# CS 61 Section Notes 5

*(Week of 10/22 - 10/26)*

**Topics:**


- Dangerous Instructions and Process Isolation
- Virtual Memory
  - Memory Mapping
  - Address Translation
    - Some numbers
    - Some Terms
- Processes and Fork
  - What is a process?
  - Context Switching
  - Process States
  - Fork


## Dangerous Instructions and Process Isolation

Dangerous instructions are instructions which can affect another process, affect the kernel, or prevent another process from running.

One of the key ways to solve the problems posed by dangerous instructions is through process isolation. If each process can only access its own memory (meaning it cannot access other process' memory or the kernel's memory), any dangerous instructions can be controlled.

Sometimes processes need to interact with one another, and the CPU may allow it under a very limited set of circumstances. But in general the idea of process isolation is that processes are limited to their own memory.

Process isolation necessarily requires hardware support. The hardware is the only thing removed enough from the processes to determine what code is *privileged* (such as the kernel, which should be given full access to all memory), and what code is *unprivileged* (the average process, which has limited access to memory).

**Q1: What are some examples of dangerous instructions?**

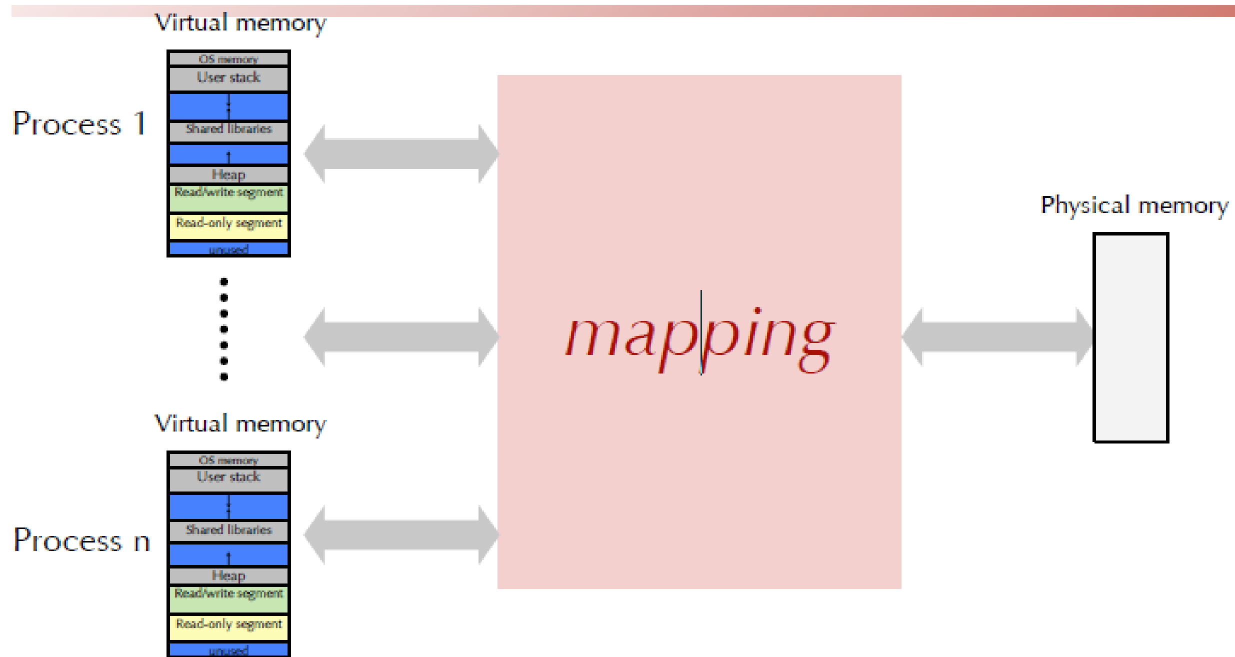
**Q2: What are some ways to (attempt to) control dangerous instructions with software?**

