Memory Management

Main Memory is a volatile array of words or bytes, each with its own unique address. Memory is shared by the CPU and I/O devices. The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution. Program must be brought into memory and placed within a process for it to be executed. The CPU fetches instructions from memory according to the value of the Program Counter (PC). These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). You may already know how a program generates a memory address and role of Memory Address Register and Memory Buffer/Data Register. Here we will explain how symbolic memory addresses are mapped to actual physical addresses, and how and when logical addresses are translated into physical addresses. Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.

The part of Operating System which manages memory activities is Memory Management Module.

- It Keeps track of which parts of memory are currently being used and by whom.
- Decides which processes to load when memory space becomes available.
- Allocate and de-allocate memory space to processes as required.

Memory Management module is responsible for relocation, protection, sharing, logical organization, and physical organization of memory. For all these activities different approaches and algorithms are used. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system.

In a single programming system, main memory is divided into two parts: one part for the operating system and other part for the program currently being executed. Normally operating system resides at lower addresses (0 – N-1) and the program resides from N address (memory location). In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes to have a separate address space for each process. For different processes separate memory spaces, must be protected to ensure that the process can access only their legal addresses. This protection can be provided by using two registers, usually a base and a limit register.
The base register holds the smallest legal physical memory address; the limit register specifies the size/range of program. Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users’ memory results in a trap to the operating system, which treats the attempt as a fatal

error. To make sure that programs are not accessing operating system area, in most of the computer systems operating system is loaded at lower address.

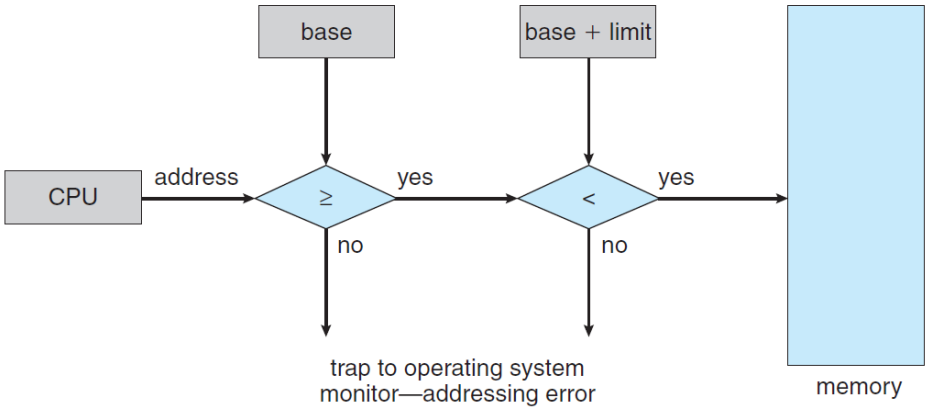

Figure-1.

Address Binding and Address Mapping/Translation

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory available at load time.

In most cases, a user program goes through several steps—some of which may be optional before being executed. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically binds these symbolic addresses to relocatable addresses (such as "16 bytes from the beginning of this module"). The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 58016). Each binding is a mapping from one address space to another. The binding of instructions and data to memory addresses can be done at any step along the way:

Compile time: If it is known at compile time where the process will reside in memory, then absolute code can be generated. For example, if it is known that a user process will reside starting at location N, then the generated compiler code will start at N and extend up from there. The MS-DOS .COM-format programs are bound at compile time. Only disadvantage is that if, at some later time, the starting location changes, then it will be necessary to recompile this code.

Load time: At compile time it is not known where the process will reside in memory, then the compiler must generate relocatable code. Final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a *logical address*, whereas an address seen by the memory unit (loaded into the Memory-Address Register of the memory) is commonly referred to as a *physical address*. The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this situation, at sometimes logical address is referred as a virtual address. The set of all logical addresses generated by a program is a logical address space. The set of all physical addresses corresponding to these logical addresses is a physical address space. The run-time mapping from virtual to physical addresses is done by a hardware device called the Memory Management Unit (MMU). There are different methods to accomplish such mapping (will discuss later) and simplest mapping is a generalization of the base and limit register scheme. The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory. The user program never sees the real physical addresses. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range B + 0 to B + max for a base value B). The user program generates only logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.

