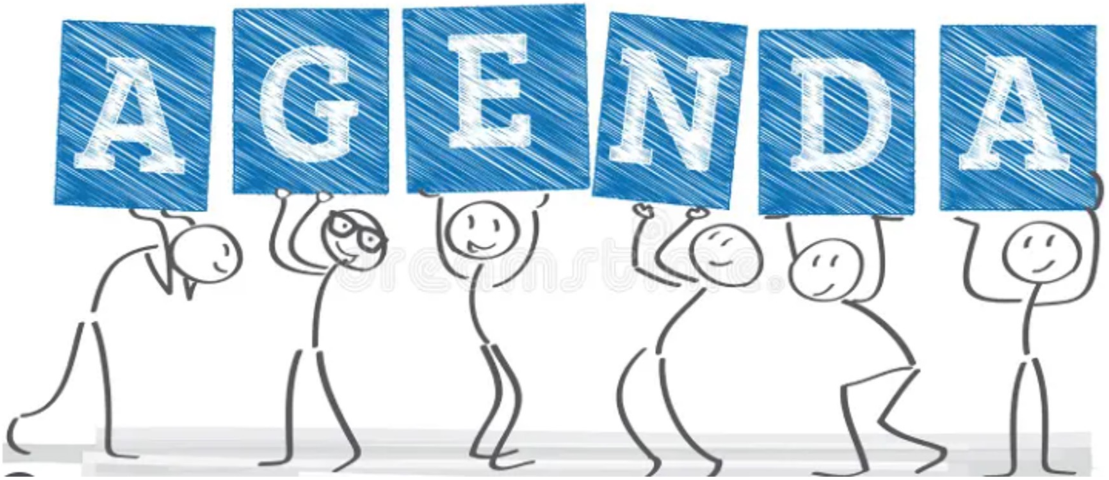
Operating Systems

Lecture 6.1

Contiguous Memory Allocation Schemes

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

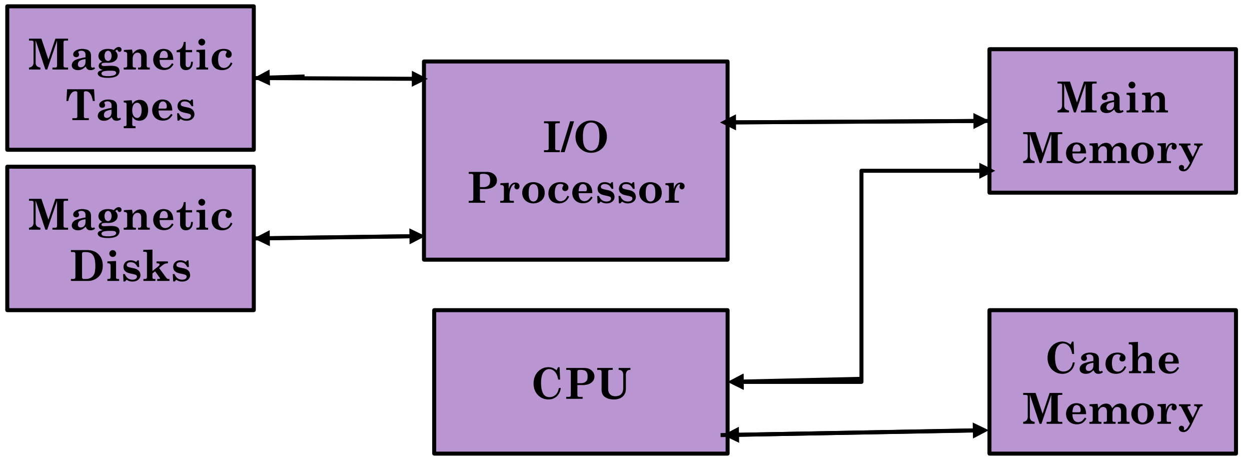


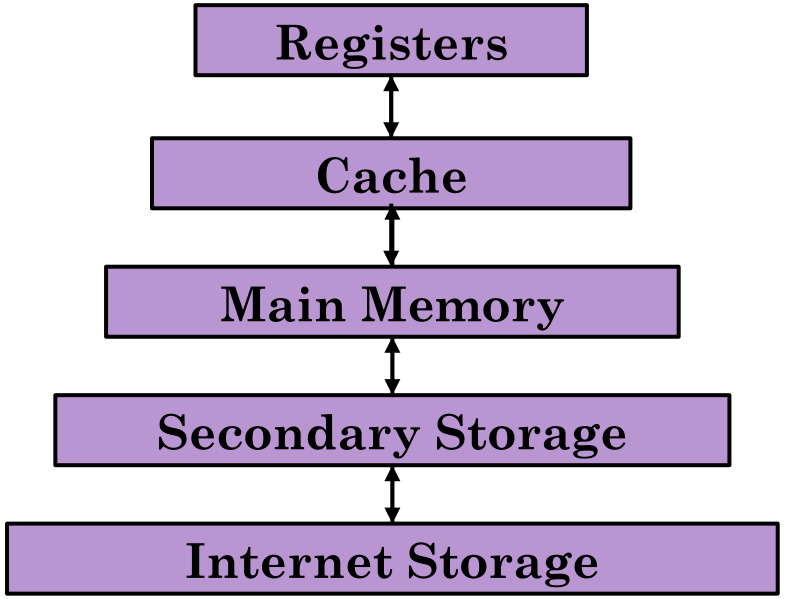
- Overview of Memory
- Swapping
- Overlaying
- Address Binding
  - Program-Time
  - Compile-Time
  - Load-Time
  - Run-Time
- Overview of Memory Allocation Techniques
  - Contiguous
  - Non-Contiguous
- Address Translation in Contiguous Memory
- Contiguous Memory Allocation Techniques
  - MFT
  - MVT
  - Buddy Partitioning Scheme

Memory Overview

Memory Hierarchy

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





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/deallocate 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 deallocate
  - Swapping processes in and out of main memory

- **Von Neumann vs. Harvard Architecture**
  - In Von Neumann, we have a unified memory space for instructions and data. Simpler memory management but potential security concerns.
  - In Harvard, we have separate memory spaces for instructions and data. Enhanced security but increased complexity in memory allocation.

Address Space Abstraction (Illusion vs Reality)

Aspect	Application View (Illusion)	Hardware View (Reality)	Enabling Mechanism
Memory Size	Giant, virtually unlimited address space	Limited physical RAM (typically GBs)	Virtual memory + demand paging
Memory Ownership	Each process has private, dedicated memory	All processes share same physical memory pool	Memory Management Unit (MMU)
Address Layout	Consistent, predictable address ranges	Fragmented, dynamic physical addresses	Address translation
Memory Access	Direct access to any virtual address	Complex mapping through page tables	Hardware address translation
Memory Protection	Complete isolation from other processes	Shared memory requiring active protection	Access control bits + privilege levels
Memory Expansion	Memory appears infinitely expandable	Fixed physical capacity, expansion via storage	Swapping to disk
Concurrency	Each process runs in isolation	Multiple processes compete for same resources	Context switching + memory mapping

Swapping

Swapping

Swapping is a memory management technique where a process is temporarily moved out of main memory to a **backing store** (usually a disk) and later brought back for continued execution. This allows the operating system to manage memory more efficiently, especially under multitasking or priority-based scheduling.