# Virtual Memory

Virtual addressing is a kind of abstraction which allows processes to work with the memory assigned to them without knowing anything about that memory's location on disk. Each process gets its own private memory space. Virtual addressing is a powerful tool which has multiple benefits:

- Memory Management
- Caching
- **Process Isolation**

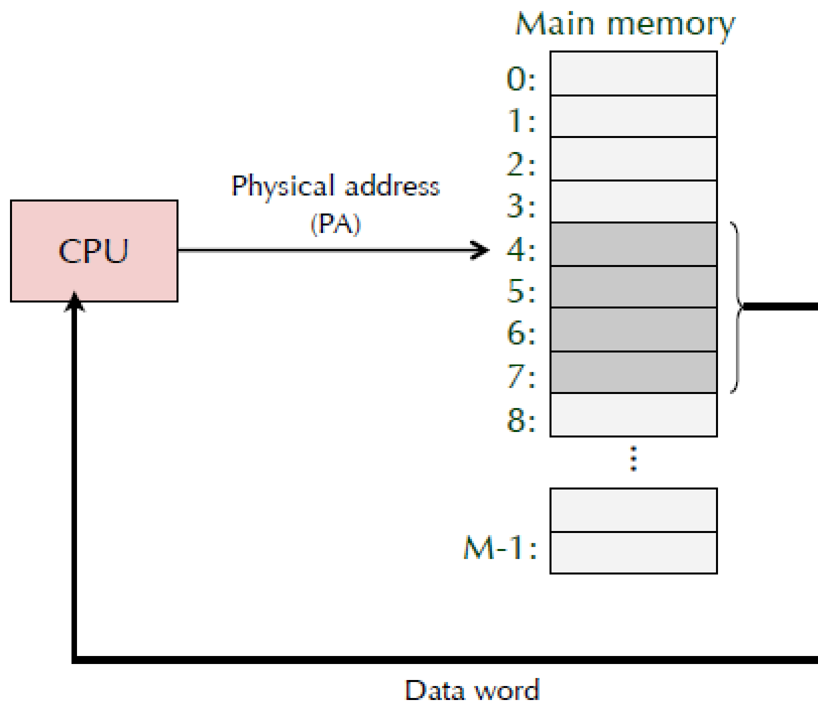
## Memory Mapping



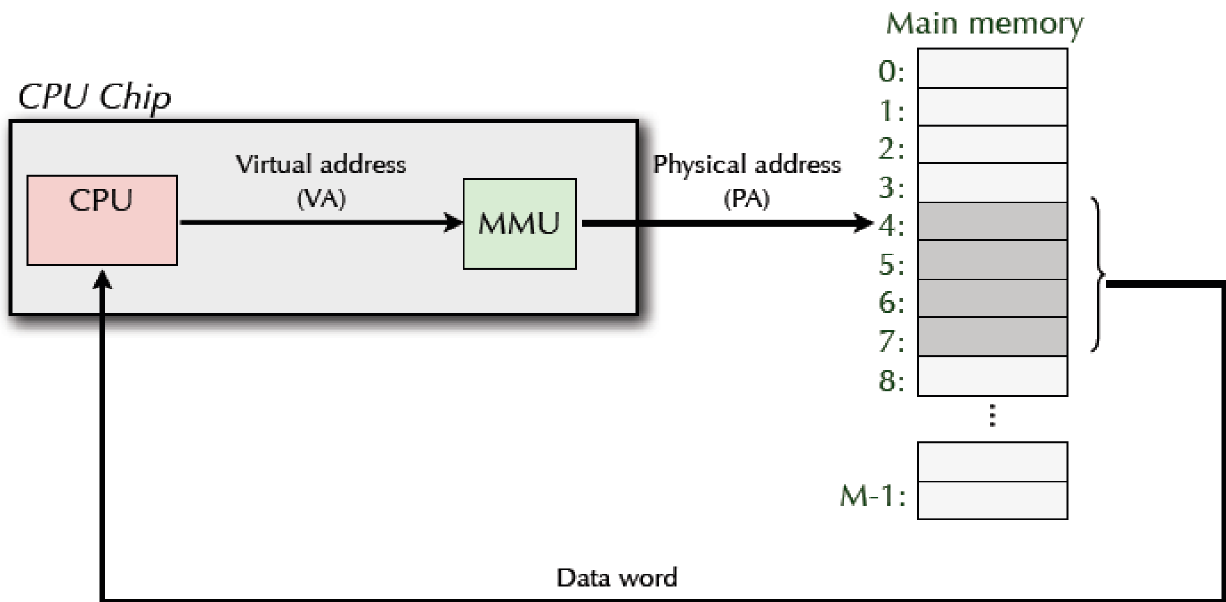

Each process believes it has a continuous stretch of memory, represented by the little memory frames on the left of the diagram above. This is not necessarily the case. In practice, the process' memory could be scattered across the disk, or even overlapping with another process' memory! All of this mapping is performed by a separate entity, which does not disclose any of its information to any of the processes. This gives us process isolation!

## Address Translation

Physical Addressing - Process asks for memory, CPU generates a physical address to a location on disk and passes it to the disk. The disk pulls up the data at that location and passes it to the CPU.




Virtual Addressing - Process asks for memory, CPU generates *virtual address*, the *Memory Management Unit (MMU)* translates the virtual address into a physical address and requests it from the disk, the disk pulls up the data and passes it to the CPU.



## Some numbers

Let us say we have a n-bit addressing system. Then our virtual address space is of size $2^n$.

