operations
  - ◆ Advantages
  - ◆ Challenges
  - ◆ Example: Socket interface

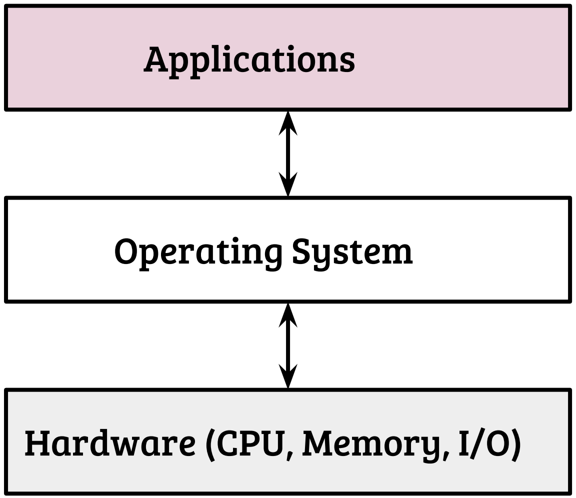
Resource multiplexing

- ➔ Alternate 1: No multiplexing
- ➔ Alternate 2: Cooperative multiplexing
- ➔ Alternate 3: Let application K provide software abstraction for each resource and multiplex the actual resources
  - ◆ Advantages
  - ◆ Challenges
  - ◆ Nature of interfaces
  - ◆ Example: File System, Virtual Memory etc.

Resource management

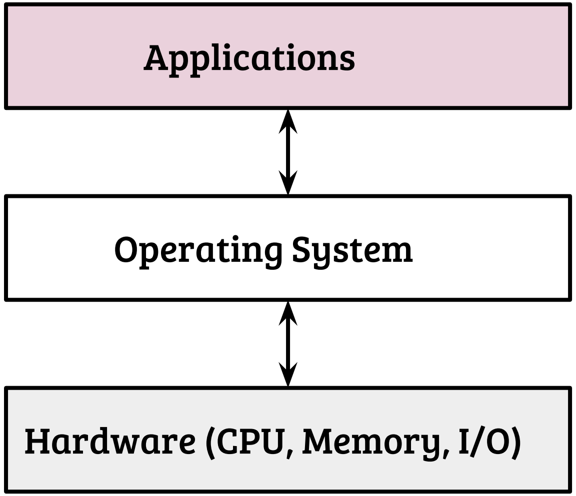
- Alternate 1: “Free for all” management
- Alternate 2: Let application K manage all the underlying resource
  - Resource utilization
  - Performance and scalability
  - Fairness
  - QoS
  - ….
  - Example: Process priority

What is an Operating System?



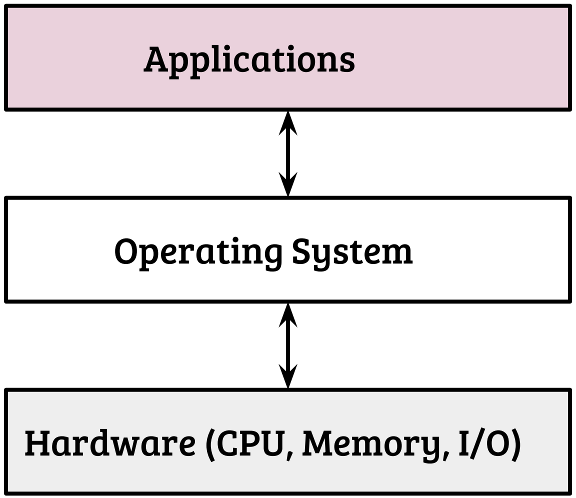
- ➔ A middleware between application and hardwares
  - ◆ Design goals: simple and easy to use APIs, efficient use of hardware features
- ➔ A resource multiplexer
  - ◆ Design goals: isolation, efficiency, control
- ➔ A resource manager
  - ◆ Design goals: efficiency, fairness, policy support

Why study OS?



- ➔ S1: I want to be an application developer, why should I care about OS design?
- ➔ S2: I want to be a hardware architect, Why should I bother?
- ➔ S3: I am interested in theory of CS, Why should I bother?