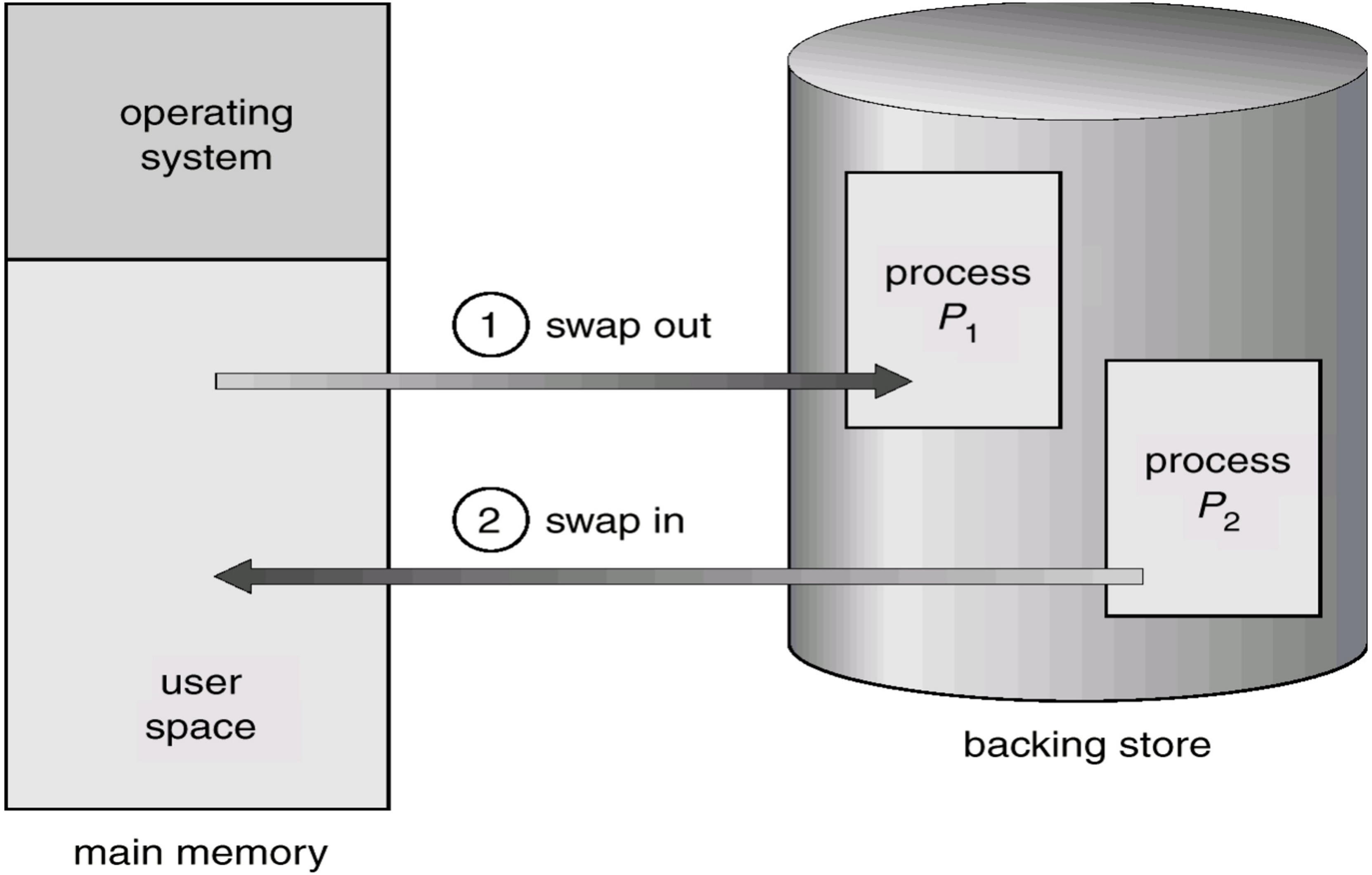
Backing Store

- A **fast disk** large enough to hold memory images of all active processes.
- Must support **direct access** to memory images for quick retrieval.
- Acts as temporary storage during process swapping.

Roll Out, Roll In

- A **variant of swapping** used in **priority-based scheduling**.
- **Roll Out**: Lower-priority process is swapped out to free memory.
- **Roll In**: Higher-priority process is loaded into memory for execution.

Schematic View Of Swapping



Example: Swapping Cost Calculation

Given the following data, calculate the total swapping cost in time:

- **Process Size** = 1 MB
- **Transfer Rate** = 5 MB/sec
- **Disk Latency** = 8 milliseconds (0.008 sec)
- **Seek Time** = 0 (ignored)

Swapping involves **two transfers**:

- **Swap Out** (Memory → Disk)
- **Swap In** (Disk → Memory)

- Each transfer cost:
  Transfer Time = Process Size / Transfer Rate = 1 / 5 = 0.2 sec
  Total Time Per Transfer = Transfer Time + Latency = 0.2 + 0.008 = 0.208 sec

- So, total cost for **both swap out and swap in**:
  Total Swapping Cost = 2 × 0.208 = 0.416 sec

Overlaying

Overlaying

Overlaying is a memory management technique that allows a process to execute even when its total size exceeds the available physical memory. It works by retaining only the currently required instructions and data in memory, while dynamically loading and unloading other parts as needed. This approach is manually implemented by the programmer and is especially useful in systems with limited memory resources.

Key Characteristics

- Only essential code and data segments are kept in memory at any given time.
- When new segments are needed, they replace the ones that are no longer required.
- The operating system does not manage overlays; instead, a **user-defined overlay driver** controls the loading and unloading of segments.

