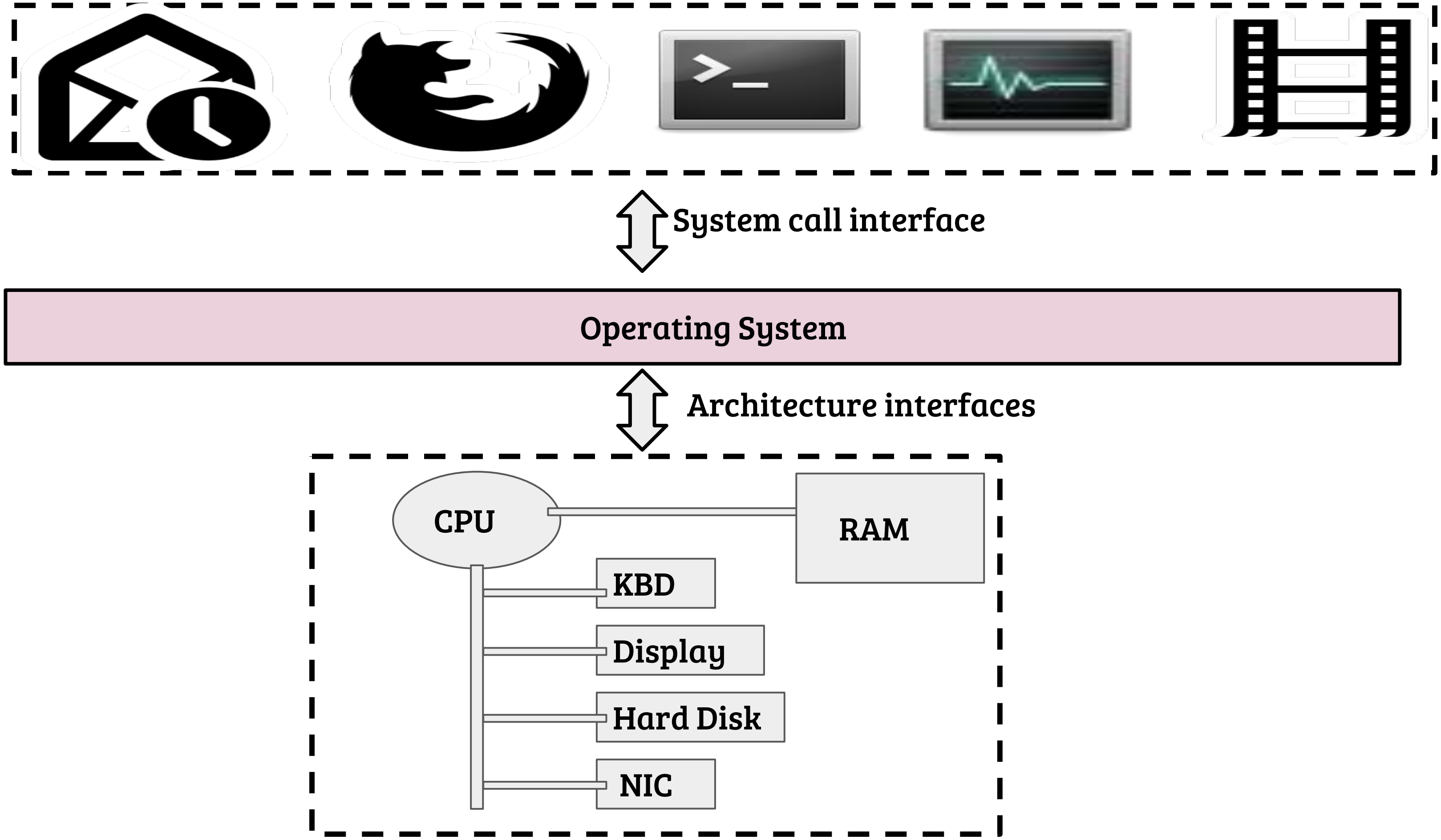
Operating Systems

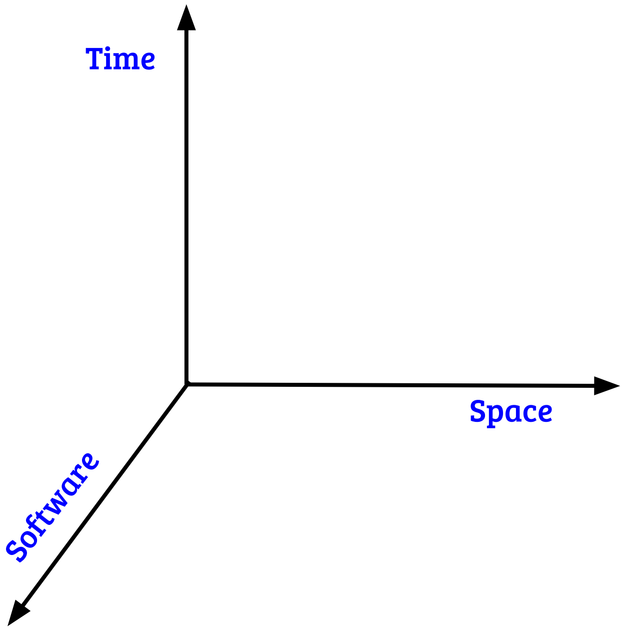
Resource multiplexing

Debadatta Mishra, CSE, IITK

Resource sharing: Multiplexing/Virtualization

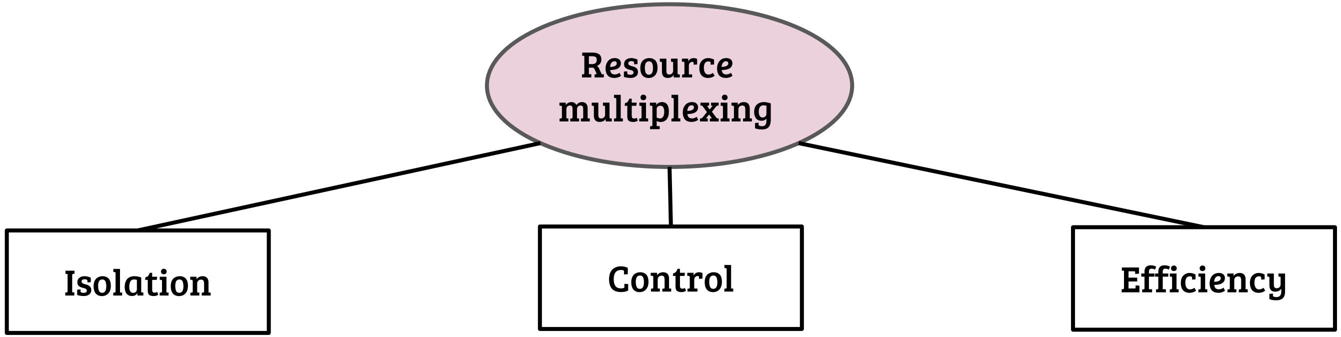


Multiplexing/Virtualization mechanisms



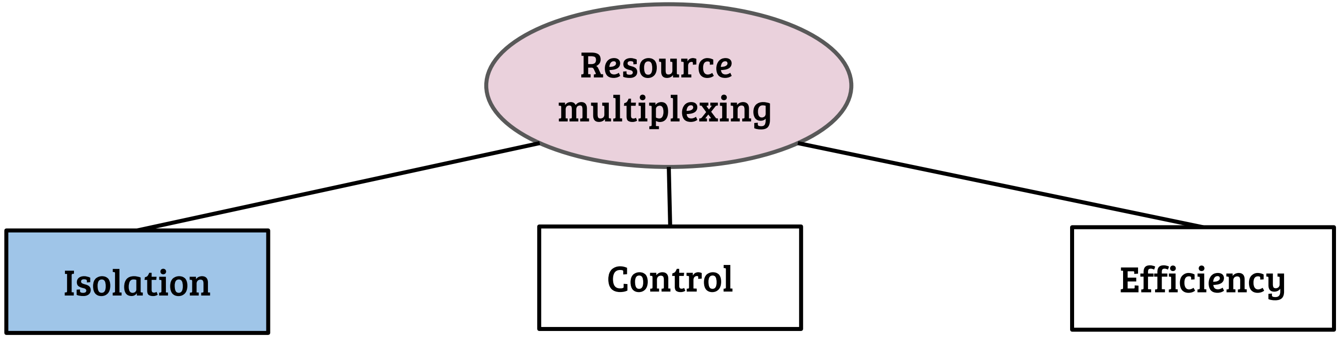
- Time sharing
  - A resource is allocated to different applications at different times
  - Resource should support "visible state" along with operations like "save" and "restore"
  - Example: a single CPU
- Space sharing
  - Resource can be partitioned into smaller units. Example: Memory
- Software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