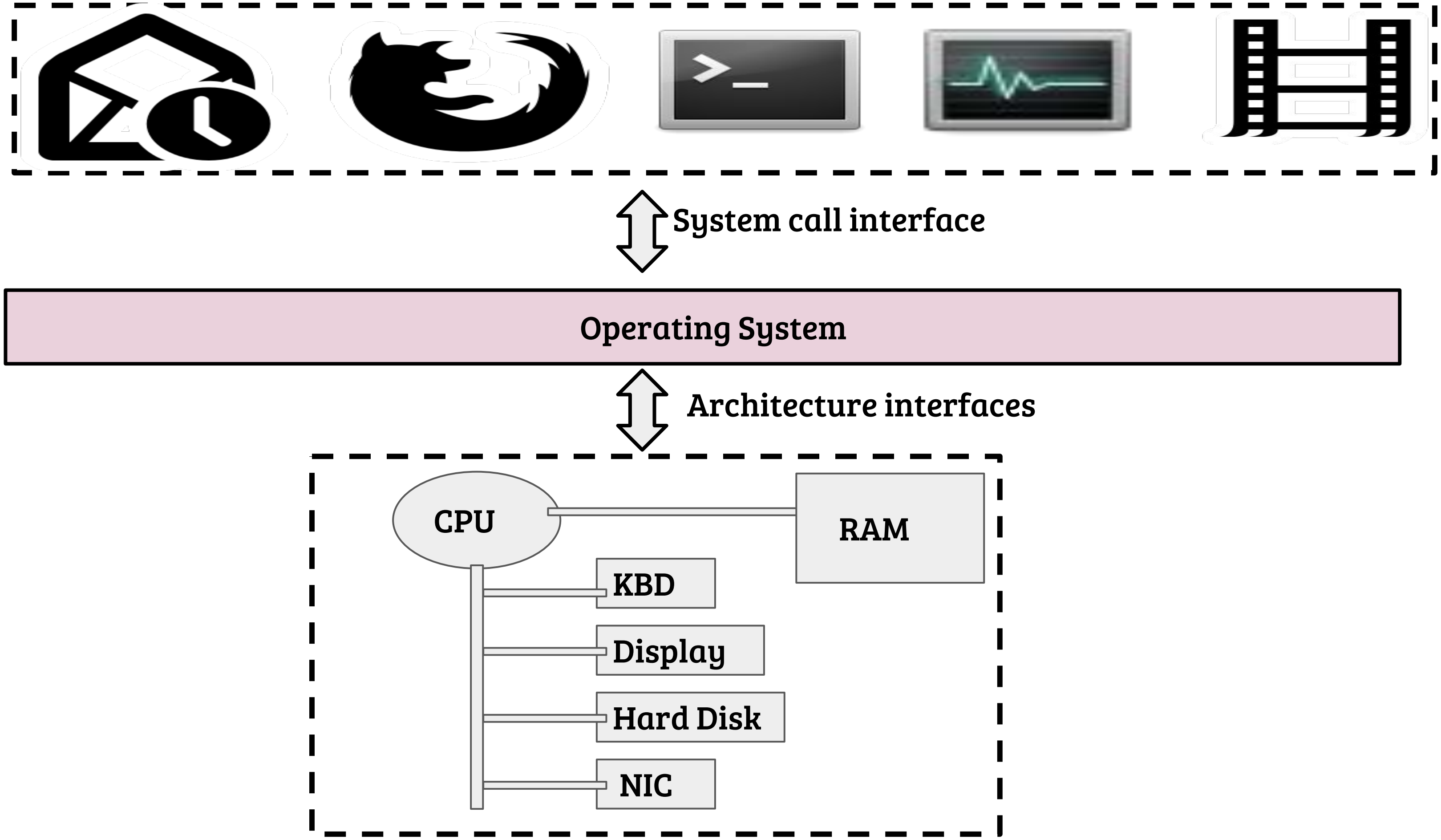
Why study OS?



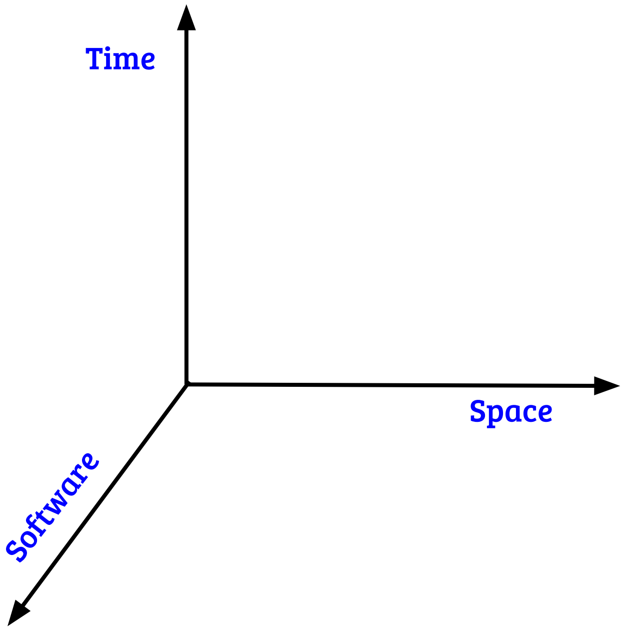
- ➔ S1: I want to be an application developer, why should I care about OS design?
- ➔ S2: I want to be a hardware architect, Why should I bother?
- ➔ S3: I am interested in theory of CS, Why should I bother?

