CMP325
Operating Systems
Lecture 19, 20

Contiguous Memory Allocation

Fall 2021

Arif Butt (PUCIT)

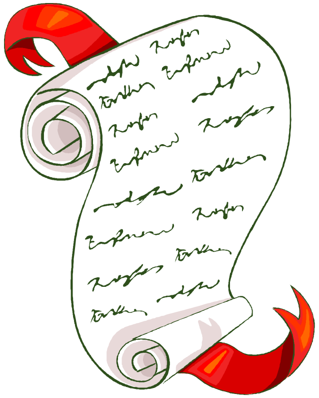
Note:

Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice video lectures on the subject of **OS with Linux** by Arif Butt available on the following link:
http://www.arifbutt.me/category/os-with-linux/

Today's Agenda

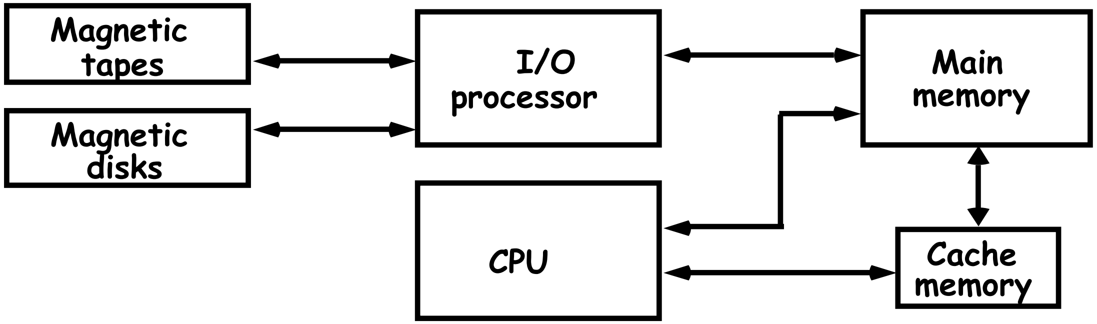


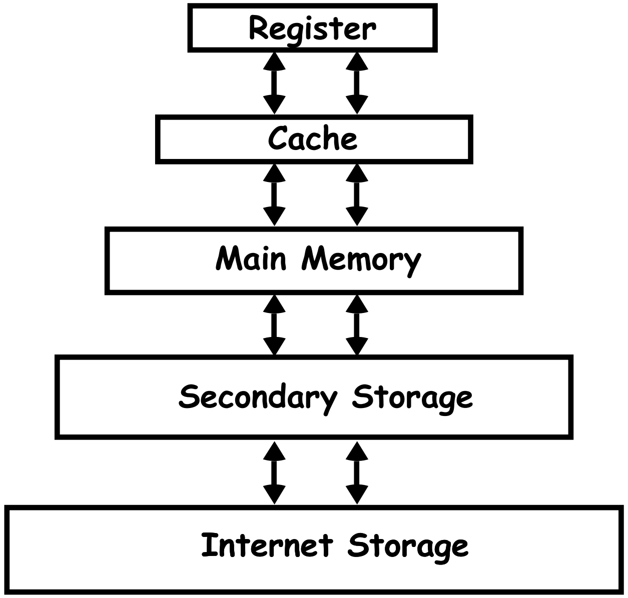
- Review of Previous Lecture
- Memory Hierarchy
- Purpose of Memory Management
- Address Translation
- Types of Address Bindings and Linking
- Overlaying
- Contiguous Memory Allocation Techniques
  - MFT
  - MVT
- Buddy Partitioning Scheme

MEMORY HIERARCHY

- Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system.

Auxiliary memory





PURPOSE OF MEMORY MANAGEMENT

To ensure fair, secure, orderly and efficient use of memory

- **Fair** means, fair distribution of memory among processes
- **Secure** means, once a process is brought in memory it should not overwrite another process and must never cross its address space
- **Orderly** means, should follow some algorithm to allocate/de allocate the memory to processes
- **Efficient** means, if a process needs 10KB, it should be given 10KB and not 100KB

- **Above tasks can be performed by doing following:**
  - Keeping track of used and free memory space
  - When, where and how much memory to allocate and de-allocate
  - Swapping processes in and out of main memory
- **Von Neumann vs. Harvard Architecture**

ADDRESS SPACE ABSTRACTION

- **Address Space**
  - All memory data
  - i.e. program code, data, stack, heap,...
- **Hardware Interface (Physical Reality)**
  - Computer has one **small**, **shared** memory
- **Application Interface (Illusion)**
  - Each process wants **large**, **private** memory

How can we close this gap?

ADDRESS SPACE Illusions

- **Address Independence**
  - Same address space can be used in different address spaces, yet remain logically distinct.
- **Protection**
  - One address space cannot access data in another address space
- **Virtual Memory**
  - Address space can be larger than the amount of physical memory on the machine

ADDRESS SPACE Illusions

ILLUSION	REALITY
• Giant address space protected from others (unless you want to share)	• Many processes sharing one address space
• More whenever you want it	• Limited memory

LOGICAL & PHYSICAL ADDRESSES

- **Logical Addresses/Virtual Addresses.** An address generated by the process / CPU; refers to an instruction or data within the address space of the process. Process generate address within its own address space. Set of all logical addresses generated by a process comprises its logical address space.

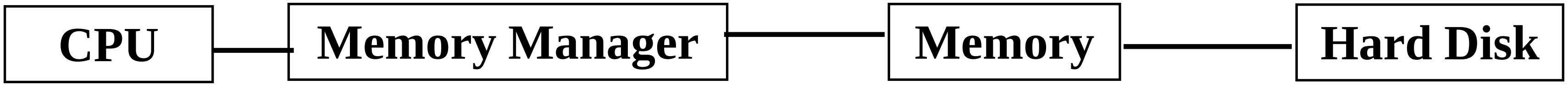
- **Physical Addresses.** An address for a main memory location where instruction or data resides is called the physical address. Set of all physical addresses corresponding to the logical addresses comprises the physical address space of that process.

- The run time mapping of logical to physical address is done by a piece of the CPU h/w called the memory management unit (MMU).

- Note: **Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme**

UNI & MULTI PROGRAMMING SYSTEMS

- Main memory is used to store data & programs for execution. Main memory is considered as an array of bytes, every byte having a unique address.
- **Uni-programming System.**
  - Main memory is divided into two parts. One for the OS and other for the program currently being executed. Only one program can execute at a time. Once it is over the next program is loaded. If a program is larger than the user program area, the OS loads & executes it portion by portion(overlaying).
- **Multiprogramming System.**
  - User program area is further sub divided to accommodate multiple processes / programs. This task of sub division is carried out dynamically by the OS and is known as Memory Management. Memory Manager is responsible to keep track of which parts of memory are in use and which parts are not in use to bring programs /parts of programs form auxiliary memory to main memory & vice versa.



CPU

Memory Manager

Memory

Hard Disk

Addr generated by CPU is called Logical Addr

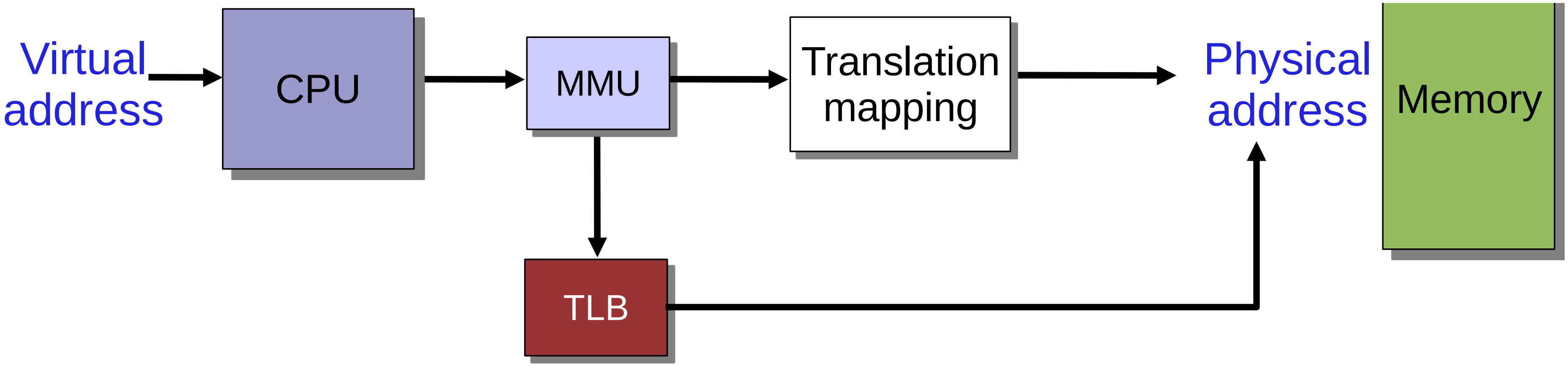
Address seen by memory unit is called Physical / Absolute addr

Hard Disk Address is called Virtual Address

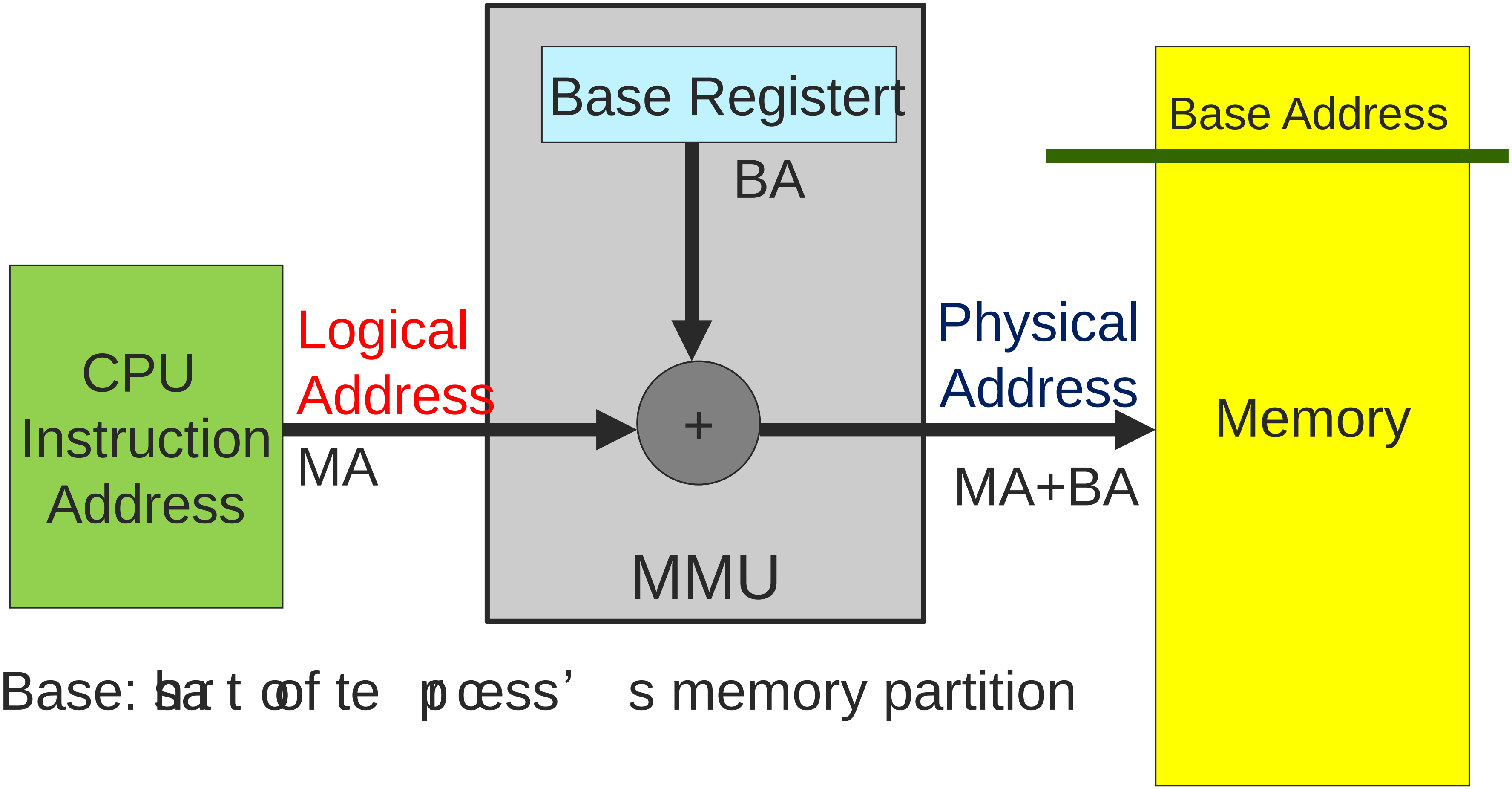
ADDRESS TRANSLATION using Base and Limit Registers