Multiplexing/Virtualization requirements [1]



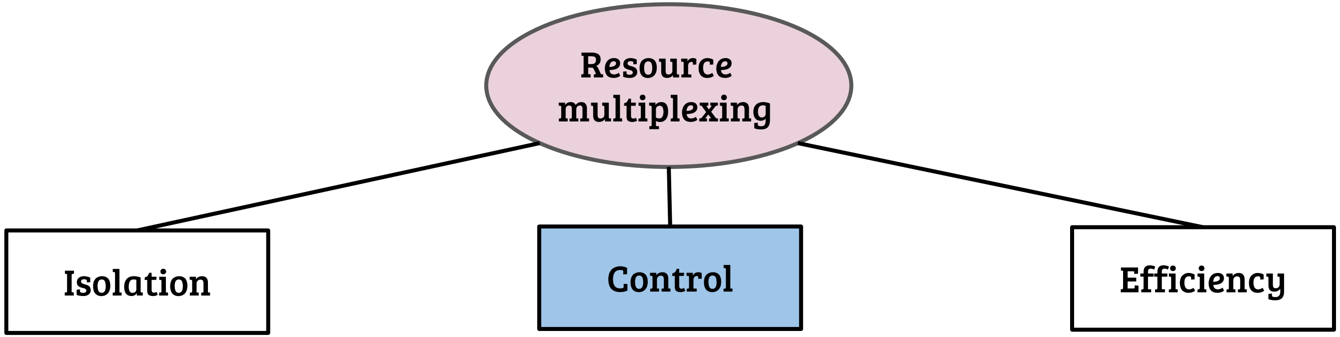
1. G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Commun. ACM 17 (7) (1974) 412–421

Multiplexing/Virtualization requirements



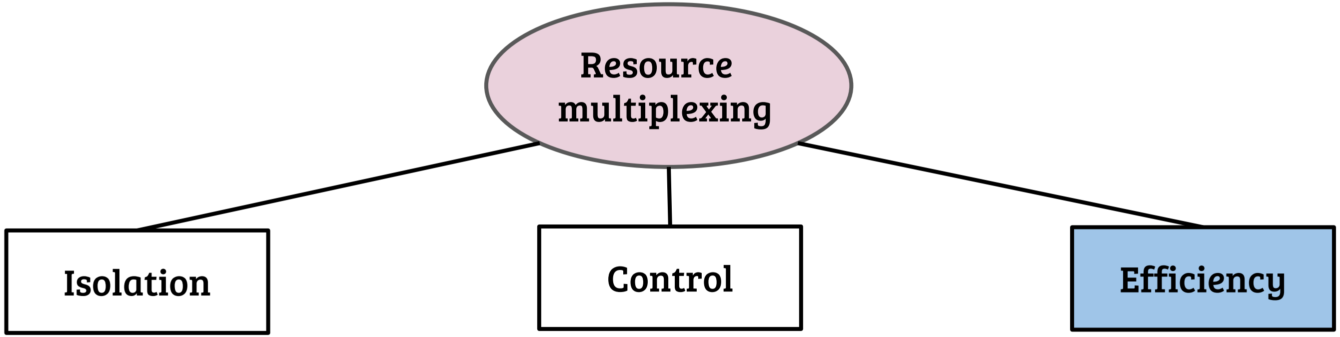
- Resources when used by one application (say A) should not be accessible from other applications, if not specifically allowed by A
- Alternate 1: All accesses to resources are through the OS (CPU?)
- Alternate 2: Resources are partitioned, but the “partitioning operations” are accessible only by the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

