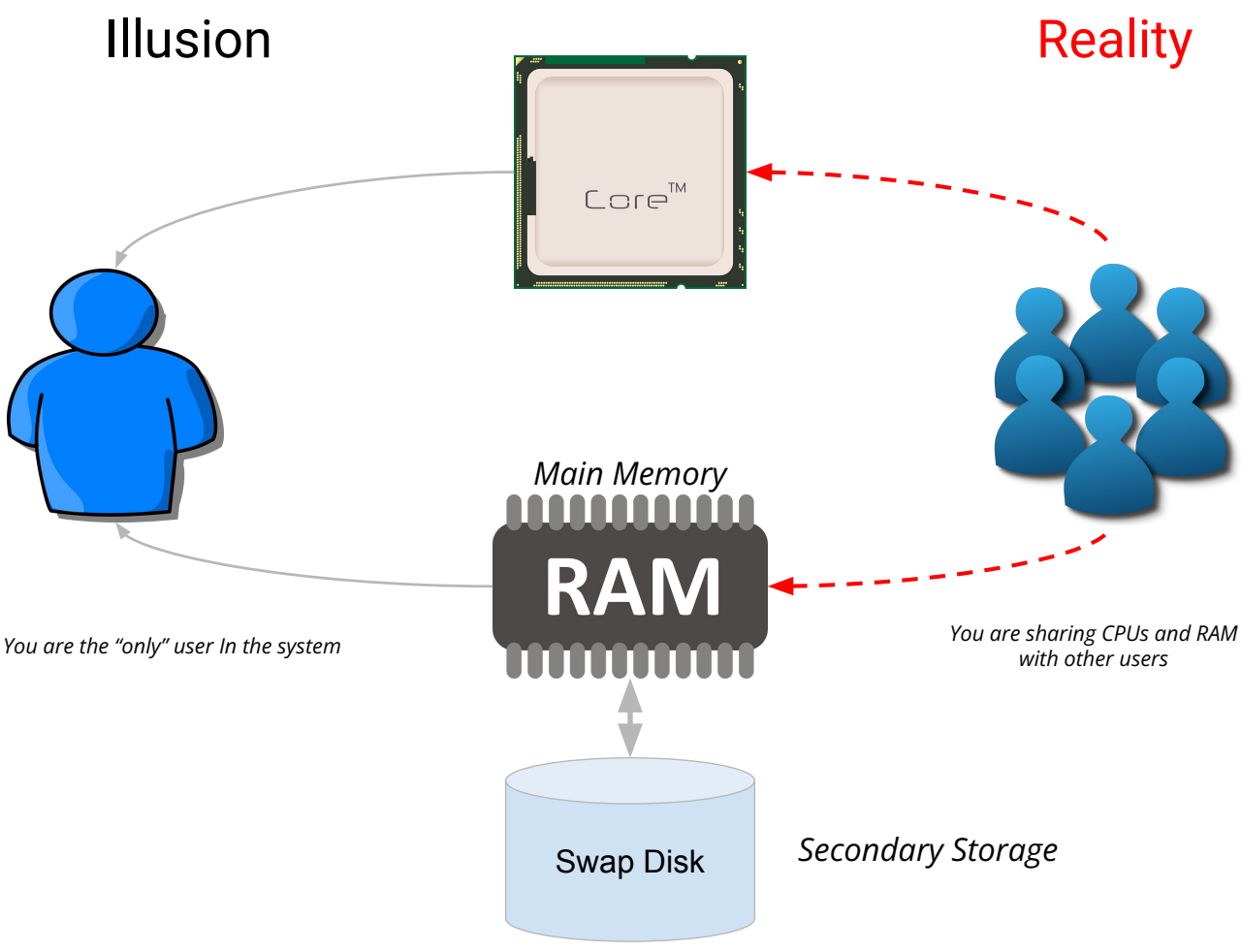
Memory Management



(Main) Memory ≠ (Secondary) Storage

32 Gigabytes (faster)

1 Terabyte (slower)

OS: The CPU Illusionist & Memory Illusionist

Create an illusion of

- Each user program runs on the CPU(s) **all the time**
- Each user program owns the CPU(s) **solely** to him/herself

Create the illusion of

- A process resides in RAM **all the time**
- A process owns the **entire RAM** to itself
- The code (generated by the compiler) starts at **address "zero"**
- Code and data are **contiguous** in memory
- A process has access to (**tera|peta|exa)bytes*** of RAM space