Let us say that our physical address space is $2^m$. It follows that $m <= n$, or we couldn't address all of our physical space. For the most part, we will be dealing with 32-bit addressing space in this class.

We break down both our physical and virtual space into chunks known as *pages*. Let us say that our pages are $2^p$ bytes long. It follows that our virtual address space contains $2^{n-p}$ pages. We will typically deal with pages which are $2^{12}$ or 4KB in size. This means that we have $2^{20}$ pages of virtual memory.

## Some Terms

NOTE: Not all of these terms are consistent across various forms of literature.

Page - A chunk of memory, either virtual or physical, of a fixed size. We can access a particular byte by first indexing to the page it belongs to using the page number, then indexing within the page using the page offset, which tells us how many bytes into our page our target is.

Memory Management Unit (MMU) - a unit in the CPU responsible for interfacing between processes and the disk.

Page Table (PT) - a data structure stored in physical memory that maps virtual addresses to physical addresses. Maps virtual page numbers to physical page numbers.

Page Table Entry (PTE) - Elements within the PT. Each PTE is an n-bit address field, along with some metadata bits embedded in the least significant nibble.

- PTE_P - The 'valid' or 'present' bit. Marks PTEs which contain valid addresses
- PTE_U - Marks pages which can be accessed by ordinary processes, or 'users'
- PTE_W - marks pages which can be written to

Virtual address

- Virtual page number (VPN)
- Virtual page offset (VPO)