Not yet convinced? You have to do it anyway :-)

Resource sharing: Multiplexing/Virtualization

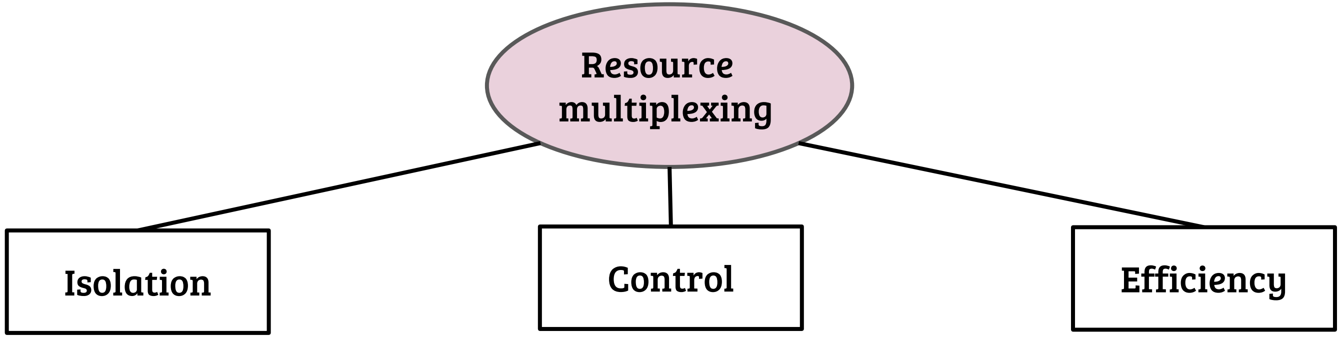


Multiplexing/Virtualization mechanisms



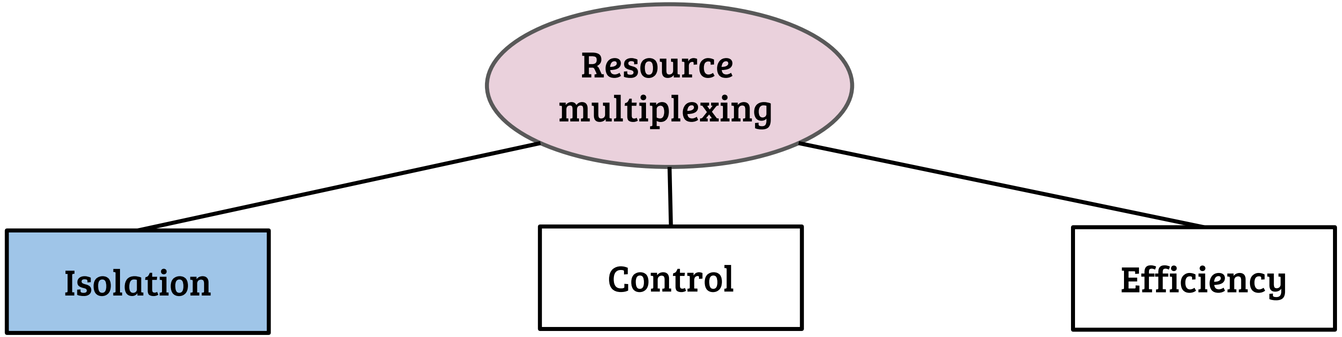
- Time sharing
  - A resource is allocated to different applications at different times
  - Resource should support “visible state” along with operations like “save” and “restore”
  - Example: a single CPU
- Space sharing
  - Resource can be partitioned into smaller units
  - Example: Memory
- Software multiplexing
  - No inherent multiplexing support from the resource
  - Every operation is through a software multiplexer
  - Example: NIC, Disk

Multiplexing/Virtualization requirements [1]