OS: Memory Management Facts

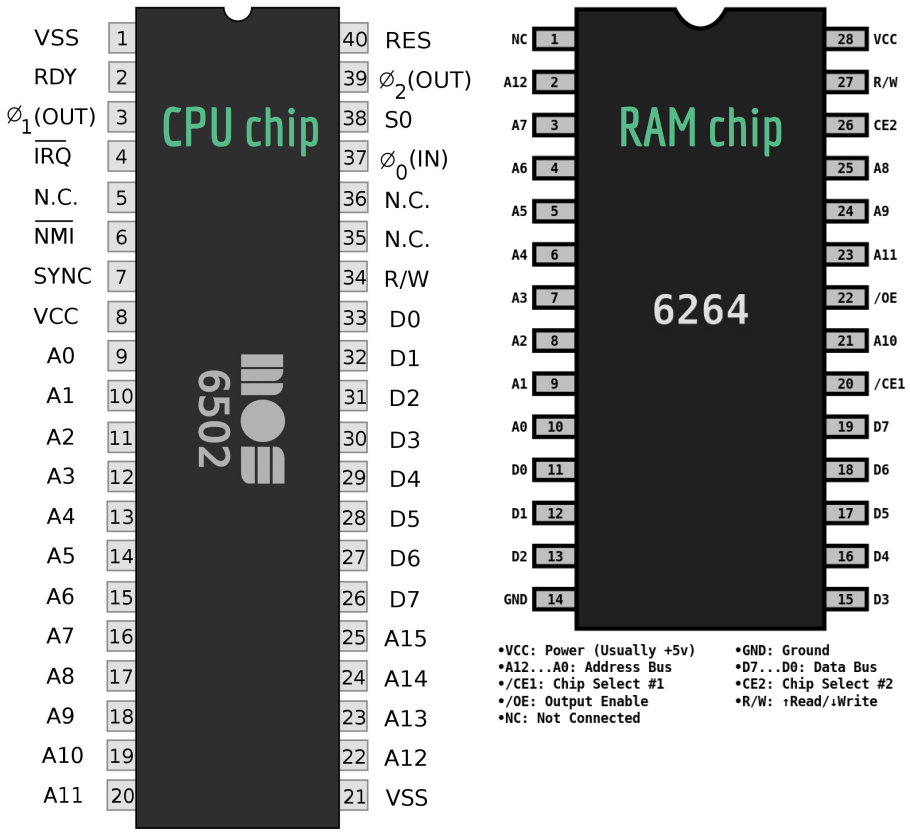
- Myth: A process owns the **entire RAM** to itself
  - **Fact: your process has to share the RAM with many other processes**
  - OS required feature: Memory Protection and Sharing
- Myth: A process resides in RAM **all the time**
  - **Fact: at times a process may be swapped out of RAM**
  - OS required feature: Process Relocation
- Myth: The code starts at **address "zero"**
  - **Fact: The code may be loaded (and reloaded) at any memory address**
  - OS required feature: load a process to a free memory region
- Myth: Code and data are **contiguous** in memory
  - **Fact: Code and data may be split into segments/pages**
- Myth: A process has access to **terabyte*** of memory space
  - **Fact: the actual amount of space accessible to a process is the RAM size**

System calls issued by a user process must be handled by the OS.

Software interrupts issued by the process allows the OS to intercept every system call

Accesses to RAM by a user process are issued *directly* to the memory hardware.

*The OS **does NOT** intercept memory accesses*
BUT invalid memory access must trigger hardware interrupt!



Ax: address pins
Dx: data pins

Memory read:

data = RAM[addr];

Memory write:

RAM[addr] = data

Required Hardware Feature #1:

Memory unit must trigger a hardware interrupt on invalid memory access

Fact #1: a process must **share** the RAM with other processes

Memory Speed Hierarchy

Type of storage	Access Time (Relative to CPU speed)	Storage Capacity
CPU registers	1x	bytes
Cache	up to 100x	Kilobytes - Megabytes
RAM	up to 1000x	Gigabytes

Memory Sharing and Protection

- Several processes may concurrently reside in RAM
  - Each process has different size
  - Each process is placed at different memory location
- Sharing: divide the RAM into regions, place a process in one region
- Protection: prohibit one process to peek into other's region
  - Protection violation must be (**initially**) *detected/reported by the memory hardware*, and (**later**) handled by the OS
- Two special registers:
  - **Base register:** the starting address of a region
  - **Limit register**: size of the region
  - These two registers are saved/restored during a context-switch

Address Binding

- A user program (in C/C++/Java/...) refers to data/variables/functions using **symbolic names**
- Compile-Time Address Binding
  - **Compiler** and **linker bind** each symbolic name to an address
- Load-Time Address Binding
  - The OS loads the binary executable to an open space in RAM
- Run-Time Address Binding
  - A process may be relocated at a different address
  - A process may call functions shared among several processes (the actual location of these functions must be resolved at runtime)
    - Windows (`.DLL`) and Unix Shared Objects (`.so`)