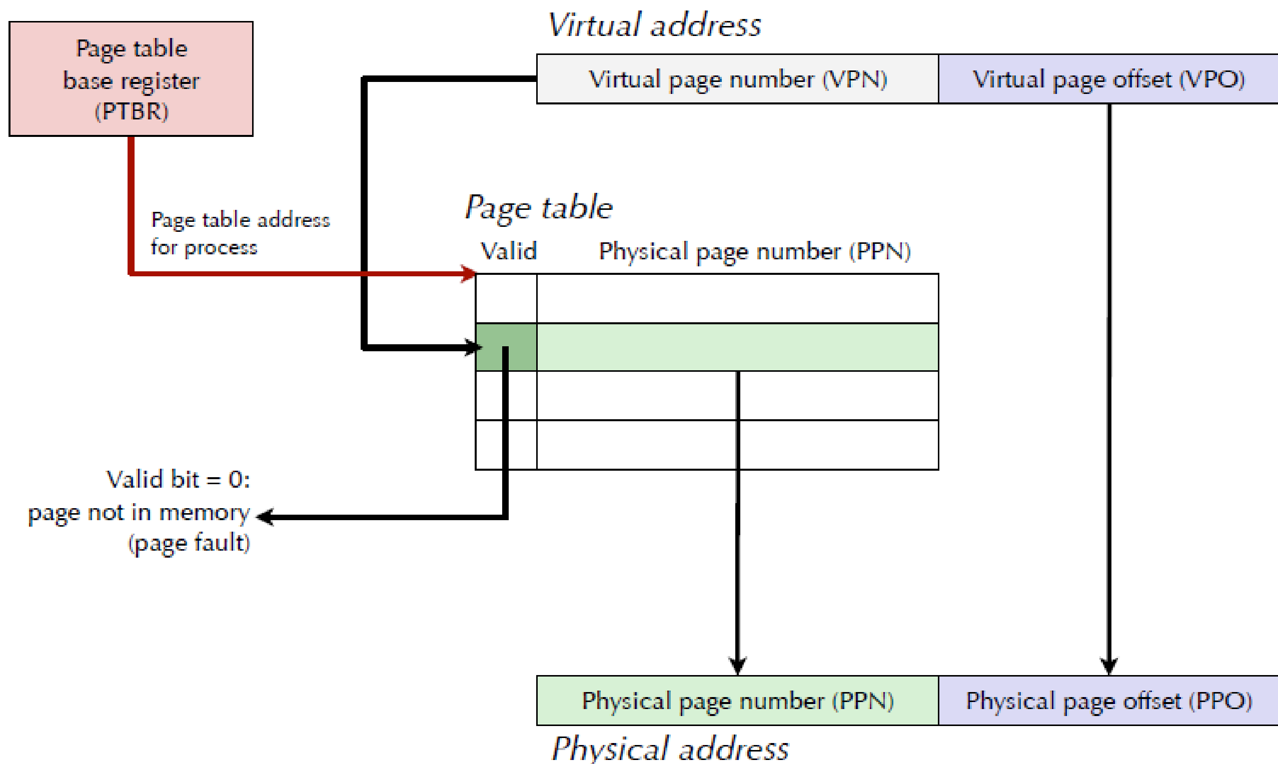
Physical address

- Physical page number (PPN)
- Physical page offset (PPO)

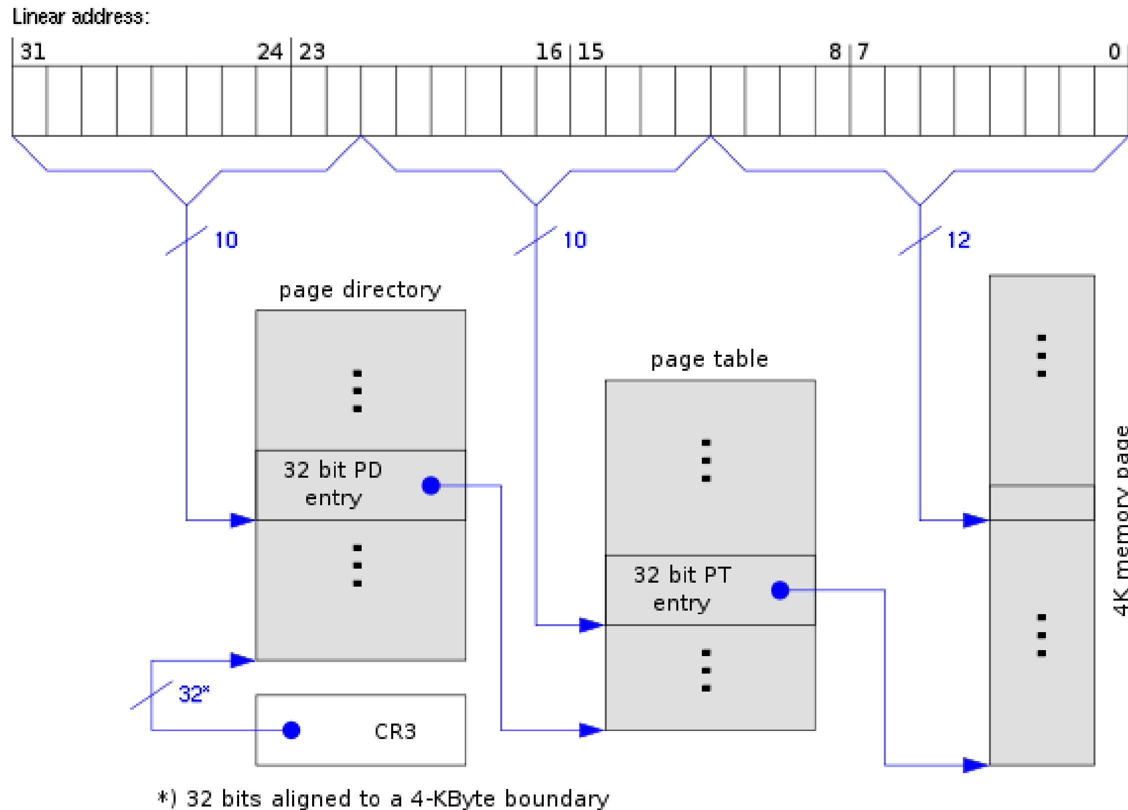
Address translation is the process of turning a virtual address into a physical address. In its simplest form, it involves 2 steps.