MEMORY MANAGEMENT UNIT

- Hardware that translates a logical address to a physical address
- Each memory reference is passed through the MMU
- Translate a logical address to a physical address
- There are lots of ways of doing it.
  - Base and Limit Registers
  - Segmentation
  - Paging
  - Paged Segmentation

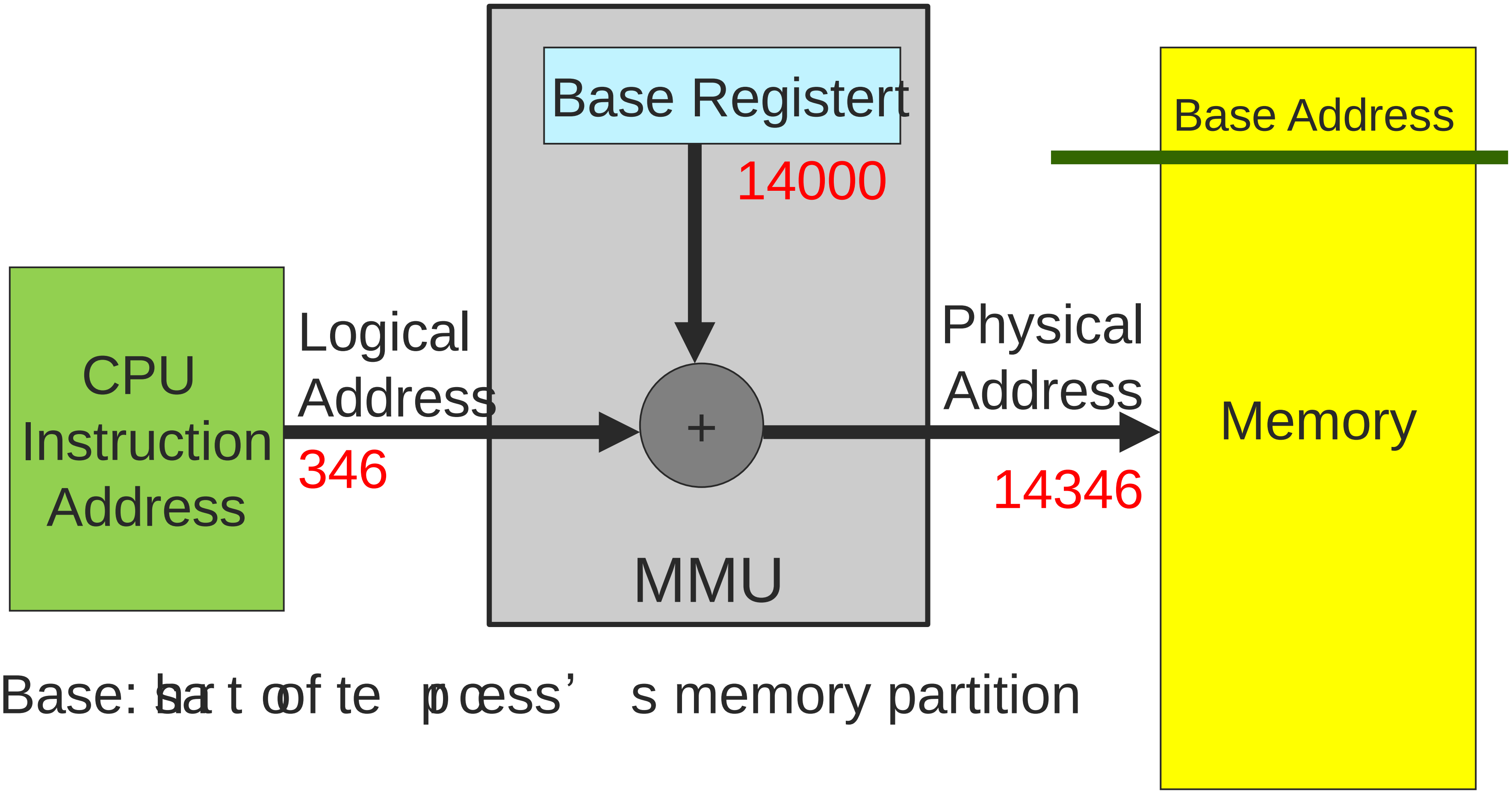


BASE REGISTER



Base: start of the process’ s memory partition

BASE REGISTER

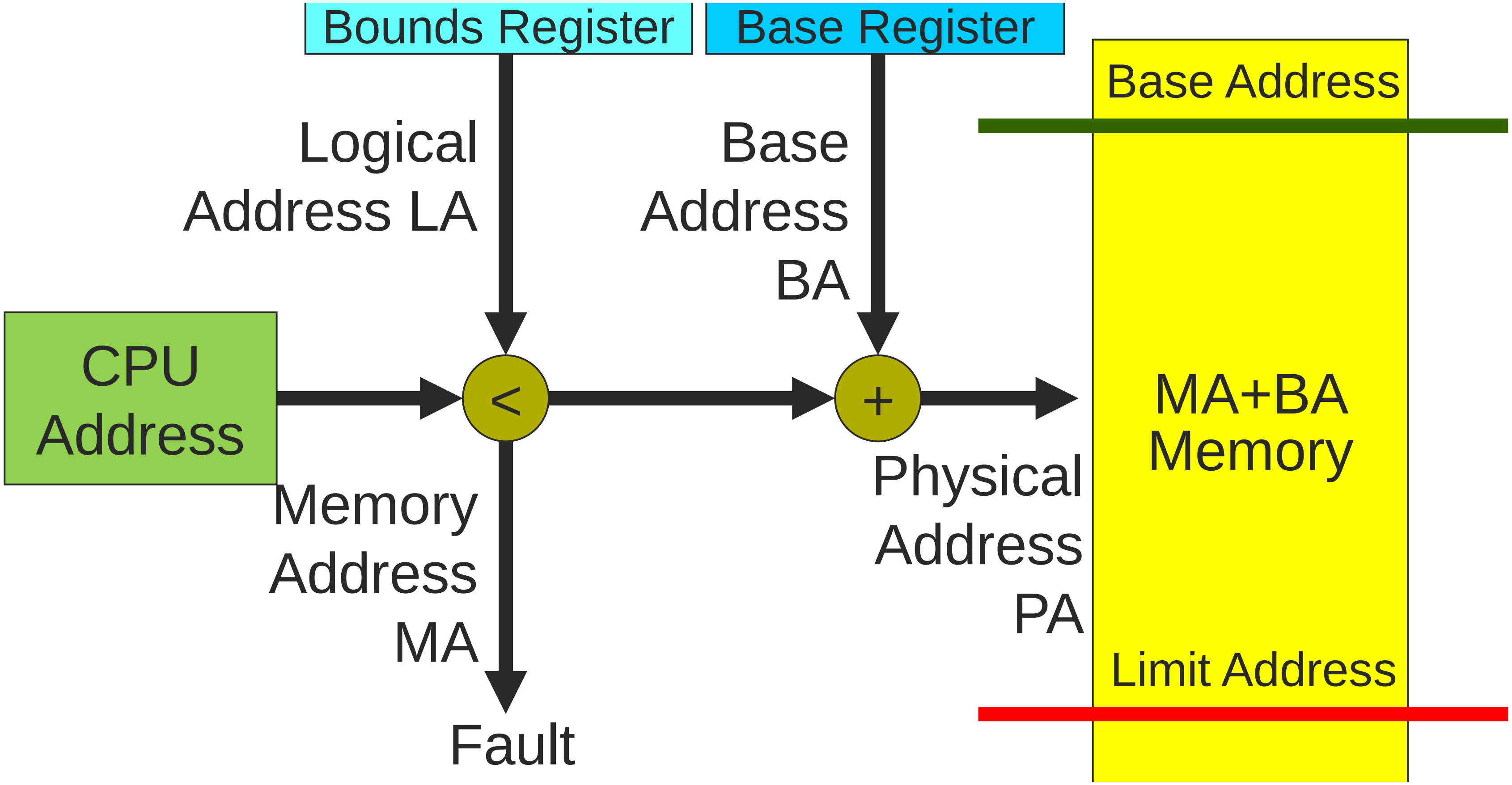


Base: start of the process' s memory partition

PROTECTION

- **Problem**
  - How to prevent a malicious process from writing or jumping into another user's or OS partition.
- **Solution**
  - Bound/Limit Register

PROTECTION



- **Base:** Start of the process's memory partition
- **Limit.** Max address in the process's memory partition

BASE AND LIMIT REGISTERS

- **What must change during context switch:**
  - The base and bound registers
- **Can a process change its own base and bound:**
  - No, only OS can change these registers
  - The program can only do it indirectly (e.g., ask for more memory in stack)
- **Process may need more memory with time, how does the kernel handle address space growing?**
  - You are the OS designer, design algorithm for allowing processes to grow

LOGICAL & PHYSICAL ADDRESSES (cont…)

BASE OFFSET ADDRESSING

- **Problem1**

  There is a Base / Relocation register, whose contents are added to every address generated by a user process at the time it is sent to memory. Draw its pictorial representation.

- **Problem 2**

  In i8086, the logical address (16 bits) of the next instruction is specified by the value of IP register. The physical address of the instruction is computed by shifting the CS register (16 bits) left by four bits and adding contents of IP to it, thus generating a physical address of 20 bits. Draw its pictorial representation.

- **Problem 3**

  In i8086, the contents of IP register is ox0B10 and contents of Code Segment contains oxD000. Compute the Physical address and also give the size of Logical and Physical address space.

ADDRESS BINDING

.

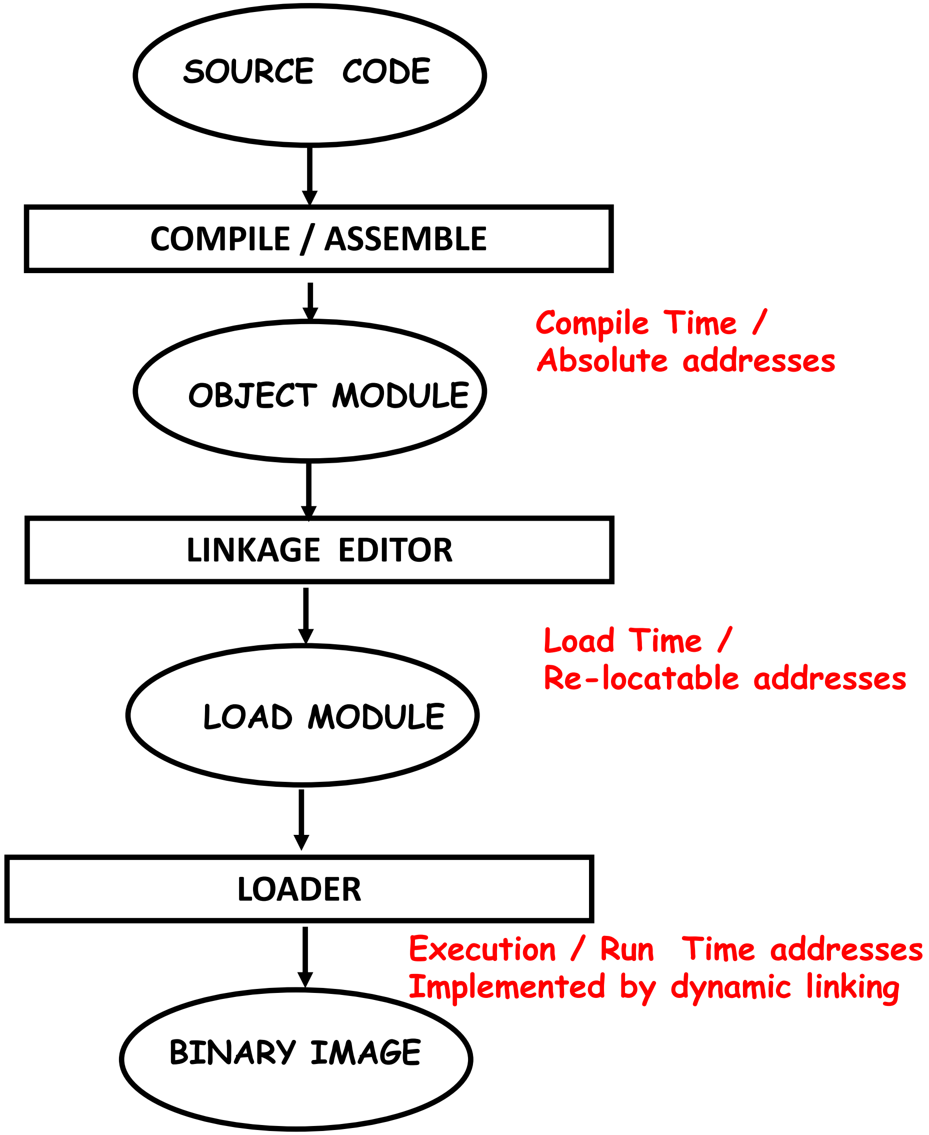
By Loader

By Linker

SOURCE CODE TO EXECUTION

Object code is in the form of Machine instruction that the underlying CPU can understand, but it is not executable because of certain references (e.g. reference to some library calls) which are not part of it.