Compiler Explorer

http://godbolt.org

From [C/C++/Java] to Process in RAM

```
int num;

int main() {
  num = 0xBEEF;
  while (num != 0);
  return 0;
}
```

```
4004f0:  55                    push  %rbp
4004f1:  48 89 e5              mov   %rsp,%rbp
4004f4:  c7 45 fc 00 00 00 00  movl  0,-4(%rbp)
  ; num = 0xBEEF
4004fb:  c7 04 25 30 10 60 00  movl  $0xBEEF,0x601030

  ; while (num != 0);
400506:  81 3c 25 30 10 60 00  cmpl  0,0x601030
400511:  0f 84 05 00 00 00     je    40051c
400517:  e9 ea ff ff ff        jmpq  400506

  ; return 0;
40051c:  b8 00 00 00 00        mov   0,%eax
400521:  5d                    pop   %rbp
400522:  c3                    retq
```

Bin-executable is produced by linker

```
4004f0: 55 48 89 e5 c7 45 fc 00 00 00 00 c7 04 25 30 10
400500: 60 00 81 3c 25 30 10 60 00 0f 84 05 00 00 00 e9
400510: ea ff ff ff b8 00 00 00 00 5d c3
```

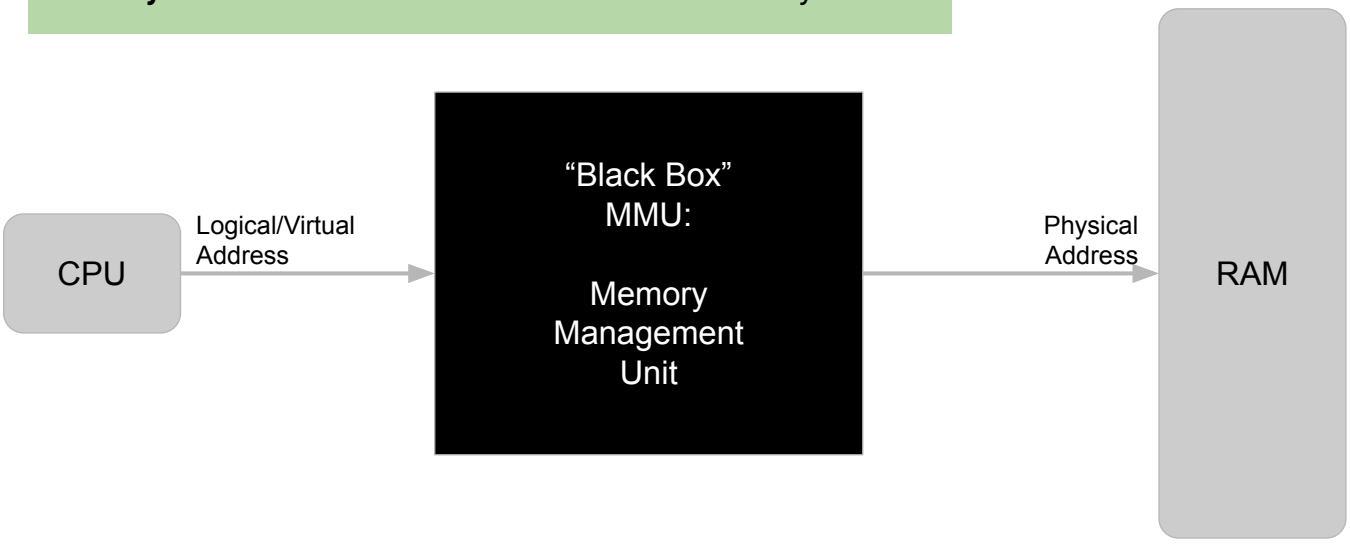
Bin-executable is LOADED into RAM by OS

***Somewhere** in RAM*

```
55 48 89 e5 c7 45 fc 00
00 00 00 c7 04 25 30 10
60 00 81 3c 25 30 10 60
00 0f 84 05 00 00 00 e9
ea ff ff ff b8 00 00 00
00 5d c3
```


Mapping Logical Add. to Physical Addr.