1. Look up the VPN in the Page Table. The entry at that index is the PPN
2. Use the VPO as the PPO

This makes sense if we think about it. Since physical pages are just as long as virtual pages, the offset should be the same no matter what page we're actually accessing on disk. However, the location of the page on disk is completely unknown to the process. So the PT maps the VPN to a PPN. It's really quite a simple process!




This is what we call a single level virtual memory. We have one page table which maps virtual addresses to physical ones. However, x86 uses a more sophisticated architecture, which we call two-level virtual memory. Instead of a single PT, we have a whole data structure of them, indexed by a Page Directory (PD). This PD is essentially a lookup table for the PTs, which of course are lookup tables for PPNs. Hence 'two-level'.




Our VPN is now split into two entities

- Page Directory Index (PDI)
- Page Table Index (PTI)

These will typically be 10 bits each, and serve as our indices for the 1st and second levels respectively.

NOTE: From an addressing perspective, this is almost the same as before. Looking up a 20 bit address in a lookup table is the same as looking up a 10 bit address in one lookup table, and then the other 10 bits in another lookup table a level down. In both cases, each unique virtual address maps to a PTE. The difference is how the PTs are stored in memory.

Imagine we are given a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 4-byte words
- Physical addresses are 16 bits wide.
- Virtual addresses are 20 bits wide.
- The page size is 4096 bytes.
- There are 4 page tables (and the size of the page directory index reflects this)

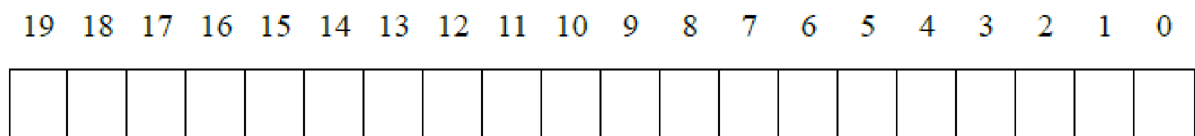
The Page Directory and one of the page tables look like this (all values in hex):

| Page Directory | |
|---|---|
| Page Directory Index | Entry |
| 0x0 | 0xF000 |
| 0x1 | 0xD000 |
| 0x2 | 0xE000 |
| 0x3 | 0xC000 |

| Page Table @ 0xE000 | | |
|---|---|---|
| Page Table Index | Entry | PTE_P |
| 0x2A | 0xA000 | 0 |
| 0x11 | 0x1000 | 0 |
| 0x1F | 0x8000 | 1 |
| 0x07 | 0x5000 | 1 |