Linkage Editor links the function calls that are not part of the object code i.e. static linking.

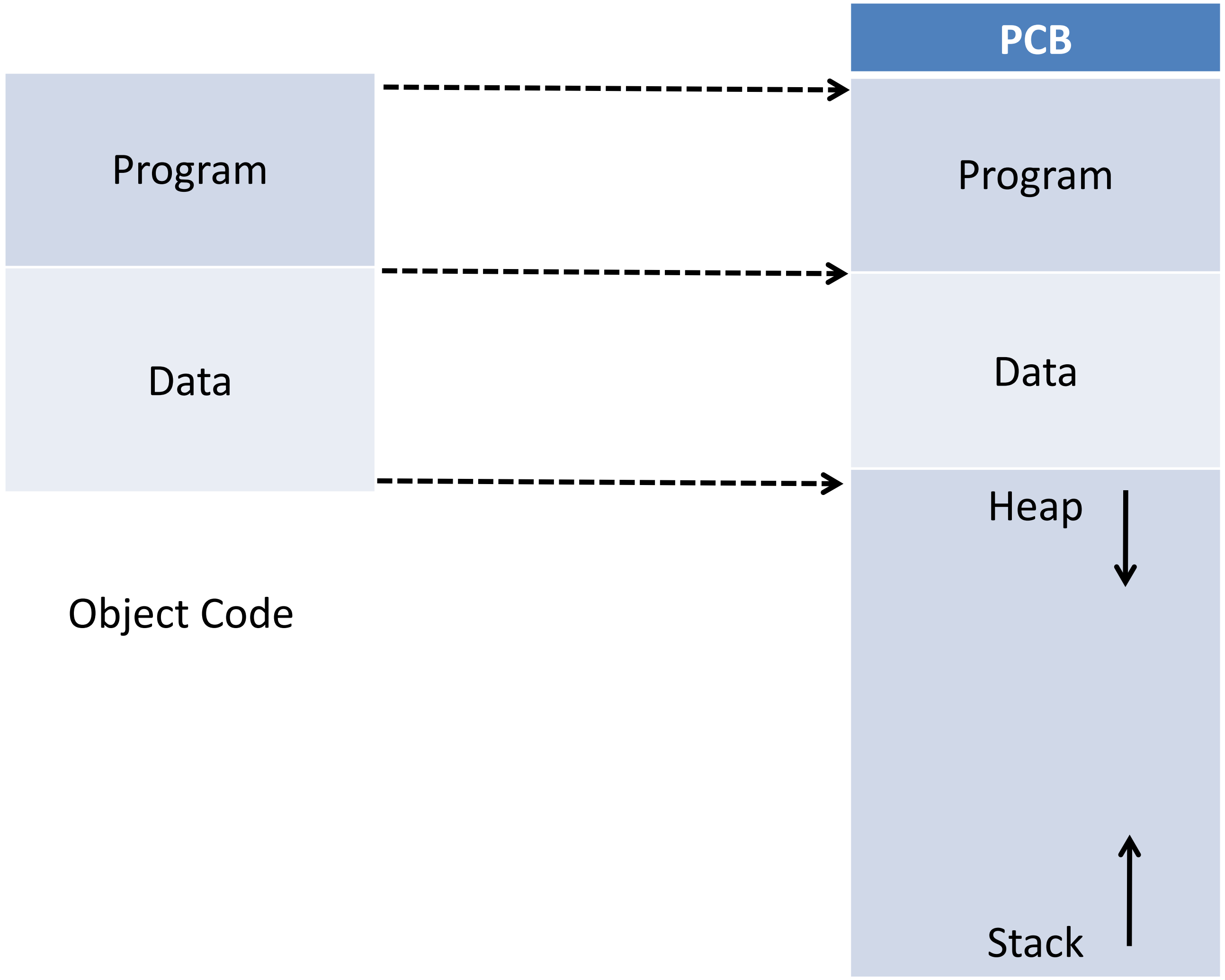


Compile Time / Absolute addresses

Load Time / Re-locatable addresses

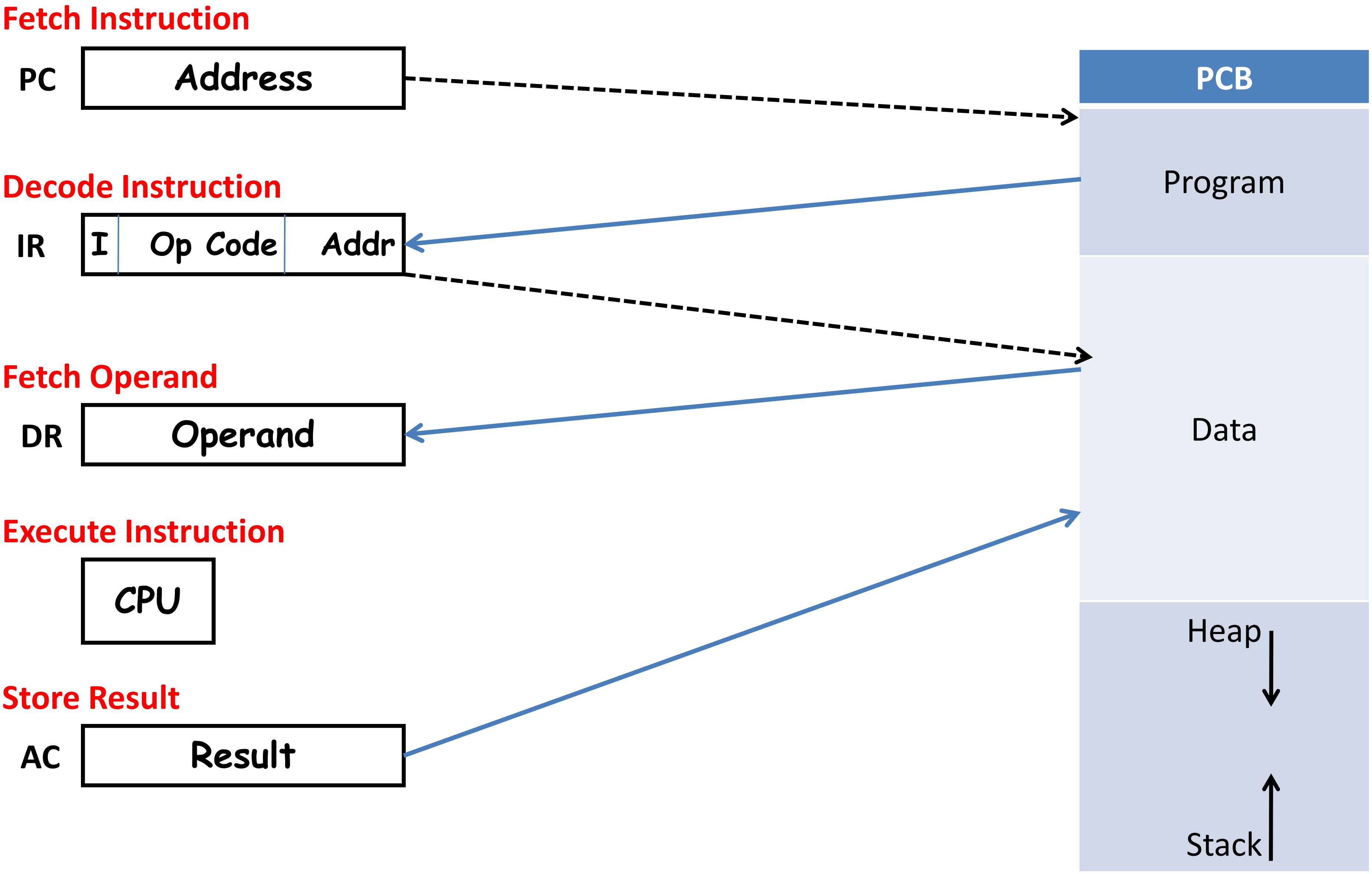
Execution / Run Time addresses
Implemented by dynamic linking

LOADING FUNCTION



Process image in main memory (Stored Program Concept)

Instruction Execution Cycle



Fetch Instruction

PC Address

Decode Instruction

IR I Op Code Addr

Fetch Operand

DR Operand

Execute Instruction

CPU

Store Result

AC Result

PCB

Program

Data

Heap

Stack

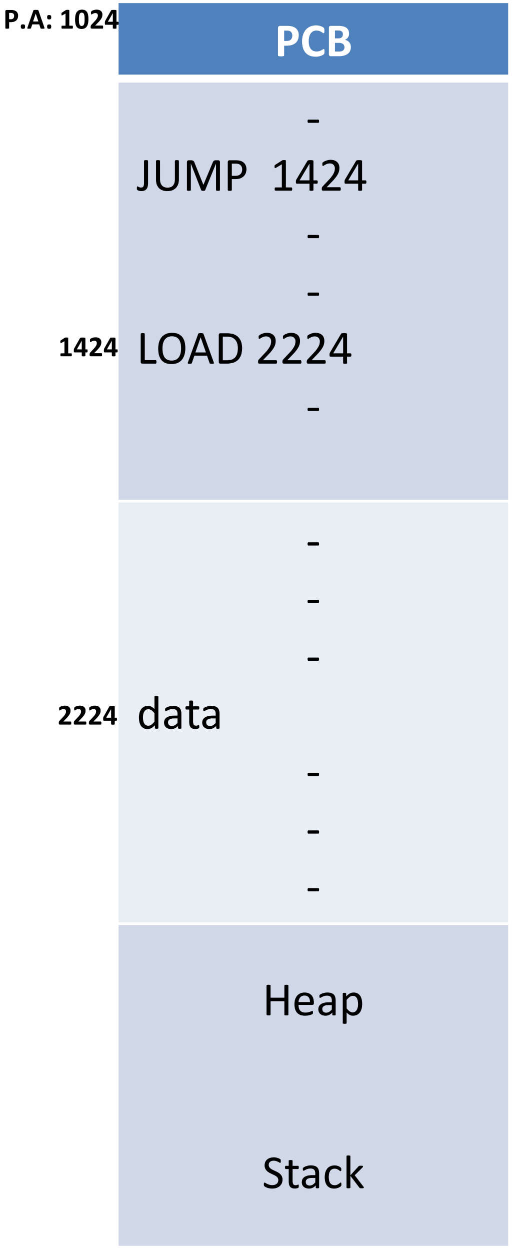
ADDRESS BINDING - LOADER

Binding Time	Function
Programming time	All actual PA are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler
Load time (Re-locatable Loading)	The compiler or assembler produces relative addressees, the loader translates these to absolute addresses at the time of program loading.
Run / Execution time (Dynamic Run Time Loading)	The loaded program retains relative addresses. The process can move from one memory region to another during execution. These are converted dynamically to absolute addresses by processor hardware.