- **Logical/Virtual Address**: address generated by the CPU
- **Physical Address**: address issued to the memory



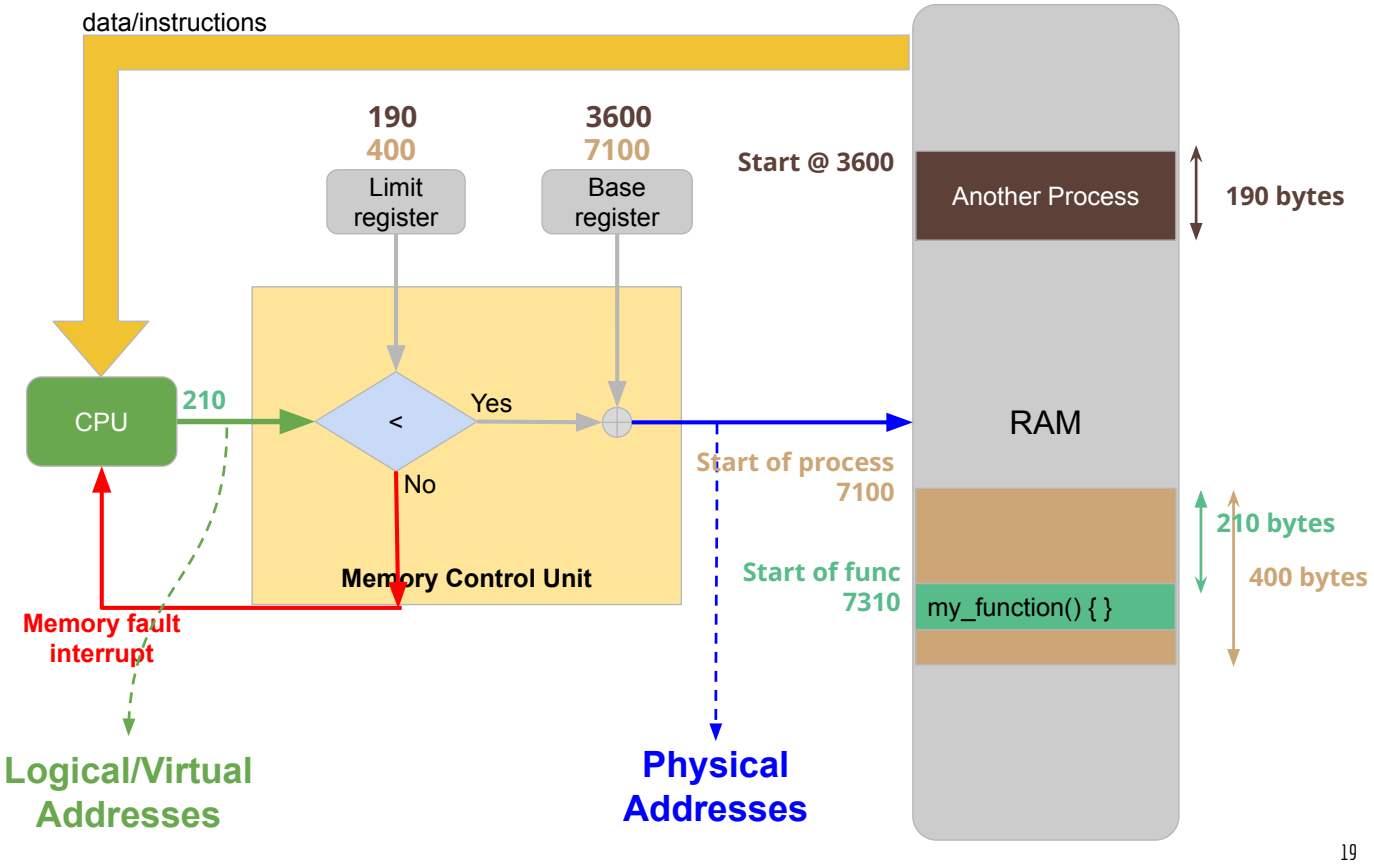
Required Hardware Feature #1:

Memory unit must trigger interrupt (memory fault) on all attempts to access invalid address

Required Hardware Feature #2:

Memory unit must automatically translate logical addresses to physical addresses

Base & Limit Reg (0-based Logical Addr)



Dynamic Loading/Linking

- Problem: Large Process Size, Limited RAM size
- Dynamic Loading ("Overlay")
  - Routines needed by a process are loaded on-demand
  - No special OS support needed, the user process is responsible for loading its overlay
- Dynamic Linking & Shared Libraries
  - Opposite variant of **static linking**
  - At linking time, the linker includes only a stub about the target functions in the shared library (PLT = Procedure Linkage Table)
  - The actual linking to the shared libraries are postponed until **runtime**
  - When a new version of shared libraries becomes available, the user program can invoke the newer version **without** recompilation/relinking

Runtime Binding (Windows DLL / Unix .so)

```
# Default: linker generates a dynamic executable
clang myprogram.c  -o myprog
gcc   myprogram.c  -o myprog
```

```
# Use -static to tell linker to generate static executable
clang -static myprogram.c  -o myprog
gcc   -static myprogram.c  -o myprog

# Show shared object dependencies
ldd myprog
```


Fact #2: a process may be swapped out of RAM

Fact #3: a process may be loaded (or reloaded) to any address (*different from assumed by the compiler*)