Mechanism of Executing Programs

- Initially, programs & data exist on disk (non-volatile) storage.
  - Usually in the form of files.
- When we execute a program, the OS loads it into RAM and allocates resources so that CPU can execute its instructions.
- Many programs are existing in RAM (including OS itself) and apparently executing in the same time. To be a Trusted Computing Base:
  - We need to protect RAM of each process & isolate its code and data.
  - We need to protect the files on the non-volatile storage from unauthorized access
  - We need to allow sharing of resource and maximize their utilization.

Protections in Operating Systems

- Each program (OS or applications programs) runs in the computer memory (RAM) as a "***process***".
  - A process is an abstract entity representing a *running* program.
  - It is the unit to which resources are allocated.
  - The OS itself runs as one or more processes.
- These processes share the physical resources such as CPU, RAM, Disk, Network, ...
- The OS provides "***isolation***" and "protection" of each process with the illusion that each process has ALL the resources (virtualization)
- The OS also provide sharing services if some processes request this.

Protections in Operating Systems

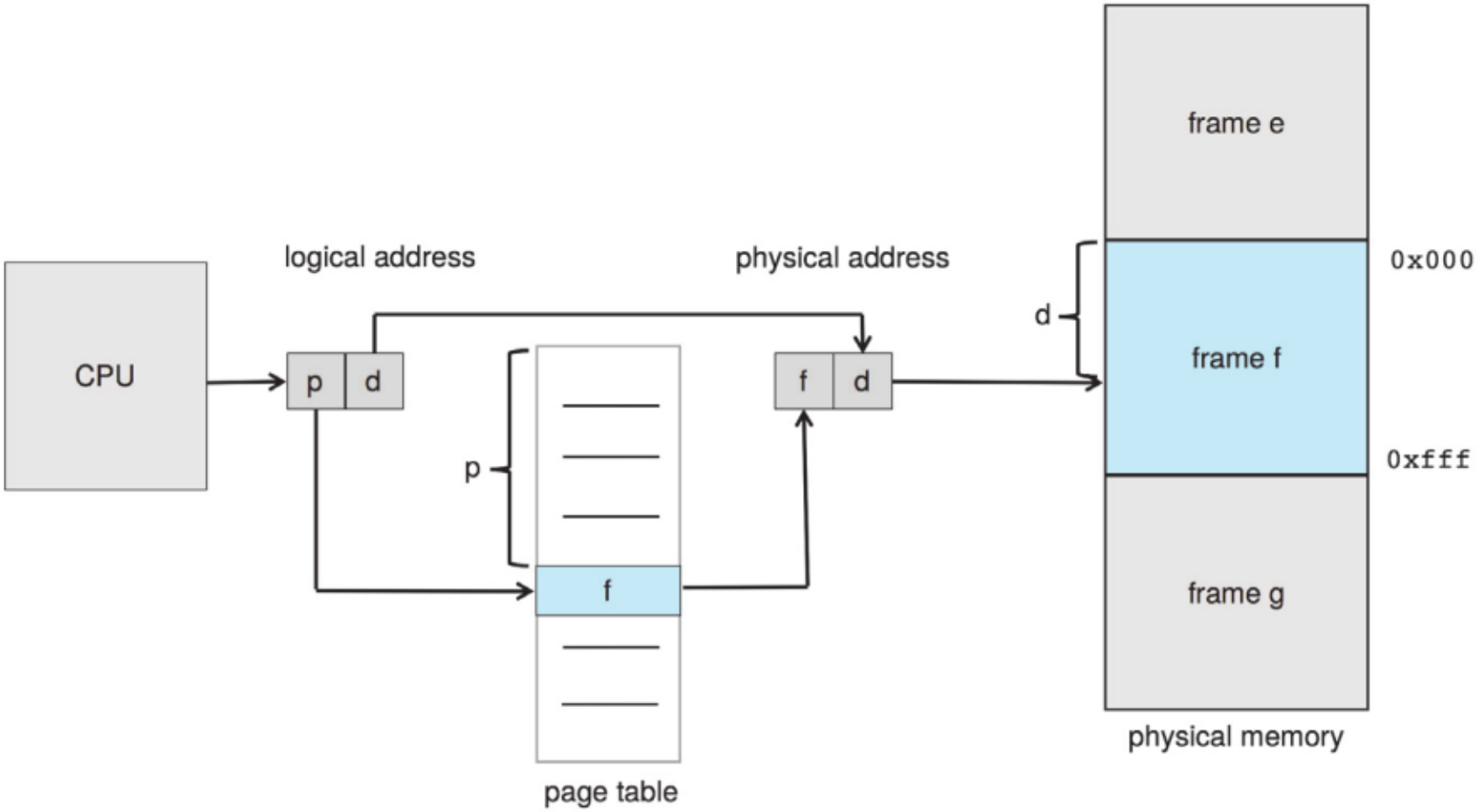
- The CPU executes in different **execution modes** (also known as **execution rings**), which serve as execution environments with varying privileges.
  - The most privileged ring is ring 0 (Kernel mode), and privileges are revoked as moving up to a higher ring.
- The OS runs in "***Kernel mode***" that allows it execute privilege & I/O instructions and protect its data structure and hardware and manage all processes.
- User applications run in less privilege "***user mode***" and cannot execute these privilege & I/O instructions.
  - These apps will call the OS (system call) to perform these operations on behalf of them.