BINDING OF INSTRUCTIONS AND DATA TO ADDRESSES

Programming Time

- All actual physical addresses are directly specified by the programmer in the program itself
- **Limitation:**
  - Very difficult for programmers to specify address
  - Used normally in uni-process programming environment



P.A: 1024

PCB

-

JUMP 1424

-

-

1424 LOAD 2224

-

-

-

-

2224 data

-

-

-

Heap

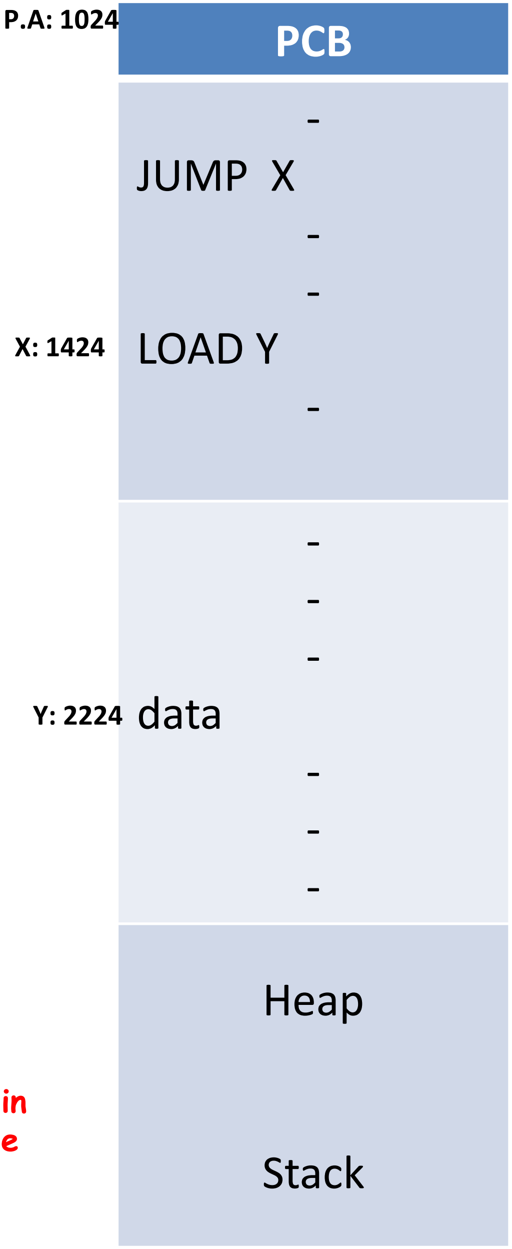
Stack

BINDING OF INSTRUCTIONS AND DATA TO ADDRESSES

Compile Time

- The program contains symbolic address references. These are converted to Physical Addresses by the compiler/assembler
- If you know at compile time where the process will reside in memory, the absolute code can be generated by the compiler.
- **Limitation:**
    - Process must reside in the same memory region for it to execute correctly. If the addresses are not free, we cannot load the program for the execution, even if lot of memory space is available

So in program time as well as in compile time address binding, we can load a process in memory if and only if the absolute addresses for instructions and data are free inside the memory



P.A: 1024

PCB

-

JUMP X

-

-

X: 1424 LOAD Y

-

-

-

-

Y: 2224 data

-

-

-

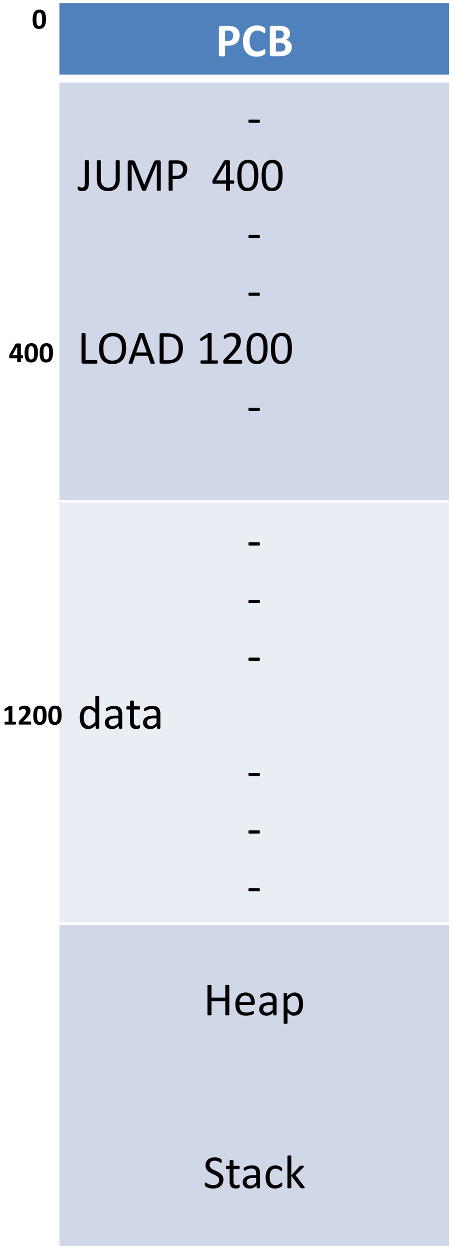
Heap

Stack

BINDING OF INSTRUCTIONS AND DATA TO ADDRESSES

Load Time (Re-locatable)

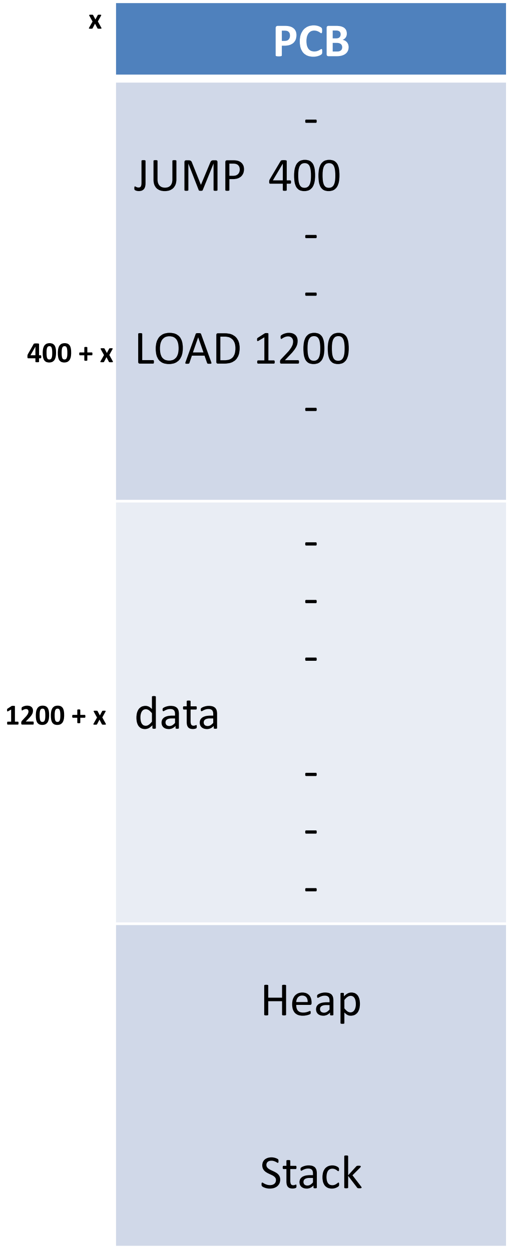
- Initially addresses within the process are relative to start address. Final binding is delayed until load time. Process can be loaded at any free slot in memory. Data and instructions are assigned addresses by loader at load time.



BINDING OF INSTRUCTIONS AND DATA TO ADDRESSES

Load Time (Re-locatable)

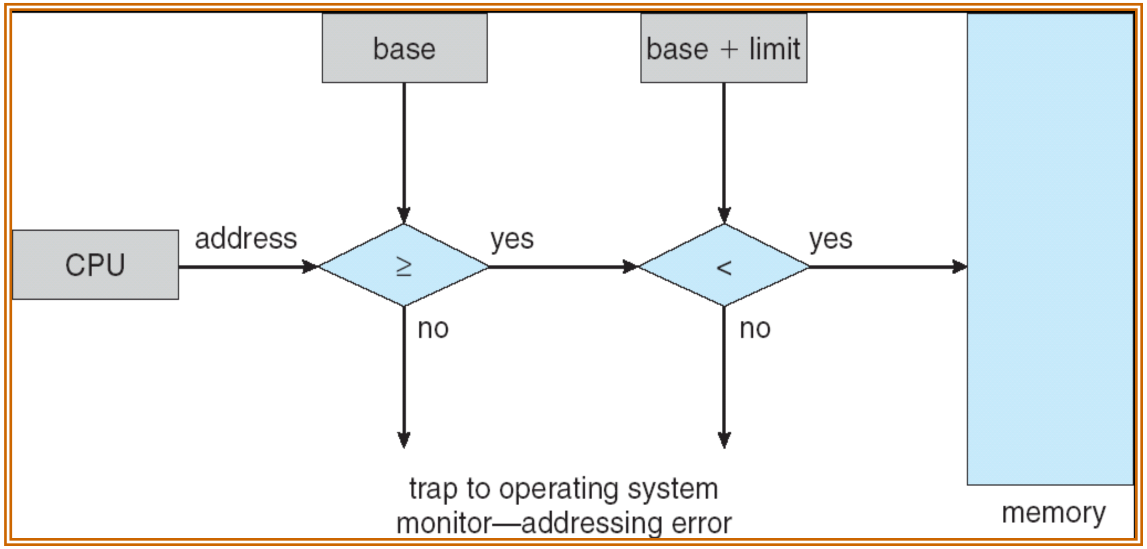
- Initially addresses within the process are relative to start address. Final binding is delayed until load time. Process can be loaded at any free slot in memory. Data and instructions are assigned addresses by loader at load time
- **When this program is loaded into memory at address x then:**

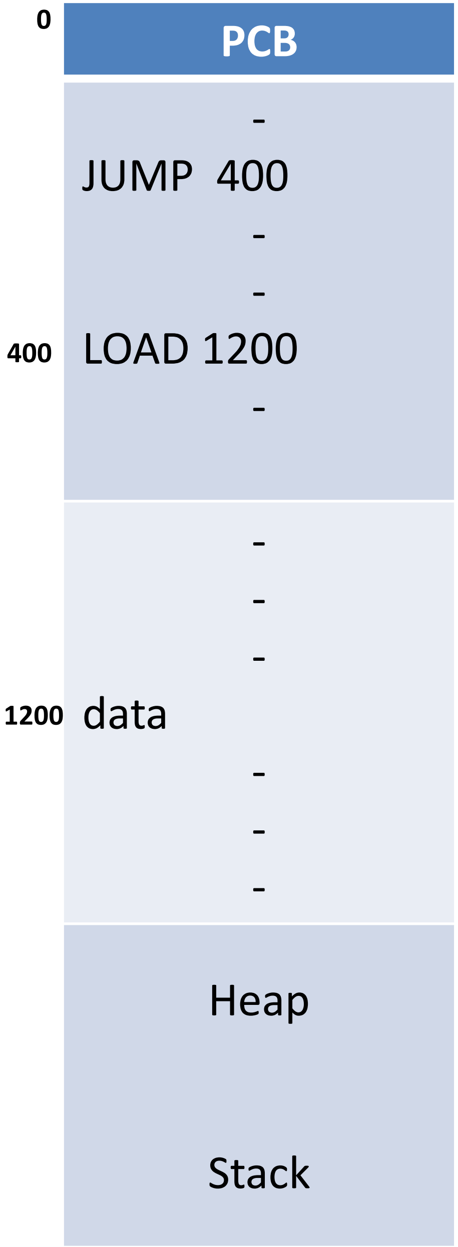


BINDING OF INSTRUCTIONS AND DATA TO ADDRESSES

Dynamic/Run Time

- The loaded program retains relative addresses
- The relative addresses are converted dynamically to absolute addresses by CPU hardware
- The process can move from one memory region to another during execution
- Need hardware support for address maps (e.g., *base* and *limit registers*)