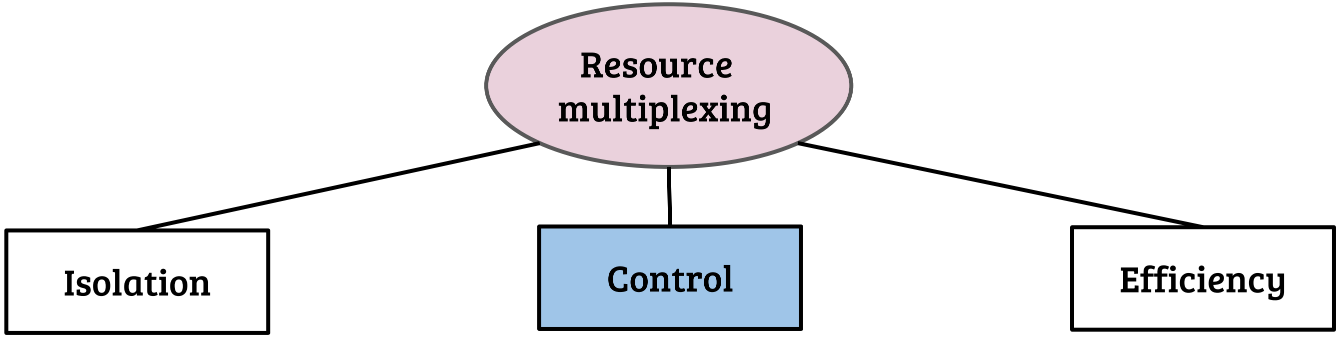
1. G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Commun. ACM 17 (7) (1974) 412–421

Multiplexing/Virtualization requirements



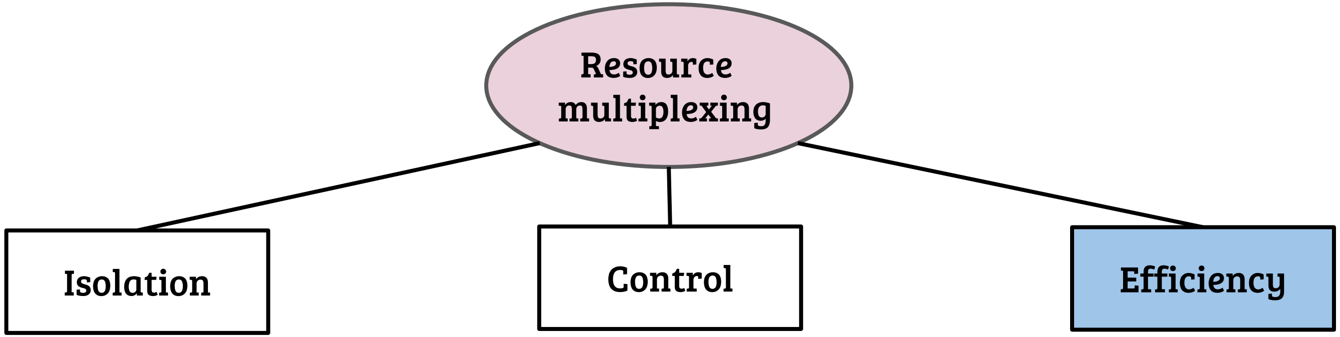
- ➔ Resources when used by one application (say A) should not be accessible from other applications, if not specifically allowed by A
- ➔ Alternate 1: All accesses to resources are through the OS (CPU?)
- ➔ Alternate 2: Resources are partitioned, but the “partitioning operations” are accessible only by the OS. How to achieve this?

Multiplexing/Virtualization requirements



- ➔ OS can “gain control” of any resource at any point of time
- ➔ Alternate 1: All accesses to resources are through the OS
- ➔ Alternate 2: An event driven OS intervention, in the worst case after a configured time interval
  - ◆ Event configurations should be allowed only from OS
  - ◆ How?

Multiplexing/Virtualization requirements

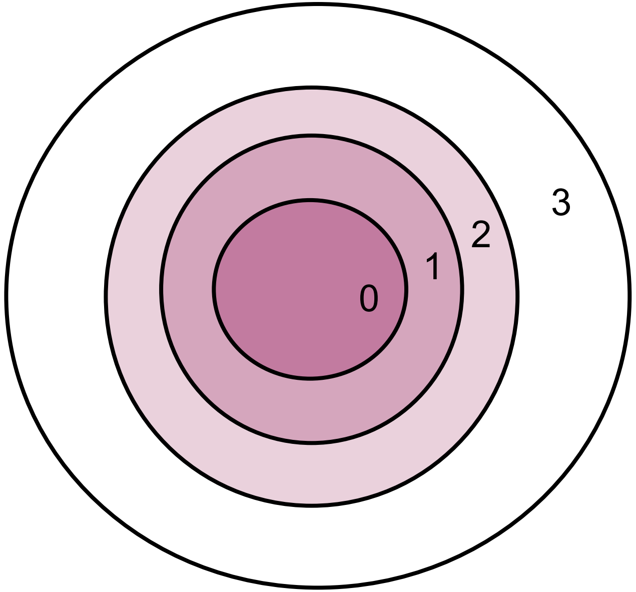


- ➔ Applications should use the resource directly → without OS intervention
- ➔ All accesses to resources are through the OS, not efficient :(
- ➔ How to apply restrictions to direct access (required for isolation and control)?