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

Multiplexing/Virtualization requirements

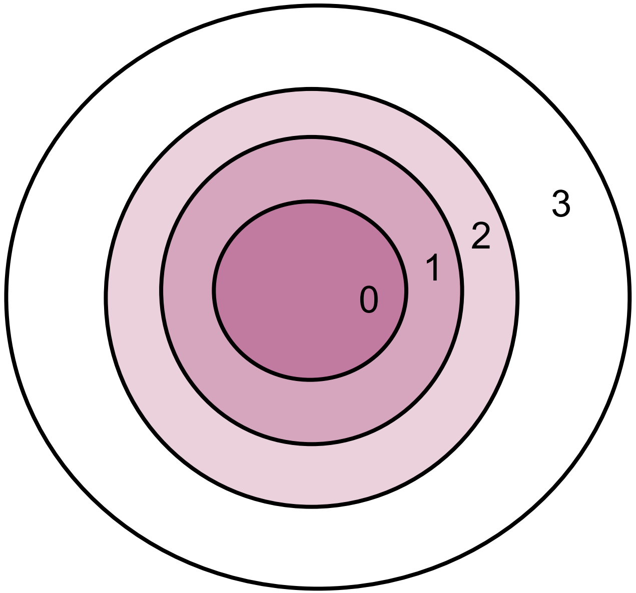


- Applications should use the resource directly → without OS intervention
- All accesses to resources are through the OS, not efficient :(
- How to apply restrictions to direct access (required for isolation and control)?

Limited direct access

- What to limit?
  - Instructions, Operands or Both
- Where to limit?
  - Hardware, Software or 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
- Linux uses 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);
}
```

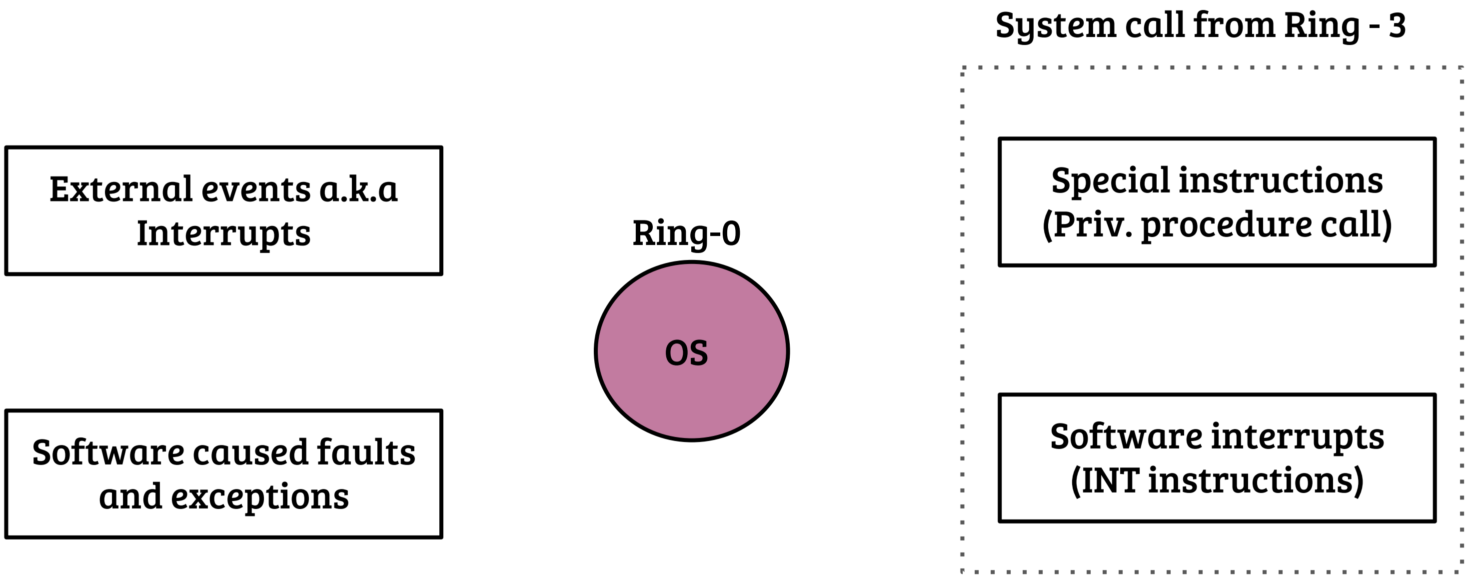
- Read CR3 register
- Executed from user space ⟶ Protection fault
- We are using “mov” instruction, but the operand is “privileged”

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)
        );
}
```