0 PCB

-
JUMP 400
-
-
400 LOAD 1200
-

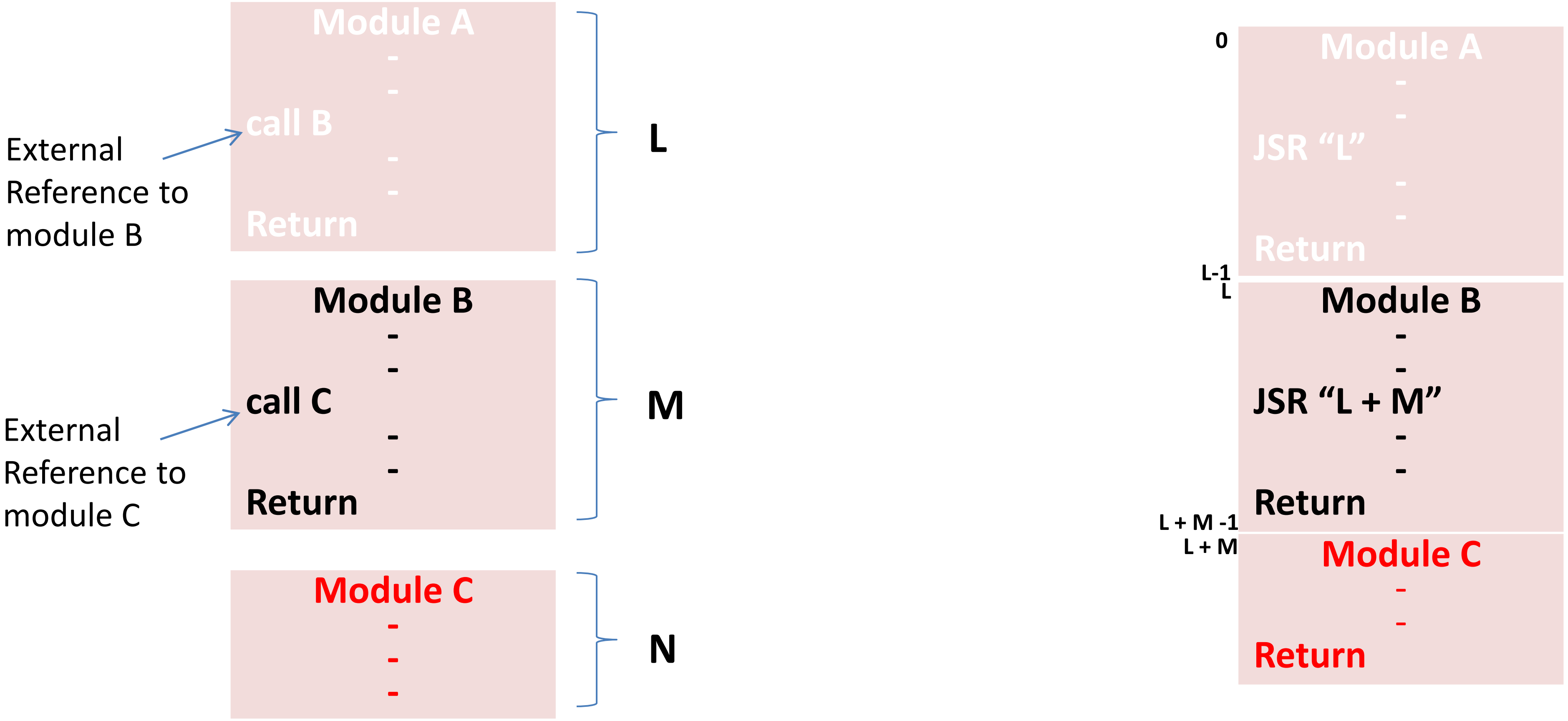
-
-
-
1200 data
-
-
-

Heap

Stack

LINKING AND LOADING

- The function of a linker is to take as input a collection of object modules and produce a load module, consisting of an integrated set of program and data modules to be passed to the loader
- Each intra module reference must be changed from a symbolic address to a reference to a location within the over all load module



External Reference to module B

Module A
-
-
call B
-
-
Return

L

External Reference to module C

Module B
-
-
call C
-
-
Return

M

Module C
-
-
-

N

0

Module A
-
-
JSR "L"
-
-
Return

L-1
L

Module B
-
-
JSR "L + M"
-
-
Return

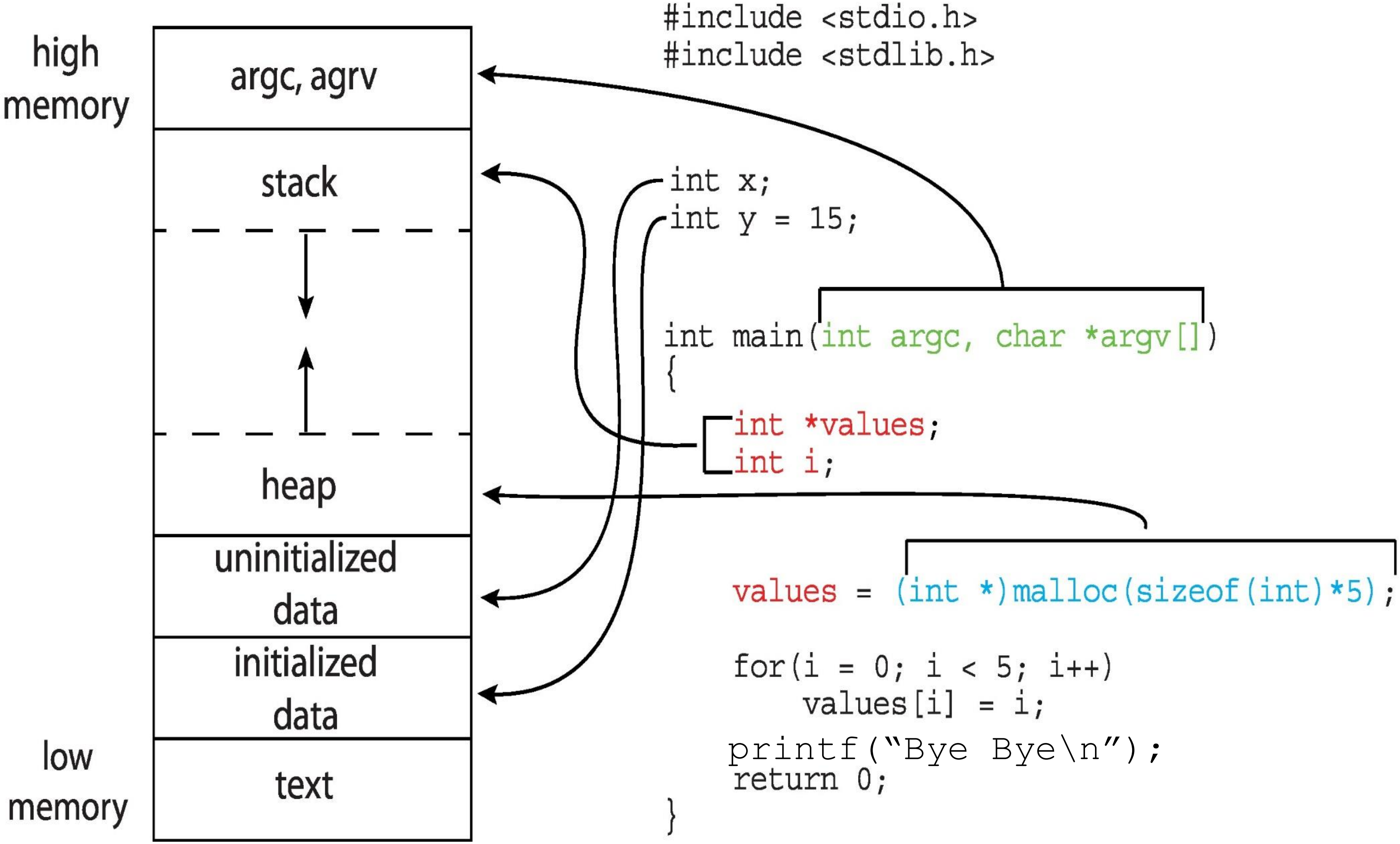
L + M -1
L + M

Module C
-
-
Return

ADDRESS BINDING - LINKER

Binding Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module Creation (Static Linking)	All object modules have been assembled using relative addresses. These modules are linked together and all references restarted relative to the origin of the final load module.
Dynamic Linker defer the linkage of some external modules, i.e., the load module contains unresolved references to other programs. These references can be resolved either at load time or at run time	
Load time dynamic linking	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main memory.
Run time dynamic linking	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted, OS locates the module, loads it, and links it to the calling module.

Mapping of a C Program into Process Address Space



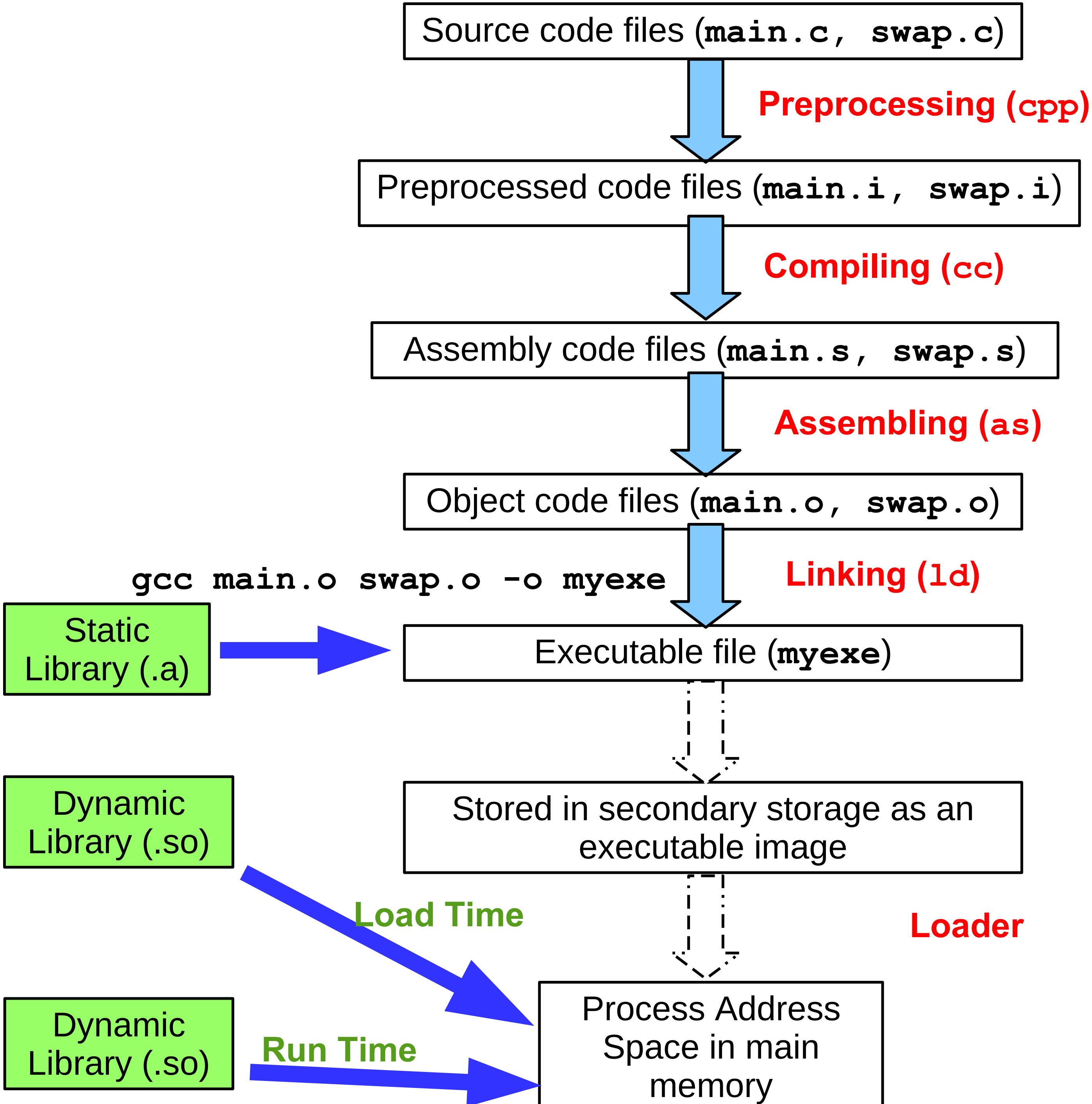
```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;
    printf("Bye Bye\n");
    return 0;
}
```



Source code files (**main.c, swap.c**)

Preprocessing (cpp)

Preprocessed code files (**main.i, swap.i**)

Compiling (cc)

Assembly code files (**main.s, swap.s**)

Assembling (as)

Object code files (**main.o, swap.o**)

```
gcc main.o swap.o -o myexe
```

Linking (ld)

Static Library (.a)

Executable file (**myexe**)

Stored in secondary storage as an executable image

Dynamic Library (.so)

Load Time

Loader

Dynamic Library (.so)