Program Size, Address Range, Address Bits

Before we discuss, memory allocation schemes, better to clarify address space range, and address bits required for a given and available size of memory. You may recall address bits being used and passed to Memory Address Register to fetch a memory location. Memory Size, Address Range, and Address Bits are related and using a simple manipulation you can know one from the other. Following table explains the relationships.

Memory Size	**2^N**	**Address bits**	**Address Range**
16B	2^4	4	0 – (2^4 - 1) 0 - 15
64B	2^6	6	0 – (2^6 - 1) 0 – 31
1KB	2^{10}	10	0 – (2^{10} - 1) 0 - 1023
64KB	?	?	0 – ($2^?$ - 1)
IMB	2^{20}	20	0 – (2^{20} - 1)
16MB	?	?	?
IGB	2^{30}	30	0 – (2^{30} - 1)
64GB	?	?	?

Table-1: Memory size, address bits and address range.

For memory of 16 bytes, which is 2^4, we require 4 bits to address 0 – 15 bytes. Similarly for memory of 1 KB, which is 2^{10}, we require 10 bits to address 0 – 1023 bytes.

Memory Addressing

Address bits being represented and manipulated internally in binary form, we can for understanding and simple manipulation can use Hexadecimal representation for addresses. We can use simple table for 0 – 15 locations represented in Decimal, Binary and Hexadecimal format.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table-2: Decimal number 0 -15 in binary and Hexadecimal

By using the Table-1 and Table-2 we can easily convert a memory address into Hexadecimal format by making groups of 4 bits as explained below.
For a memory of 1KB, which is 2^{10}, we require 10 bits to address 0 – 1023 bytes.
Maximum memory location is 1023, where all bits are '1' '1111111111'
Manually manipulating and converting this string of binary values is not easy, and there is a chance of missing bit values. We can easily convert this binary string into Hexadecimal which is more readable. We start from least significant bits and make groups of 4 bits as listed below.

'1111111111' → 11- 1111-1111

There may not be enough bits to make the last group of 4 bits, in this case we can add '0' as most significant bits to make a group of 4 bits as listed below.

'1111111111' → 0011- 1111-1111

These groups of 4 binary values can easily be converted into Hexadecimal values with the help of Table-2 (which you must have remembered by now to manually manipulate memory address). We can simply write hexadecimal values, which will be memory address in Hexadecimal. For memory location '1023' address in Hexadecimal will be '3FF'.

We will be using this address manipulation for different memory allocation schemes to compute physical address from logical address. So better to clearly understand this process and do some exercises for address computation.
Let us compute memory address for memory location '512'. In computer, memory addressing always start from '0'. Memory bits required to address '512' ?
Have a look at Table-1, for memory size of '512' bytes, we need 9 bits, and address range of that memory will be 0 – 511(means all 9 bits on →'111111111' → '1FF'). We have to compute memory address of location '512', which is one location more than location '511'. We can compute it by adding '1' to memory address ('111111111') which is '1000000000' in binary and when making groups of 4 for Hexadecimal it is '200'
Carefully look the pattern of binary string, when we add one binary value, where all least significant bits are '1', and most significant bit is '0', then most significant bit becomes '1' and least significant bits become '0'

'0011' add '1' and it becomes '0100'

Similarly for Hexadecimal where maximum value is 'F', if add '1' when least significant values are 'F', then after addition, most significant value is added by '1' and all least significant values become '0'

'1FF' add '1" and it becomes '200'

You can easily compute memory address in Binary and Hexadecimal by factorizing the value to nearest power of 2, and then by adding or subtracting small number in binary or Hexadecimal form.

Let us do a simple exercise, to compute memory address and range for 48B.
Not a power of 2. (However 48 = 32 + 16).

32 → 2^5 → 5 bits → 0 - 2^5 -1 → '0' - '11111' → '00' - '1F' (0 – 31)
16 → 2^4 → 4 bits → 0 – 2^4 -1 → '0' - '1111' → '0' - 'F' (0 – 15)

To address 64bytes, we need 6 bits (all '1" for maximum memory location).
To address 48th byte, we will need 6 bits, however not all bits will be '1'.
We can compute binary addresses by performing addition on binary or Hexadecimal values of addresses. We already computed, 31st byte address as '11111' and '1F' in binary and hexadecimal respectively. By adding '1' location we can have address of 32nd location which is '100000' and '20' in binary and hexadecimal respectively. To compute address of 48th location, we add 16 to 32 for decimal computation. Similarly, we can add binary and hexadecimal equivalent of 16 to binary and hexadecimal equivalent of 32.