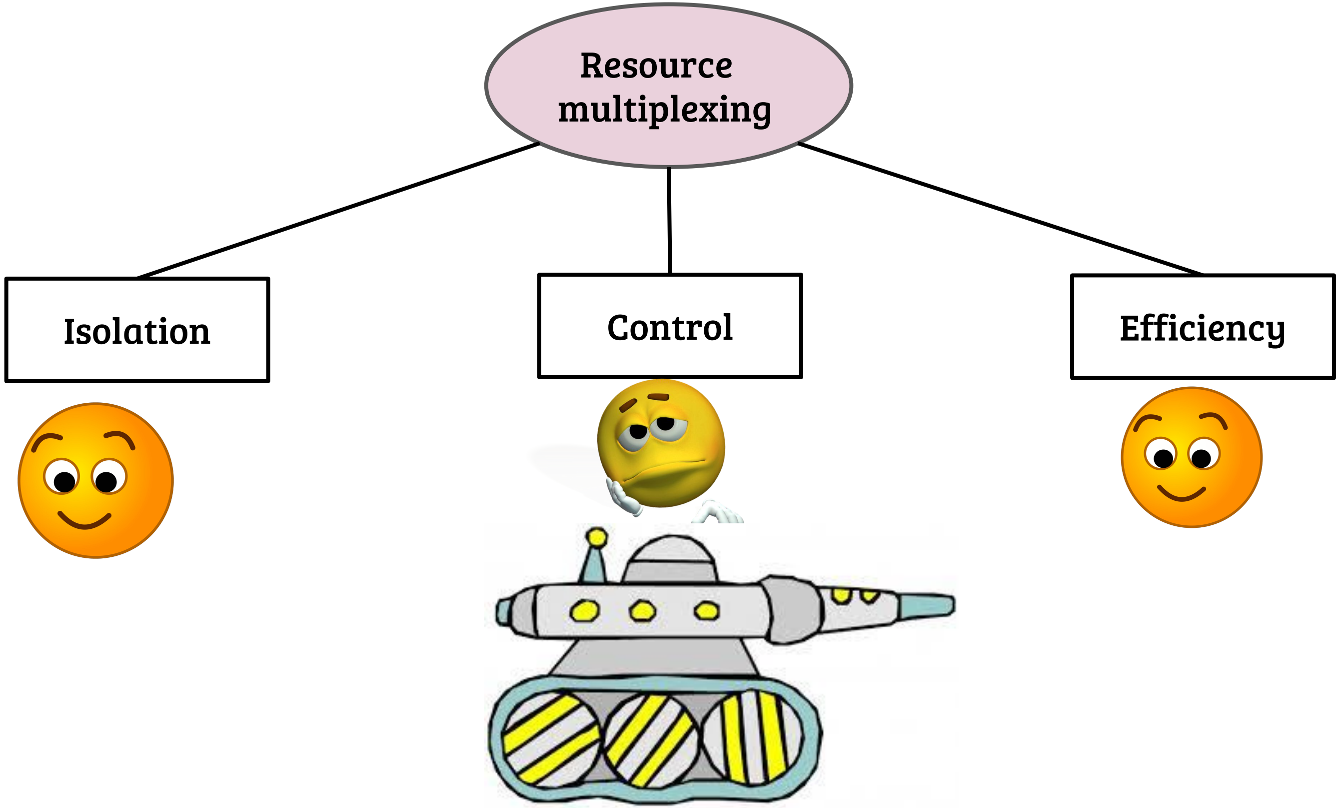
- Reading the content of code segment register CS (using MOV) is allowed
- Direct write to code segment register CS (using MOV) is not allowed

Entry into ring-0: necessary evils!



- Why necessary?

Virtualization requirements



Architectural primitives for limited direct execution