Run Time

Process Address Space in main memory

STATIC & DYNAMIC LINKING

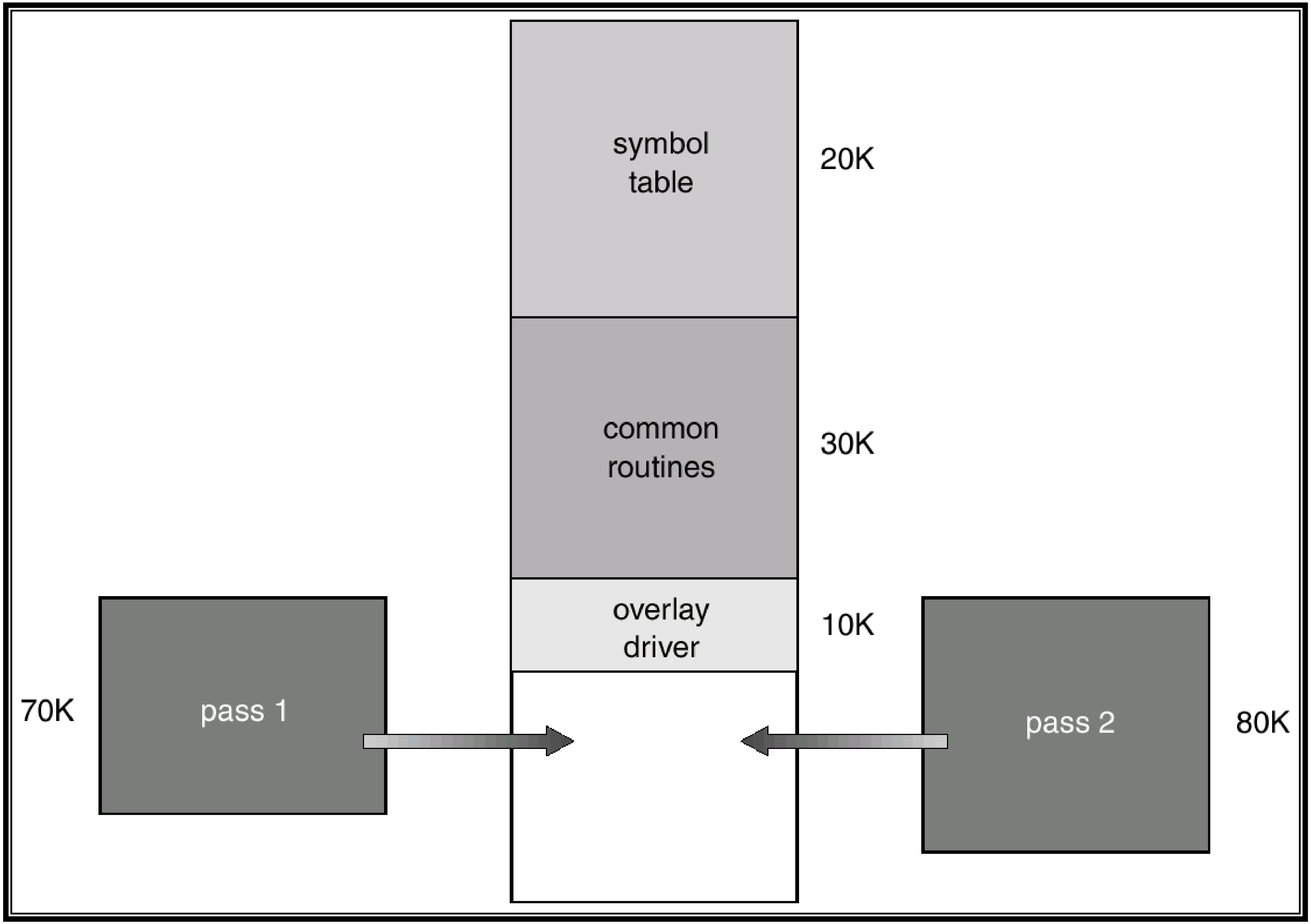
- In static linking, system language libraries are linked at compile time.
- In dynamic linking the linking decision is postponed until run time
- A library call is replaced by a piece of code called stub, which is used to locate memory resident library routine
- During execution of a process stub is replaced by the address of the relevant library code and the code is executed
- If library code is not in memory it is loaded at this time of requirement
- **Advantages of Dynamic Linking**
  - Less time needed to load a program
  - Less memory space needed
  - Less disk space needed to store binaries
- **Disadvantages of Dynamic Linking**
  - Time consuming Run time activity resulting in slower program execution. (i.e. getting some piece of code from disk and loading it in memory)
  - gcc compiler use dynamic linking by default, use –static option to force static linking

OVERLAYING

OVERLAYING

- Overlays allow a process to be larger than the amount of memory allocated to it
- Keep in memory only those instructions and data that are needed at any given time
- When other instruction and data are needed they are loaded into the space occupied by previous instruction and data that are no longer needed
- Implemented by programmer
- **Example**. Lets take the example of a 2 pass assembler/compiler
  - Pass 1 – 70 K (Parsing and syntax analysis)
  - Pass 2 – 80 K (Generate object code)
  - Symbol table –20 K (Contains language grammar & is constructed in Pass 1)
  - Common Routines – 30K
- Over lay driver is a user defined program (nothing to do with OS) that decides what to load first, when to unload and what to load next
- The pictorial diagram showing overlaying technique of above example is shown on next slide
- Process size is 200 K, while memory allocated for the process is 150K. So we need to do overlaying

OVERLAYING

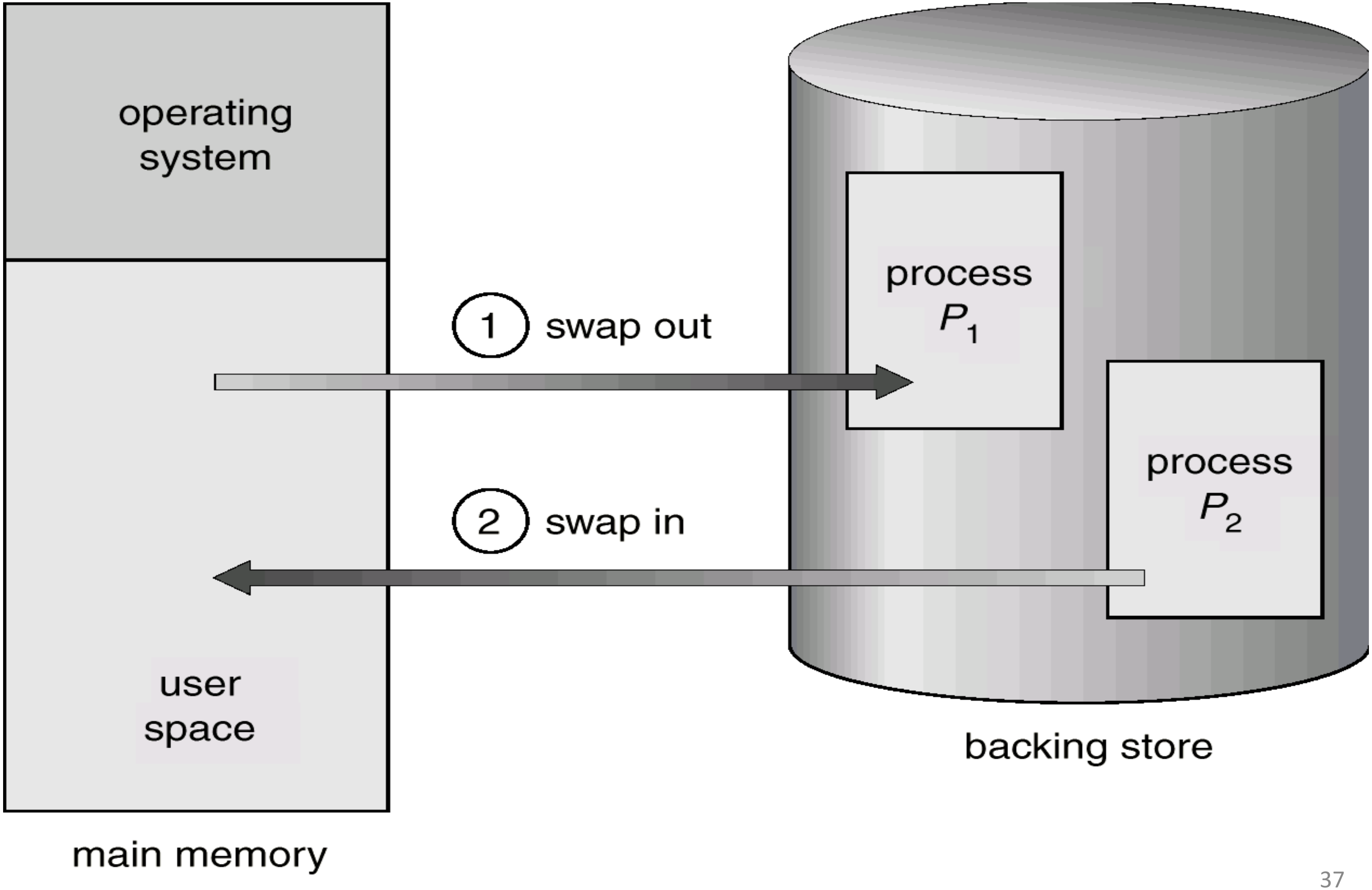

symbol table
20K
common routines
30K
overlay driver
10K
70K
pass 1
pass 2
80K

SWAPPING

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- ***Roll out, roll in*** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

- **Problem 4.** Consider a process of size 1 MB. Disk to memory transfer rate is 5 MB per second. Consider the average disk latency to be 8 msec (assume no seek time). Calculate the total cost of swapping

SCHEMATIC VIEW OF SWAPPING



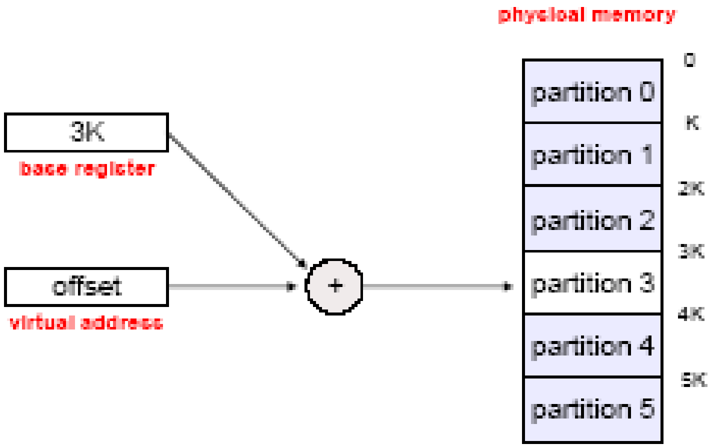
CONTIGUOUS SPACE ALLOCATION

CONTIGUOUS ALLOCATION

- Entire process will come in memory at a time and it will be loaded at one place i.e. contiguous
- The memory is divided into two main parts:
  - **User Space**; contain the user programs
  - **Kernel Space**; contains the OS kernel
- A process is placed in a single contiguous area in memory
- **Base** (re-location) and **limit** registers are used to point to the smallest memory address of a process and its size respectively
- Two techniques used for contiguous allocation are:
  - Multiprogramming with Fixed Tasks **(MFT)**. (Multiple fixed size partitions)
  - Multiprogramming with Variable Tasks **(MVT)**. (Multiple variable size partitions)

M F T

- Multi programming with Fixed Tasks; Fixed Partitioning
- Memory is divided into several fixed size partitions. Each partition may be of same or different size. Each partition contain exactly one process / task
- For now, think of a program as having a contiguous logical address space that starts at 0, and a contiguous physical address space that starts somewhere else
- When a partition is free, a process is selected from the input queue and is loaded in the free partition. When the process terminates, the partition becomes available for another process
- Hard ware requirements are a Base Register, which is loaded by the OS when it switches to a process
- P.Address = L. Address + Base Register
- Used by IBM for system 360 OS/MFT

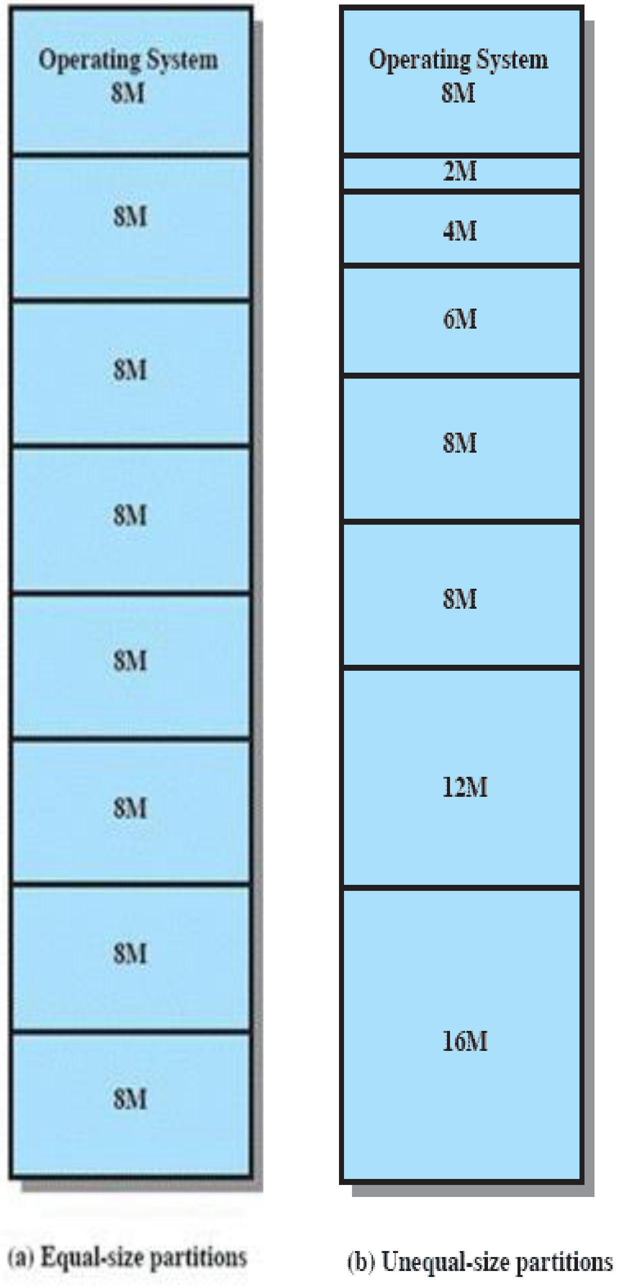


M F T (cont…)

