Unit: III

Lecture: 1

Memory Management

"Programs expand to fill the memory available to hold them…"

Parkinson's Law-A generalized view

Memory Hierarchy → computers have a few megabytes of very fast, expensive, volatile cache memory, a few GB of medium-speed, medium priced, volatile main memory and a few TB of slow, cheap, non volatile magnetic or solid state disk storage plus also DVDs and USB sticks.

The part of the operating system that manages the memory hierarchy is called the memory manager.

It job is:

- to efficiently manage memory,
- keep track of which parts of memory are in use, allocate memory to processes when they need and deallocate it when they are done.

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

The CPU fetches instructions and data of a program from memory → both the program and its data must reside in the main (i.e. RAM and ROM) memory.

- The management of lowest level of cache memory is normally done by the hardware so for a programmer, no need to worry about that instead this unit will focus on the programmer's model of main memory and how it can be managed.

Modern multiprogramming systems are capable of storing more than one program, together with the data they access, in the main memory.

A fundamental task of the memory management component of an operating system is to ensure safe execution of programs be providing:

- **Sharing of memory**
- **Memory Protection**

Memory Abstraction:

Exposing physical memory to processes has several major drawbacks:

- If user programs can address every byte of memory, they can easily trash the operating system, intentionally or by accident, bringing the system to a grinding halt.
- It is difficult to have multiple programs running at once.
  - On personal computers, it is common to have several programs open at once (a word processor, an email program, a Web

browser), one of them having the current focus, but the others being reactivated at the click of a mouse.

What if there is no memory abstraction?

Early computers (1960s), there were no memory abstraction. Every program simply saw the physical memory having a set of addresses from 0 to some maximum, each address corresponding to a cell contacting some number of bits (commonly eight).

e.g. an instruction like MOV REGISTER1, 1000

the computer just moved the contents of physical memory location 1000 to REGISTER 1.

- here, it was not possible to have two running programs in memory at the same time.
    - If the first program wrote a new value e.g. on location 2000, this would erase whatever value the second program was storing there. → nothing would work and both programs would crash almost immediately.

Running multiple programs without a memory abstraction:

Swapping: It is possible to run multiple programs at same time.

- The operating system will save the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts.

It was also possible to run multiple programs concurrently even without swapping with addition of some special hardware.

For example: In early models of IBM360:

- Memory was divided into 2KB blocks and each was assigned a 4-bit protection key held in special registers inside the CPU.

A machine with a 1 MB memory needed only 512 of these 4-bit registers (because 1MB =1024KB → so 1024KB/2KB = 512)

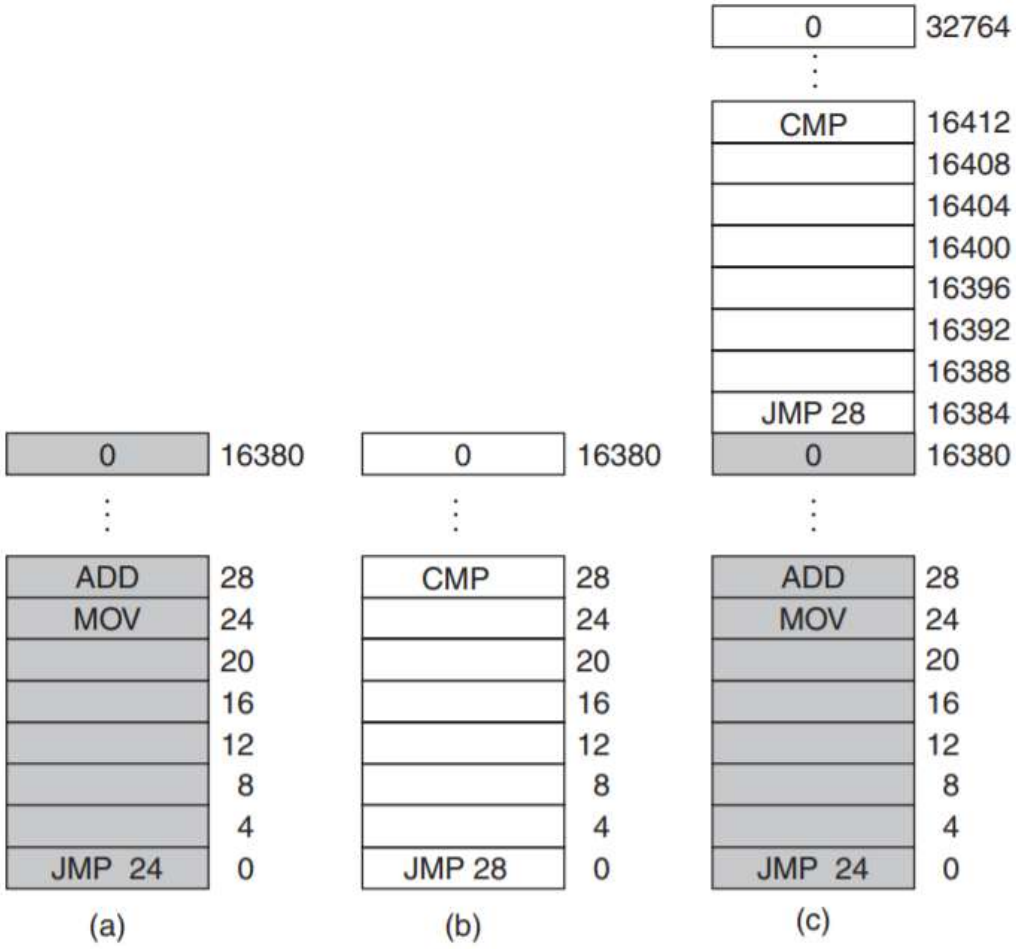
for a total of 256 bytes of key storage

(because 4 bit protection key and total registers are 512 which holds 4 bit protection key → 256 bytes of key storage (8 bits = 1 byte)).

The PSW(Program Status Word) also contained a 4-bit key.

- The 360 hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key.
    - Since only the Operating System could change the protection keys, user processes were prevented from interfering with one another and with the operating system itself.

But, still there was a major drawback:



A 16 KB program

another 16 KB program

the two programs loaded consecutively into memory

→ relocation problem

Consider two programs, each 16 KB in size as shown in Fig(a) and Fig(b).

1. the first program starts out by jumping to address 24, which contains a MOV instruction.
2. the second program starts out by jumping to address 28, which contains a CMP instruction.

When the two programs are loaded consecutively in memory starting as address 0, we have the situation as shown in fig (c).

After the programs are loaded, they can be run. Since they have different memory keys, neither one can damage the other.

But the problem:

- After the first program has run long enough, the operating system may decide to run the second program, which has been loaded above the first one, at address 16384.
    - The first instruction executed is JMP 28, which jumps to the ADD instruction in the first program, instead of CMP instruction it is supposed to jump to → program will crash.

The core problem is that the two programs are referencing absolute physical memory.

Instead each program should reference a private set of addresses local to it.