Limited direct access

- What to limit?
  - Instructions
  - Operands
  - Both
- Where to limit?
  - Hardware
  - Software
  - Both
- However, applications need gateways
  - Example 1: Application wants to sleep
  - Example 2: Application wants to expand its memory allocation
  - Example 3: Application wants to communicate with other application (legitimately!)

X86: rings of protection



- 4 privilege levels: 0→ highest, 3→ lowest
- Some instructions and access to CPU registers are allowed only in privilege level 0.
  - Example: Access to registers responsible for memory partitioning, e.g., CR3, segment registers
- OSs build limited access mechanisms using the architectural support as basis
- Most OSs use only two levels → 0 and 3
- Subtle architectural mechanisms to switch between privilege levels

Privilege enforcement example - 1 (Linux x86_64)

```
#include<stdio.h>
main()
{
        asm volatile("hlt");
  }
```

- ➔ HLT → Halt the core till next external interrupt
- ➔ Executed from user space → Protection fault
- ➔ Action: Linux kernel kills the application

Privilege enforcement example - 2 (Linux x86_64)

```
#include<stdio.h>
main()
{
        unsigned long cr3_val;
        asm volatile("mov %%cr3, %0;"
                : "=r" (cr3_val)
                :
                );

        printf("%lx\n", cr3_val);
}
```

- ➔ Read CR3 register
- ➔ Executed from user space → Protection fault
- ➔ We are using “mov” instruction, but the operand is “privileged”

Privilege enforcement example - 3 (Linux x86_64)

```
#include<stdio.h>
main()
{
      unsigned long cs_val;
      asm volatile ("mov %%cs, %0;"
                    : "=r" (crs_val)
                    :
          );
       printf("%lx\n", cs_val);
       asm volatile ("mov %0, %%cs;"
          :
          : "r" (cs_val)
        );
}
```

- ➔ Reading the content of code segment register CS (using MOV) is allowed
- ➔ Direct write to code segment register CS (using MOV) is not allowed

I want to sleep! How to go about it?

- ➔ Alternate 1: Execute a tight-loop

```
while(elapsed_time < sleep_time);
```

- ➔ Alternate 2: Execute HLT instruction
  - ◆ But HLT is not allowed
  - ◆ How to end the sleep?

May be OS can help!

Application (Ring 3)

……….
sleep(X)
………..

Entry Door

```
do_sleep(int time)
  {
    CreateTimer(time);
    If(ready_to_run == 0)
      HLT
    else
       nApp = SelectApp( );
       ReplaceApp(this, nApp);
  }
```

Operating System (Ring 0)

Entry into ring-0: necessary evils!

External events a.k.a Interrupts

Software caused faults and exceptions

Ring-0

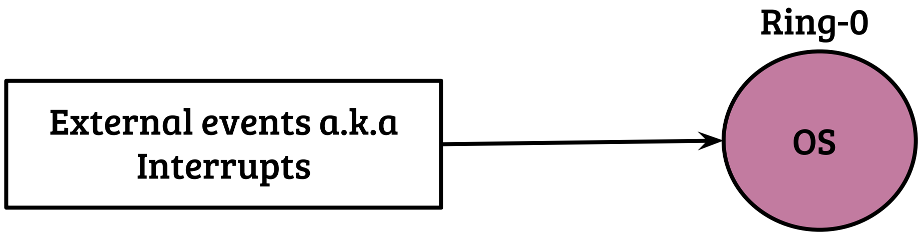
OS

System call from Ring - 3

Special instructions (Priv. procedure call)