- **Equal size partitions (Fig (a))**
  - Any process whose size is less than or equal to that partition size can be loaded into an available partition
  - The OS can swap a process out of a partition, if none are in a ready or running state
  - A program may not fit in a partition, in which case the programmer must design the program with overlays
  - Main memory use is inefficient, because any program, no matter how small, occupies an entire partition; which results in internal fragmentation

Un-Equal size partitions (Fig (b))

- Lessens both above problems, however, doesn't solve completely
- Programs up to 16 M can be accommodated without overlay
- Smaller programs can be placed in smaller partitions, reducing internal fragmentation



(a) Equal-size partitions

(b) Unequal-size partitions

M F T (cont…)

M F T with Multiple Input Queues

- Every partition will have a separate queue of processes
- One input Queue per partition
- Every process coming to the memory will be moved to a separate queue depending on its size
- Pictorial diagram representing MFT with multiple i/p Queues is shown on next slide

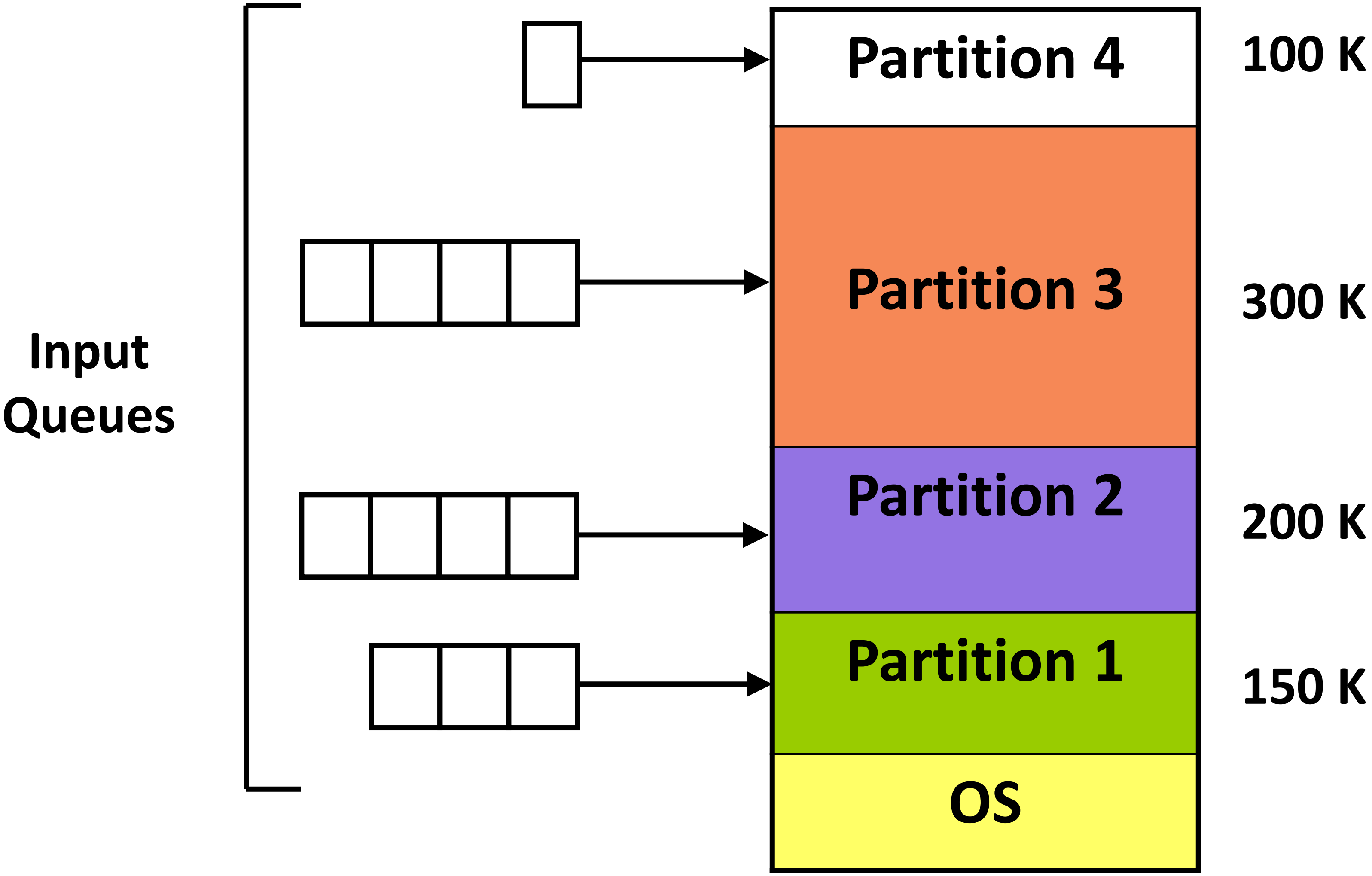
M F T with Single Input Queue

- Single input Queue for all the partitions
- Search the Queue for a process when a partition becomes empty
- Which process to be brought in depends on the algo applied
- Can be Best fit, First fit, Next fit or Worst fit
- Pictorial diagram representing MFT with single i/p Queue is also shown on next slide

Limitations

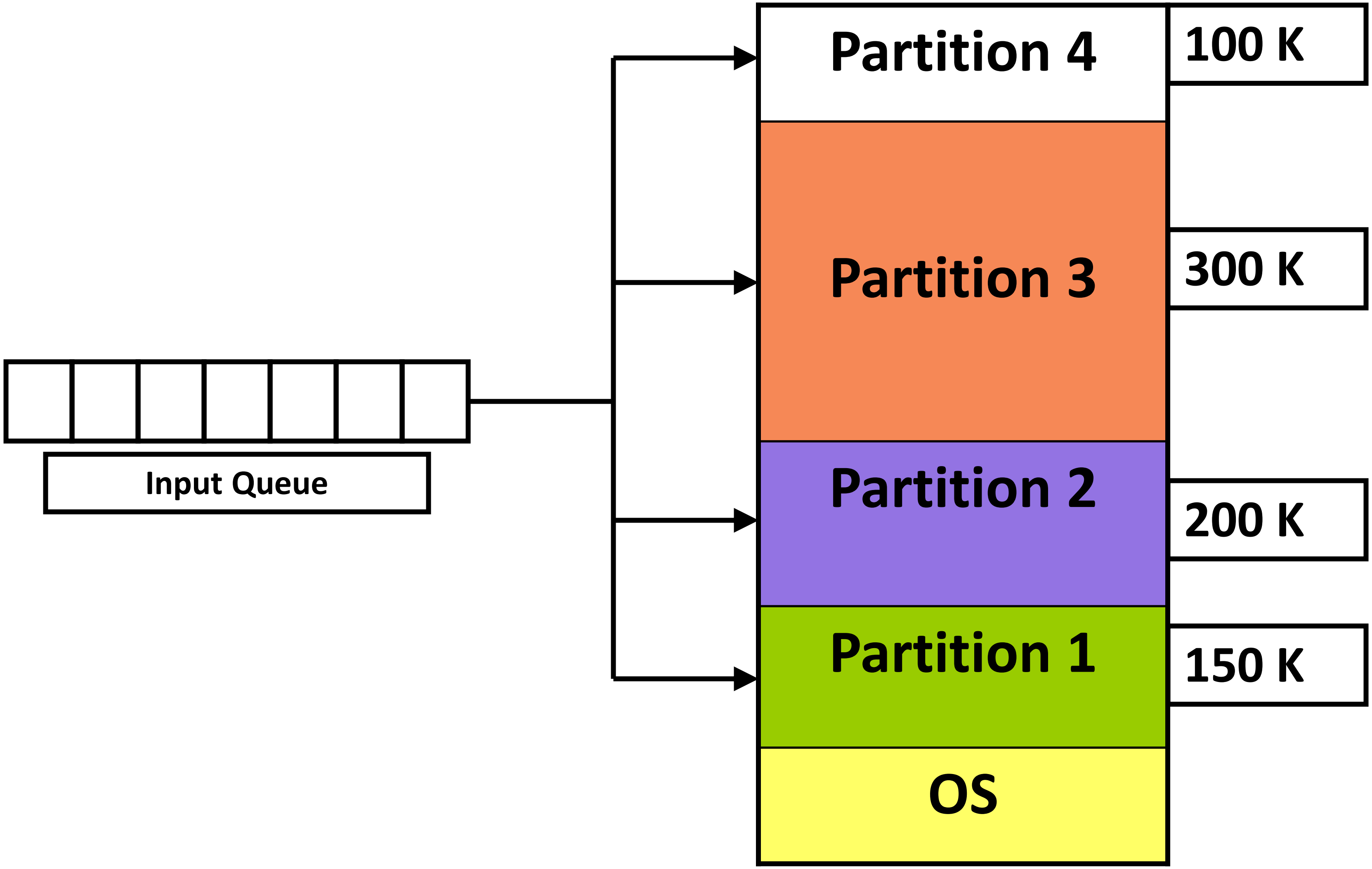
- – A process may be too big to fit into a partition, the programmer must design the program with the use of over lays
- – Load time address binding is necessary, even if the process is swapped out it will be swapped in to the same partition. (limitation of multiple input queues only)
- – Internal Fragmentation. Memory space is wasted, if a process is smaller in size than the partition, space not used by the process is lost. (this limitation is lessened in single input queue)
- – No sharing between processes

M F T (cont…)



Multiprogramming with Fixed Tasks with a queue per partition

M F T (cont…)



Multiprogramming with Fixed Tasks with one input queue

M V T / Dynamic Partitioning

- Multi programming with Variable Tasks; or Dynamic Partitioning
- Partitions are created dynamically (run time), so that each process is loaded into a partition of exactly the same size as that of the process.
- Both the number and size of partitions change with time
- Jobs can move form one partition to another. After swapping out the process can be swapped in to some other partition
- No internal fragmentation (not exactly)

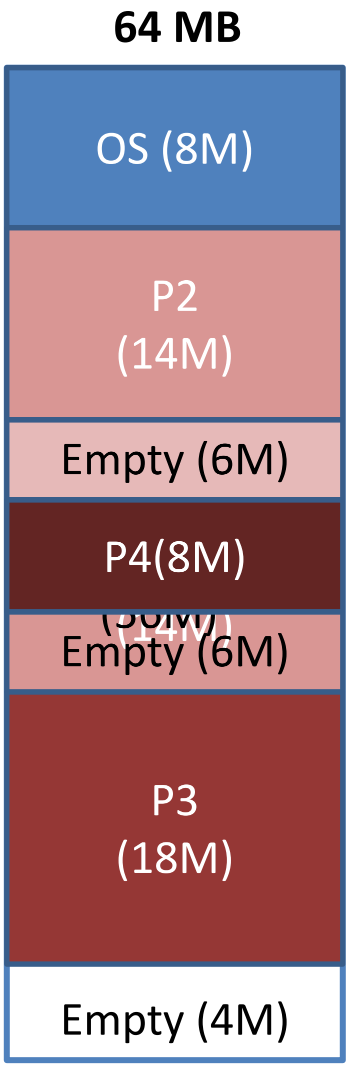
Limitation

- This technique starts well but eventually leads to a situation where a lot of small holes are available in memory. As time goes on the memory become more and more fragmented. All these holes make up enough memory to bring in a process but these holes are not contiguous

- **External fragmentation** refers to the state of memory space when the total amount of unused space exists to satisfy a request but this memory space is not contiguous

- Solution is **Compaction.** From time to time OS shifts all the processes downwards, so that all the holes are converted into one big hole. Time consuming and possible only if address binding is done at run time. OR swap out one process to bring in another

M V T / Dynamic Partitioning



- ***External Fragmentation***
- Memory external to all processes is fragmented
- Suppose now a process of size 10 MB comes, it cannot be accommodated, we do have 16 MB free memory, but that is not contiguous
- Can resolve using ***compaction***
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

M V T (cont…)

Placement Algorithms. When it is time to load / swap a process into main memory and if there is more than one free block of memory of sufficient size, then the OS must decide which free block to allocate. It can use any of the following algorithms:

- **Best Fit**
  - Scan all holes and chooses the one with a size closest to the requirement
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often
- **First Fit**
  - Scan from beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block
- **Next Fit**
  - Scan memory from the location of last placement and chooses next available block that is large enough
  - More often allocated a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