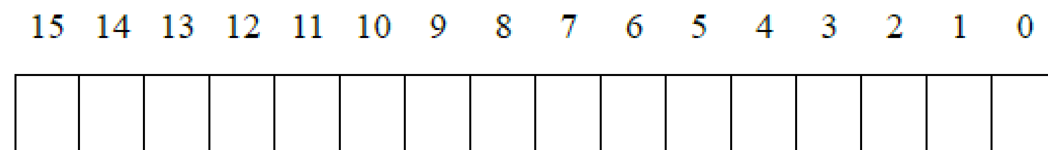
**Q3: The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields that would be used to determine the following:**

- **VPO**
- **VPN**
- **PDI**
- **PTI**




**Q4: The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:**

- **PPO**
- **PPN**




**Q5: Imagine we are given the Virtual address 0x9F37C.**

- **What is the Virtual Address in binary?**
- **What is the VPN?**
- **What is the PDI?**
- **What is the PTI?**
- **Do we page fault?**
- **What is the PPN?**

**Q6: Imagine we are given the Virtual address 0xAAA48**

- **What is the Virtual Address in binary?**

- **What is the VPN?**
- **What is the PDI?**
- **What is the PTI?**
- **Do we page fault?**
- **What is the PPN?**

**Q7: In x86, how much space must we allocate for page tables/directories in order to address 4GB?**

**Bonus Q: In x86, how much space must we allocate for page tables/directories in order to address 1B?**

# Processes and Fork

## What is a process?

- OS's abstraction for **execution** → instance of a program being executed
- A process gets a private address space
- **Why?**
    - **Virtual Memory!!**

Process **context** consists of the following:
- Address space
    - memory that can be accessed by the process
- Processor state
    - CPU registers
- OS resources
    - page tables

## Context Switching

- In general, if there is 1 CPU on your machine, only 1 process can run at a time.
- Multi-CPU machines = multiple processes.
- **How can we run multiple processes?**
    - Multi-CPU machines can run them in **parallel.**
    - Or, we could switch the CPU from one process to another. This is **context switching.**
    - A scheduler (this is the job of the OS) will decide how to switch.

**Q9: How do we context switch?** (Hint: What steps should we take to switch processes? Remember the definition of a process' context!)
- Each process will have a unique **process ID (PID)**

## Process States

- Each process will be given a state
- Ready - waiting for execution
- Running - executing on CPU
- Stopped - suspended
- Waiting, Sleeping, Blocked - waiting for something to happen
    - This is different from the ready state.

**Q9: Why is this?**

- Terminated - stopped permanently

How do we create a process?

- We use one process to create another one. This is **forking.**
- Thus, every process has a **parent process.**

**Q10: Wait! How does the first process get created?**

## Fork

- When fork is called, a new process is created with the its own private virtual memory.
- However, the child's address space is an exact copy of the parent's address space.
    - **Exception: The parent's eax is not copied over!** That is because eax stores the return value and we don't want both processes having the same return value (see below)
- **Fork returns TWICE!** Once in the parents, and once in the child.
    - Parent is returned the child's PID. Child is returned PID = 0.

**Q11: What do each of the following functions print?**

```
void fork1()
{
	printf("L0\n");
	fork();
	printf("L1\n");
	fork();
	printf("L2\n");
	fork();
	printf("Bye\n");
}

void fork2()
{
```

```
    printf(“L0\n”);
    if (fork() != 0)
    {
        printf(“L1\n”);
        if (fork() != 0)
        {
            printf(“L2\n”);
            fork();
        }
    }
    printf(“Bye\n”);
}

void fork3()
{
    printf(“L0\n”);
    if (fork() == 0)
    {
        printf(“L1\n”);
        if (fork() == 0)
        {
            printf(“L2\n”);
            fork();
        }
    }
    printf(“Bye\n”);
}
```