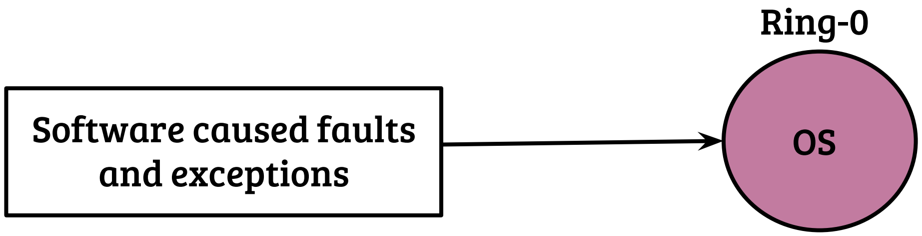
Software interrupts (INT instructions)

Entry into ring-0: External events / Interrupts



- Why OS should handle it?
  - Software multiplexing
  - Control
  - Isolation
- How OS handles it?
  - Register handlers, defines a privileged processor state to be loaded when event occurs
- Can applications handle some of the interrupts?
  - Possible, only if OS allows it

Entry into ring-0: Software caused faults



- Why OS should handle it?
  - Fault isolation
  - Recover from error conditions
  - Enhance resource utilization!
  - Examples: Divide Error, Page Fault
- How OS handles it?
  - Handling is same as interrupts, but actions depend on the type of fault

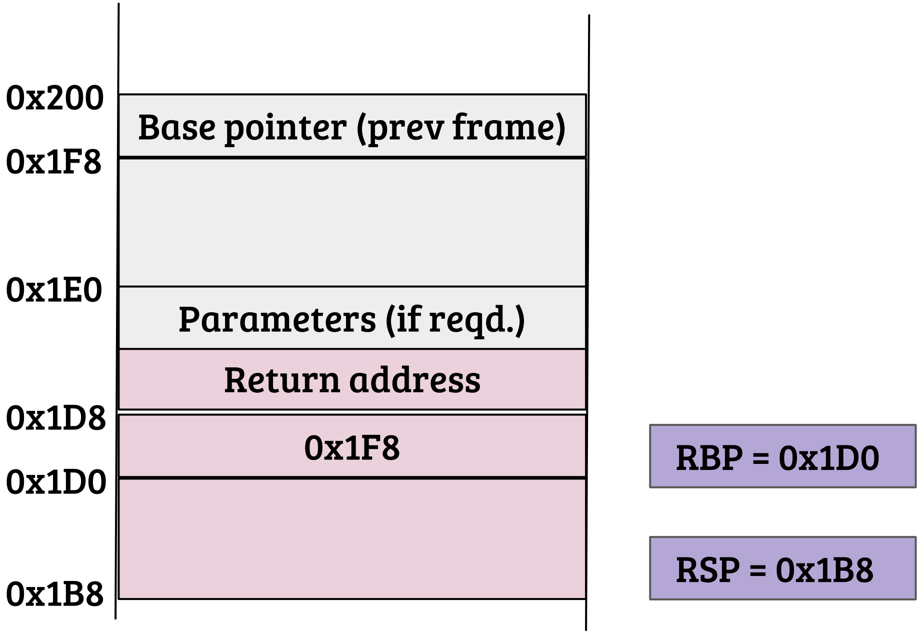
X86 background: Architectural support for applications

X86 support for subroutines/functions

```
int square (int num)
{
  ………………
}
int find_sum_squares (int len, int *arr)
{
        int sum  = 0;
        while(len){
            sum += square (arr[len-1]);
            --len;
        }
        return sum;
}
main( )
{
    int sum = 0, len = 5;
    int arr[5] = {1, 2, 3, 4, 5};
    sum = find_sum_squares (len, arr);
    printf("%d %d\n",len,  sum);
}
```

- Output of the program?
- Maintain variables with local scope
  - Nested calls
  - Can be recursive
- Alternate 1: Use registers
- Alternate 2: Use separate memory area for each active call
  - How?
  - Use a stack
- X86 enables a stack operable through instructions

RSP and RBP registers



- ➔ RSP → Current stack pointer
- ➔ Stacks grow towards the lower address
  - ◆ PUSH decreases the value of RSP
  - ◆ POP increases the value of RSP
- ➔ RBP → Current frame base pointer
- ➔ Implicit stack use during CALL and RET
  - ◆ CALL pushes return address onto stack
  - ◆ RET pops the return address from the stack and updates the RIP

Useful information for mixed code (C + ASM)

- Argument pass order: RDI, RSI, RDX, RCX, R8, R9 (if passing integers or pointers)
- Return value is generally in RAX (if return value is integer or pointer)
- Callee saved registers → RSP, RBP, RBX, R12-R15
    - If you intend to use them in ASM, preserve them
    - If you are calling a C function from ASM, save the register values
- More info can be found @ https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
- Suggest do the homeworks, should help you getting familiar

What is an application from CPU perspective?