Address Space

- **Address space** is an abstraction for all the memory locations that can -*virtually*- be reached by a process.
  - This is a contiguous array of memory locations. For 32-bit processor $=2^{32}$ and for 64-bit processors$=2^{64}$.
  - This "*virtual memory*" usually larger than the actual physical memory in the machine.
- With this abstraction, each process has its own virtual memory that effectivly isolate the process from all other processes.
- An address in address space are called "**logical address**" and must be translated to a "**physical address**" by some technique like *paging* and/or *segmentation*.

Address Translation - Paging

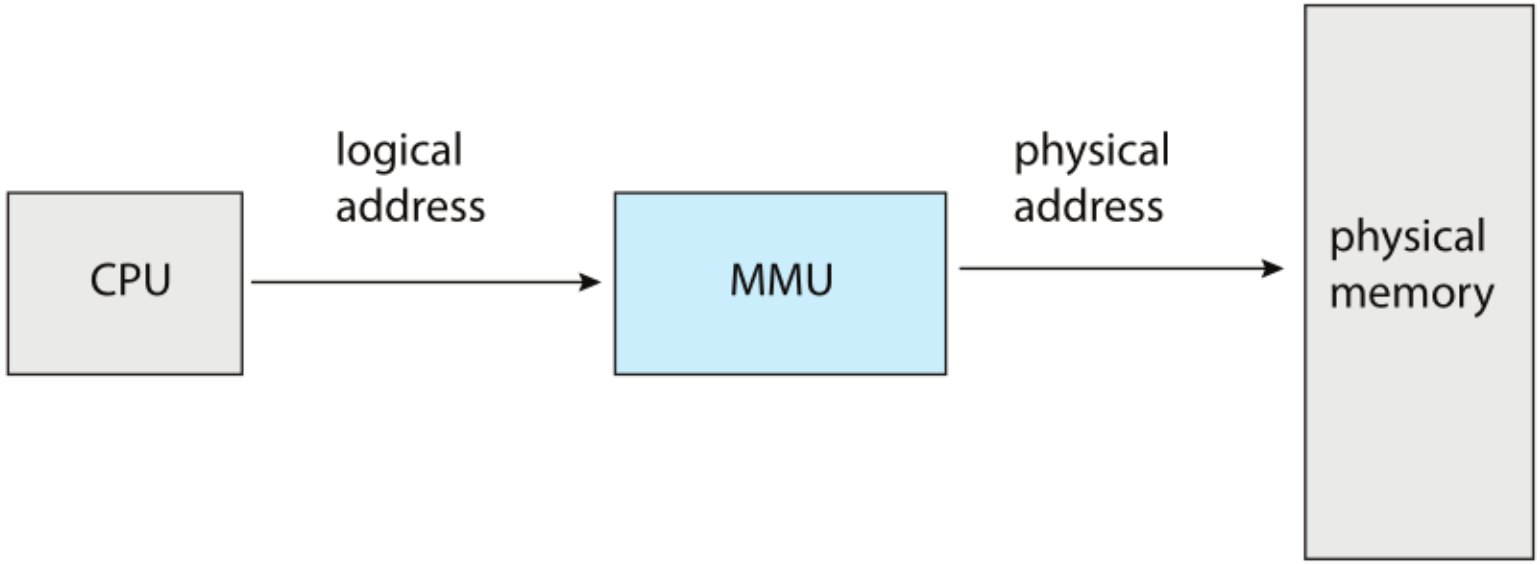
- The address space (*logical*) and the *physical* memory are partitioned into equal-sized parts.
  - Logical partitions are called “**pages**”, while physical partitions are called “**frames**”.
  - Logical address has the format (page#, offset)
  - Physical address has the format (frame#, offset).
- The OS will construct & maintain a mapping table that record the physical frame# for each logical page#.
  - Hardware support is necessary. e.g. MMU & Page fault interrupt.
- Notice that not every logical page# needs to have a physical frame. We can save many pages on the disk and bring them back when needed.



Address translation in simple paging

Fig. From Abraham-Silberschatz-Operating-System-Concepts-10th-2018

Hardware Address Translation



Memory management unit (MMU).