'100000' + '10000' → '110000' (Binary)
'20' + '10' → '30' (hexadecimal)

Similarly, we can compute address of 40th location by adding 8 to 32 for decimal computation and by adding binary and hexadecimal equivalent of 8 to binary and hexadecimal equivalent of 32.

‘100000’ + ‘1000’ → ‘101000’ (Binary)
‘20’ + ‘08’ → ‘28’ (hexadecimal)

You can verify by factorizing and adding values to compute address for memory location 56th byte and 6KB.
6KB not a power of 2. (However 6KB = 4KB + 2KB)

4KB → 2^{12} → 12 bits → 0 - 2^{12} -1 → ‘111111111111’ → ‘FFF’
2KB → 2^{11} → 11 bits → 0 - 2^{11} -1 → ‘11111111111’ → ‘7FF’

To address 8KB, we need 13 bits (all ‘1” for maximum memory location).
To compute maximum memory location of 6KB, we can add maximum value of 2KB to maximum value of 4KB. Since memory address is starting from ‘0’ location we have to add ‘1’ while adding additional addresses to get the base address of the next location.

‘FFF’ + ‘1’ + ‘7FF’ = ‘17FF’

Compute address of 6KB in binary format

Address Translation and Memory Protection

Before we discuss memory allocation schemes, we explain memory allocation, memory protection, and logical to physical address translation, by using a simple memory size of 64KB, where lower 4KB is being used by the operating system. Memory is divided in two partitions. 4KB (0 to 4KB-1) for operating system and 60KB (4KB to 64KB-1) for user processes as shown in Figure-2.

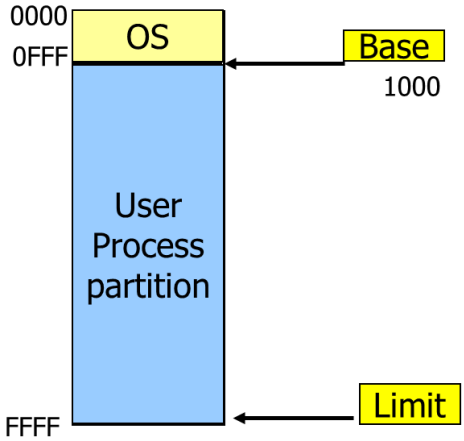

Figure-2: Physical memory addresses.

Address values are in Hexadecimal. We can use Base and Limit Registers to compute Physical address for a user process and can also see how operating system can protect itself and restricts a user process. We can use memory system in Figure-2 to place a user process of memory size 16KB. After placement of that program the memory system of Figure-2 will look like as shown in Figure-3.

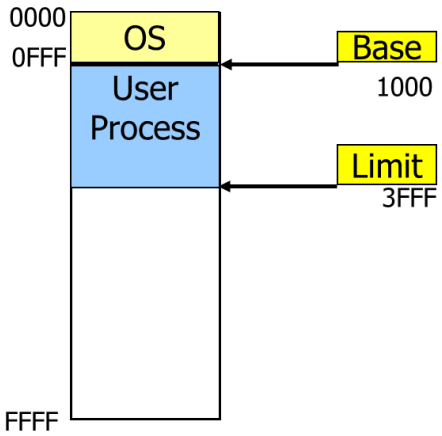

Figure-3: Physical address space of OS and user process.

From Figure-3 we can see that operating system can protect itself by comparing logical address of the process with the base register, it should be greater than or equal to base register value (in this case 1000). This basically is done at address translation time as shown in Figure-1. In the example of Figure-2, we have a process of memory size 16KB. We can easily convert a logical address less than 16KB by using base and limit register as shown in Figure-1. If we have a logical address ‘2000’ which is 8KB location, it will be converted into physical address by adding this address to Base value (‘1000’ + ‘2000’ = ‘3000’). If the process generates a logical address beyond its limit of 16KB, a trap will be generated as depicted in Figure-1.