- ➔ Is there any notion of an application from the architectural point of view?
    - ◆ Direct or Indirect correlation
    - ◆ Any basic building blocks?

What is an application from CPU perspective?

- ➔ Is there any notion of an application from the architectural point of view?
  - ◆ Direct or Indirect correlation
  - ◆ Any basic building blocks?
- ➔ Yes, in X86 it is called “TASK”

“*A task is a unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility.* ” ---Intel Software Developer Manual 3A, Ch7

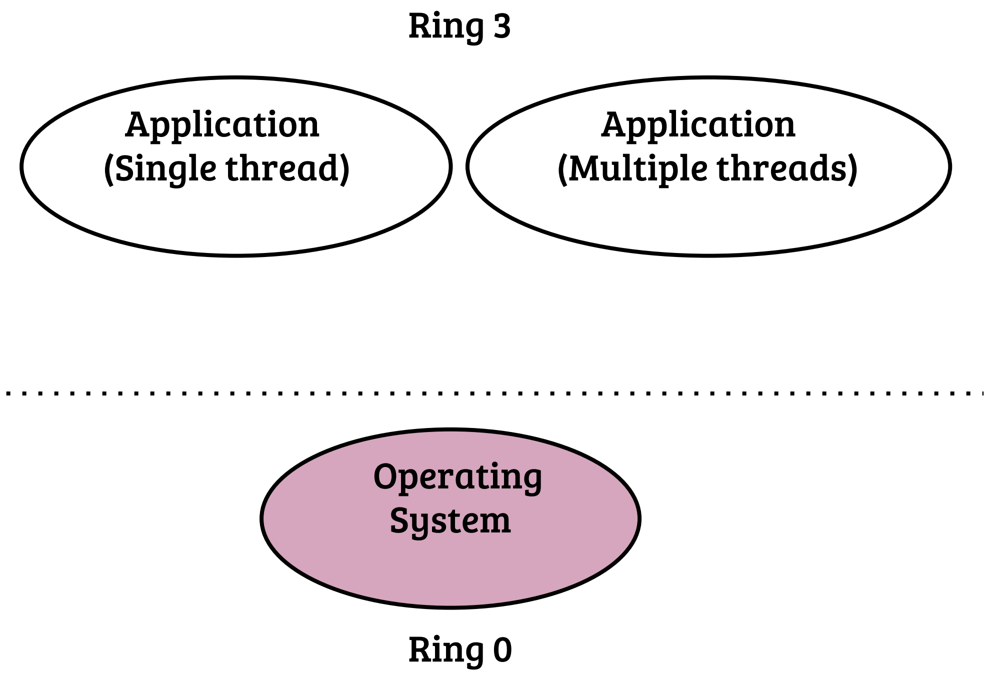
- ➔ For CS330, we will refer the hardware task state as an execution context

Important aspects of an execution context

- State of GPRs
- State of FLAGS, RIP → Current execution state
- CR3 → Memory partitioning information
- Current execution space (CS, SS, DS) → Defines privilege level
- Stack pointers for ring (0 - 2) → useful when privilege level changes
  - Why change RSP when switch from user to OS?

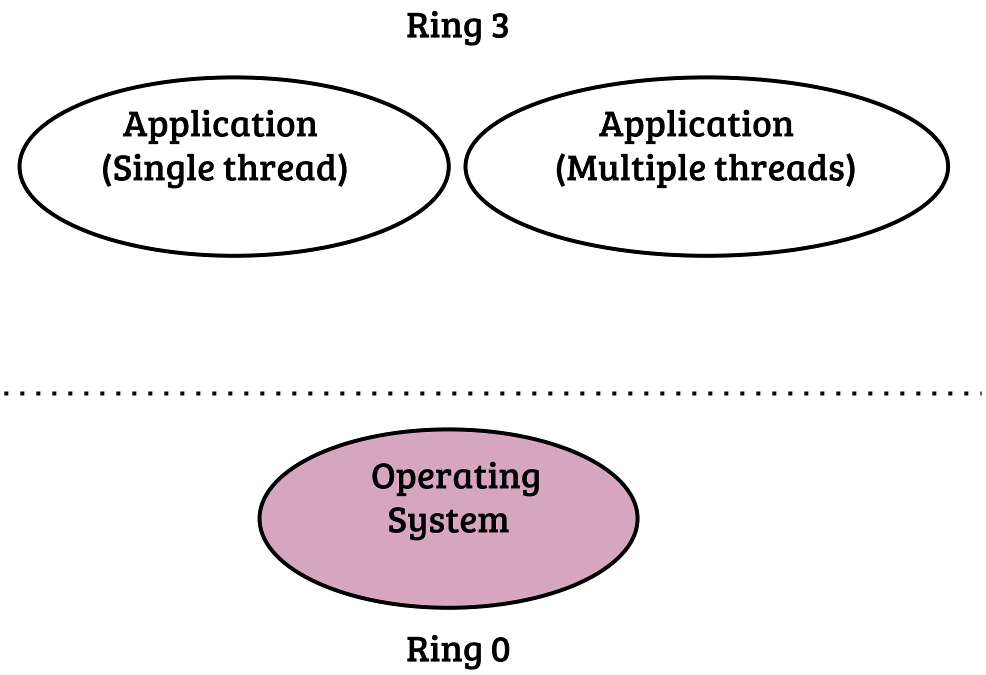
- In 32-bit X86, switching was done in hardware, but in 64-bit state is maintenance and switching is mostly in software (except for RSP and segment registers)