Fig. From Abraham-Silberschatz-Operating-System-Concepts-10th-2018

Practice Exercise

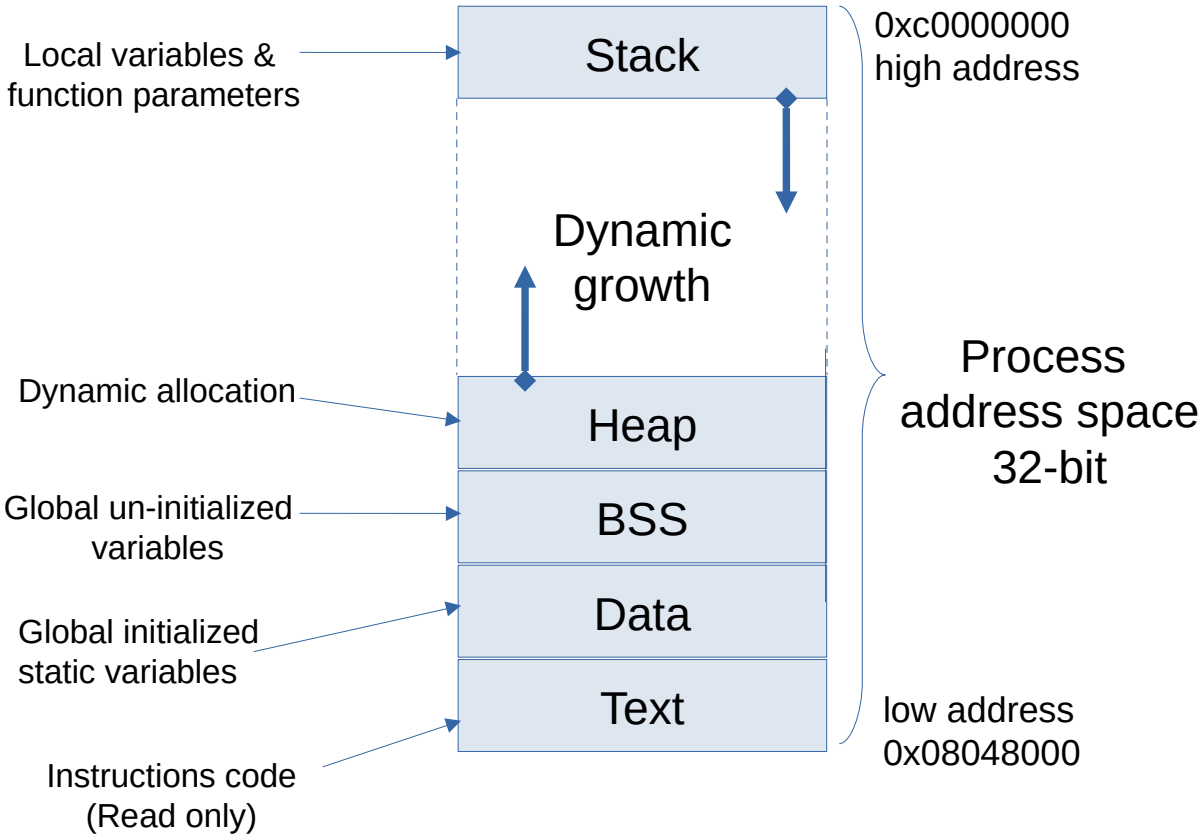
- Find the memory page size used in your PC/laptop.

Memory Management Protection & Security Issues

- Usually some bits in Page/Segment tables are used to indicate permissions to Read/Write/Execute.
- Using logical address space and address translation mapping to physical memory provides effective isolation of processes. However, there are some security issues:
  - The tables used in address translation must belong to the OS's memory so that the user cannot modify them.
    - If the user can modify the tables there is no isolation.
    - Special privileged instructions are used to modify MMU tables.
  - System calls must be done through “call gates” that ensure proper context switching and exiting.
    - In x86 architectures, we have explicit instructions to cross the system boundary: `sysenter` to enter the operating system during the system call and `sysexit` to return from the system call.
  - Systems call are far more expensive than regular procedure calls.

Process Memory Layout

- When a program is launched, the binary code is loaded into memory and the OS creates a new process, initializes the registers, and partitions the address space into the following segments:
  - Text segment: Contains the binary instructions that are executed by the CPU.
    - Usually is read-only.
  - Data segment: Contains *global initialized* and *static* data resides in the data segment.
  - Block Started Symbol (BSS) segment: *Global uninitialized* variables
    - The OS generally assigns such variables a value of 0 before handing control over to the process. However, they are still treated as uninitialized.
  - Heap: Provides dynamic memory during runtime. For example, dynamic memory management in C is performed with the functions malloc() and free().
  - Stack: is the place where *dynamic local* variables are put together with function *parameters* and process *control* information.



Executing process memory layout