SAMPLE PROBLEMS

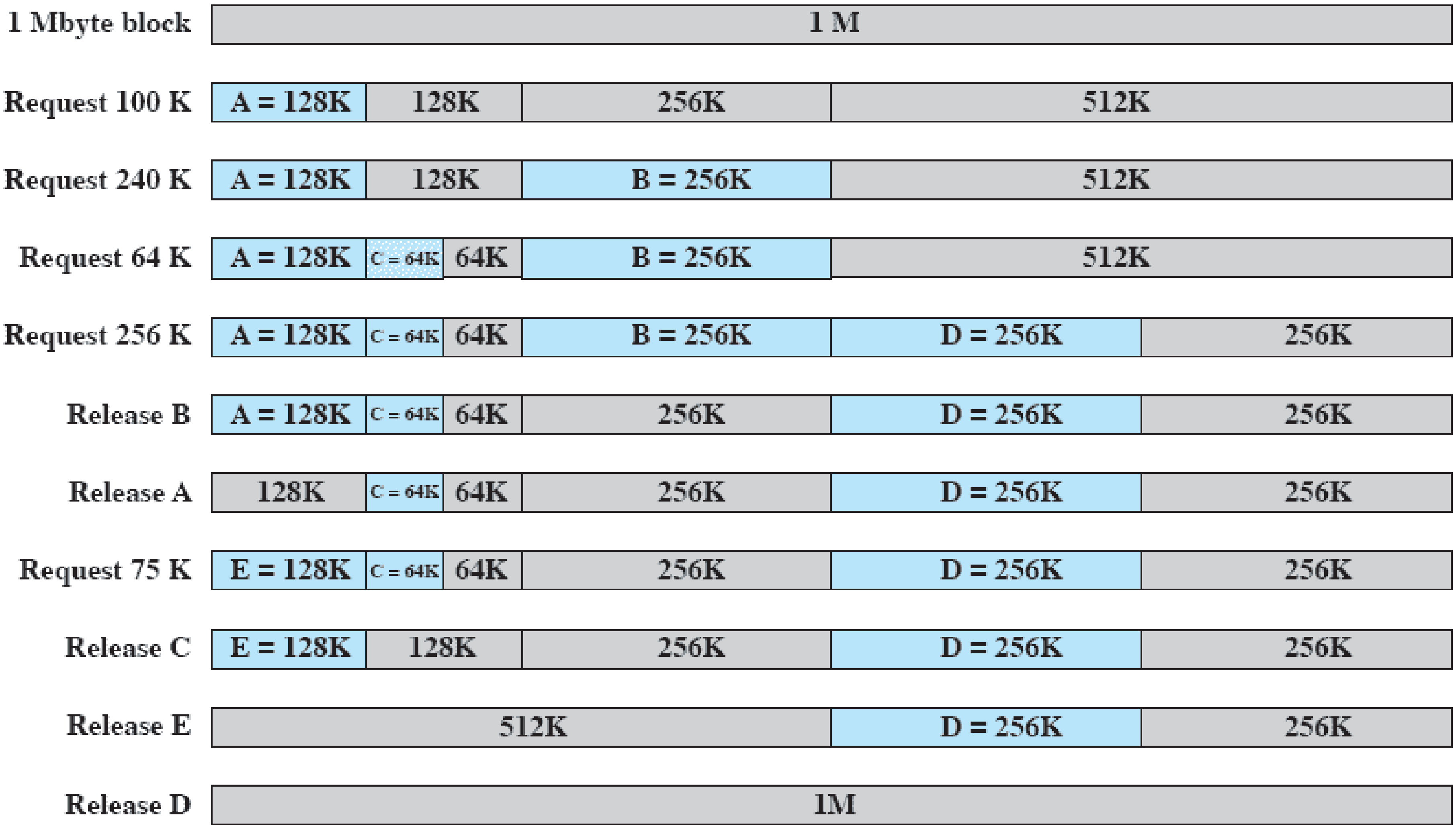
- **Problem 5**
  - Show how the available memory of 810 KB will accommodate following job sequence with all the four placement algorithms.
  - Job Sequence:

    J1 (90K),J2 (45K), J3 (180K), J4 (90K), J5 (135K), J6 (180K), J3 terminates, J5 terminates, J7 (135K), J8 (180K), J7 AND J8 TERMINATE, J9(285K).
- **Problem 6**
  - Show how the available memory of 2560 KB will accommodate following job sequence with all the four placement algorithms. OS Kernel takes 400 KB.
  - Job Sequence:
  - P1 (600K), P2 (1000K), P3 (300K), P2 terminates, P4 (700K), P5 (500K).
  - Draw the pictorial representation using MVT

BUDDY SYSTEM OF PARTITIONING

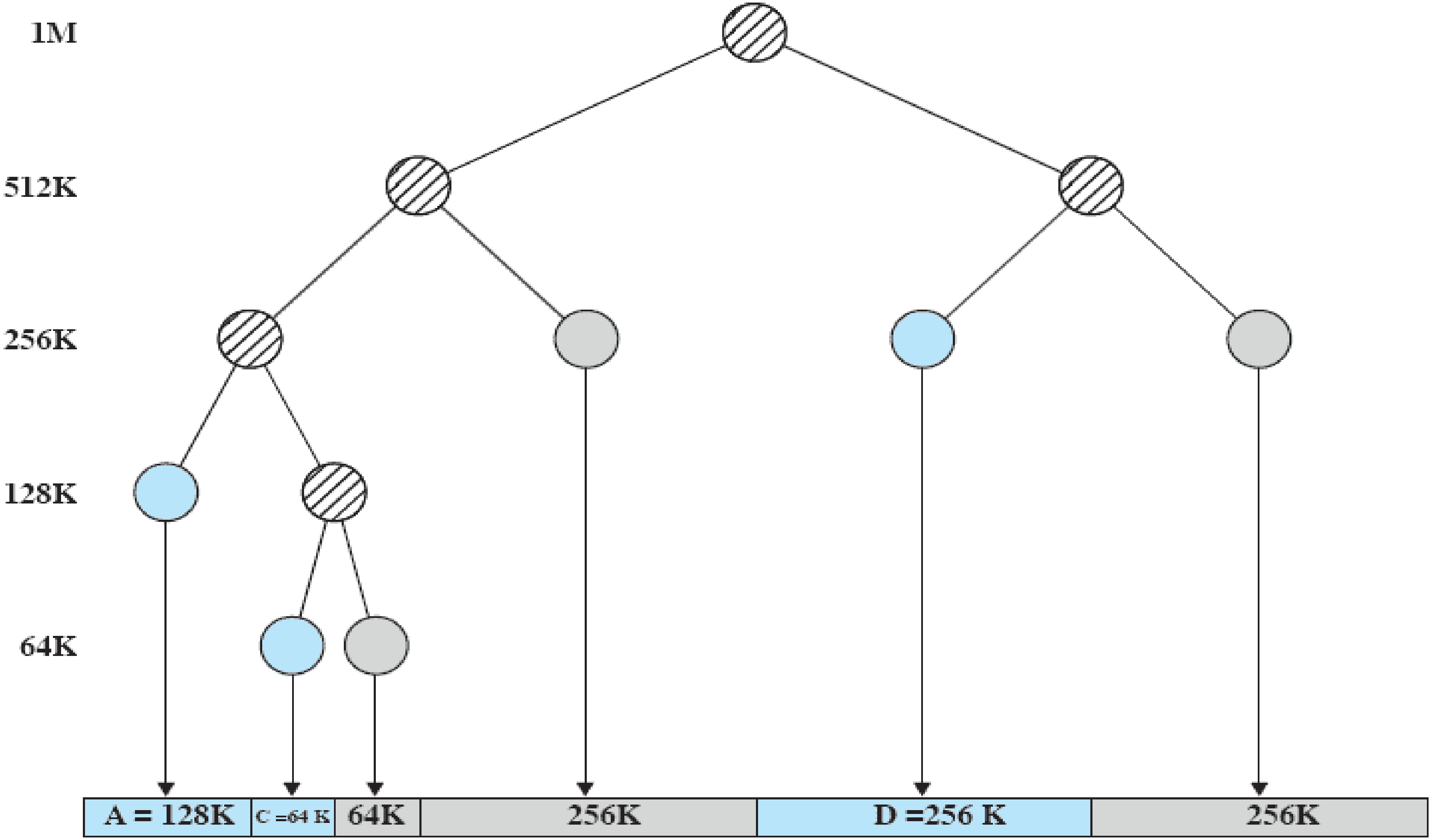
- We have seen that MFT suffers from internal fragmentation while MVT suffers from external fragmentation
- The buddy system of partitioning relies on the fact that space allocations can be conveniently handled in sizes of power of 2
- There are two ways in which the buddy system allocates space
  - Suppose we have a free slot which is the closest power of two. In that case, that free slot is used for allocation
  - In case we do not have that situation then we look for the next power of 2 free slot, split it in two equal halves and allocate one of these
- Because we always split the free slots in two equal sizes, the two are buddies. Hence the name buddy system
- At any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i
- When ever a pair of buddies on the i list both become unallocated, they are removed from the list and coalesced into a single block on the i+1 list

EXAMPLE BUDDY SYSTEM OF PARTITIONING



1 Mbyte block	1 M					
Request 100 K	A = 128K	128K	256K	512K		
Request 240 K	A = 128K	128K	B = 256K	512K		
Request 64 K	A = 128K	C = 64K	64K	B = 256K	512K	
Request 256 K	A = 128K	C = 64K	64K	B = 256K	D = 256K	256K
Release B	A = 128K	C = 64K	64K	256K	D = 256K	256K
Release A	128K	C = 64K	64K	256K	D = 256K	256K
Request 75 K	E = 128K	C = 64K	64K	256K	D = 256K	256K
Release C	E = 128K	128K	256K	D = 256K	256K	
Release E	512K	D = 256K	256K			
Release D	1M					

TREE REPRESENTATIN OF BUDDY SYSTEM



BUDDY SYSTEM OF PARTITIONING

- With 1024 K or (1M) storage space we split it into buddies of 512 K, splitting one of them to two 256 K buddies and so on till we get the right size. Also, we assume scan of memory from the beginning. We always use the first hole which accommodates the process.
- Otherwise, we split the next sized hole into buddies. Note that the buddy system begins search for a hole as if we had a number of holes of variable sizes. In fact, it turns into a dynamic partitioning scheme if we do not find the best-fit hole initially.
- The buddy system has the advantage that it minimizes the internal fragmentation. In practice, some Linux flavors use it.

SAMPLE PROBLEMS

- **Problem 8**
  - Show how the available memory of 1MB will be allocated using Buddy Memory Allocation scheme.
  - (P1:100K);(P2:240K);(P3:64 K); (P4:256 K). P2 terminates. P1 terminates. (P5:75K). P3 terminates. P4 terminates. P5 terminates.
- **Problem 9**
  - Consider a swapping system in which memory consists of the following hole sizes in memory order: 10, 4, 20, 18, 7, 9, 12 and 5 KBs
  - Which hole is taken for successive segment requests of
    - a. 12 KB
    - b. 10 KB
    - c. 9 KB
  - for First Fit?
  - Repeat the question for Best Fit, Worst Fit and Next Fit.

SAMPLE PROBLEMS

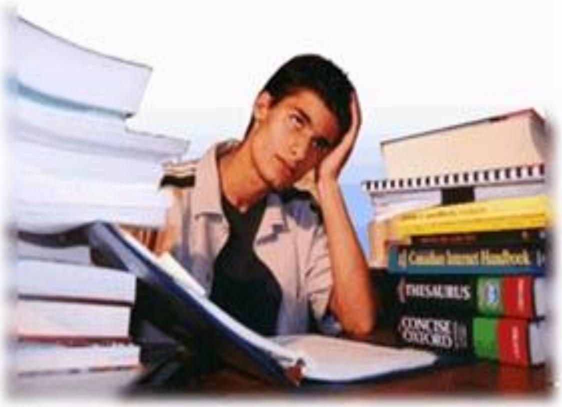
Problem 10

A swapping system eliminates empty slots by compaction. Assuming a random distribution of many empty slots and many data segments. Time to read or write a 32 bit memory word of is 10 nsec. How long does it take to compact 128 MB? For simplicity, assume that word 0 is part of an empty slot and that the highest word in memory contains valid data.

Problem 11

Given five memory partitions of 100, 500, 200, 300 and 600 KB (in order). How would each of the First Fit, Best Fit and Worst Fit algorithms place processes of 212 Kb, 417 Kb, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

We're done for now, but Todo's for you after this lecture…



- Go through the slides and Book Sections: 8.1, 8.2, 8.3
- Solve all the sample problems given in slides to understand the concepts discussed in class