Execution contexts and stacks: Application



- How many stacks in OS?
  - Single threaded
  - Multi threaded
- How many stacks in kernel space?
  - Alternate 1: One OS stack
  - Alternate 2: One OS stack for each user thread

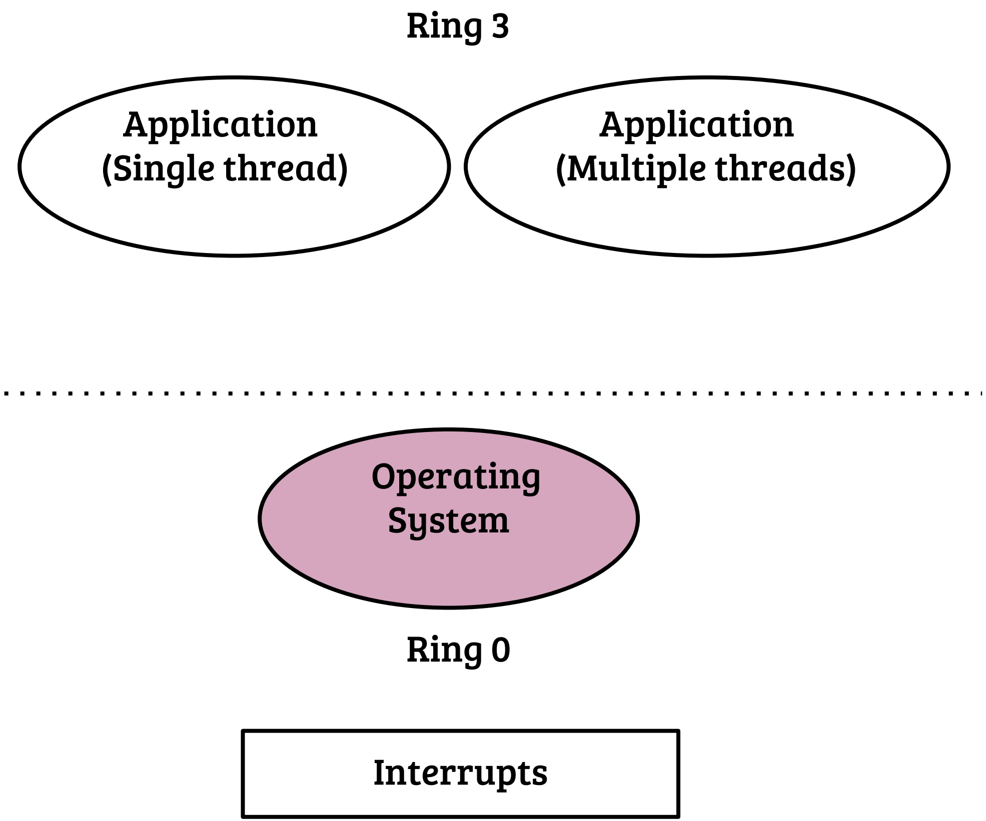
Execution contexts and stacks: Exceptions



➔ How many stacks in OS?

- Alternate 1: One OS stack
- Alternate 2: One OS stack for each exception
- Alternate 3: One OS stack for each user thread

Execution contexts and stacks: Interrupts



➔ How many stacks in OS?

- Alternate 1: One OS stack
- Alternate 2: One OS stack for each user thread
- Alternate 3: One OS stack for each interrupt
- Alternate 4: One OS stack for each CPU