Example: Two-Pass Assembler

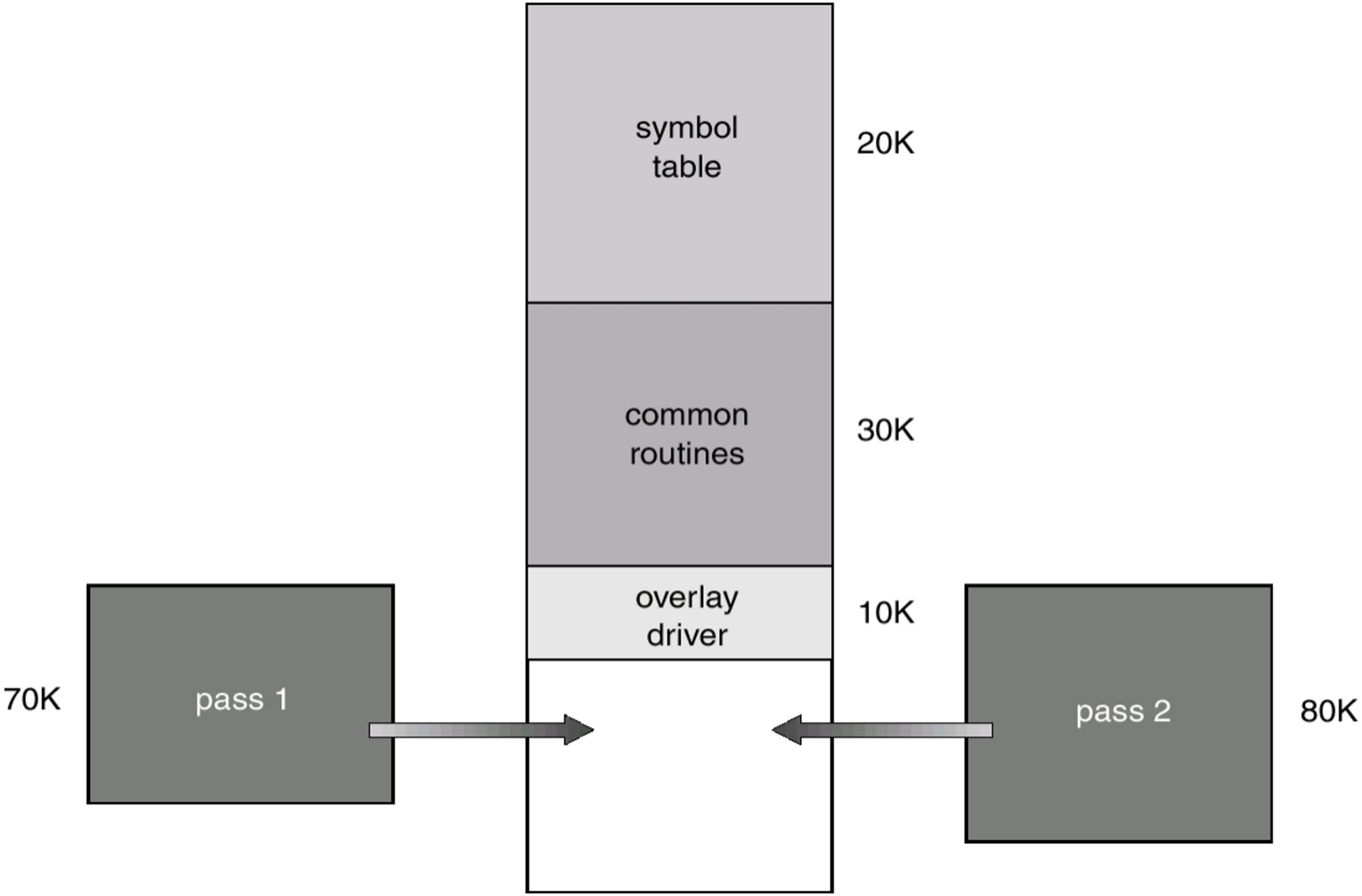
Consider a two-pass assembler with the following memory requirements:

Components	Size (in KB)	Description
Pass 1	70 KB	Performs parsing and syntax analysis
Pass 2	80 KB	Generates object code
Symbol Table	20 KB	Stores language grammar; built during Pass 1
Common Routines	30 KB	Shared functions used by both passes

- **Total Process Size**: 200 KB
- **Available Memory**: 150 KB
- **Overlay Required**: Yes

Since the total process size exceeds the available memory, overlaying is used to manage execution efficiently.

Schematic View of Overlaying

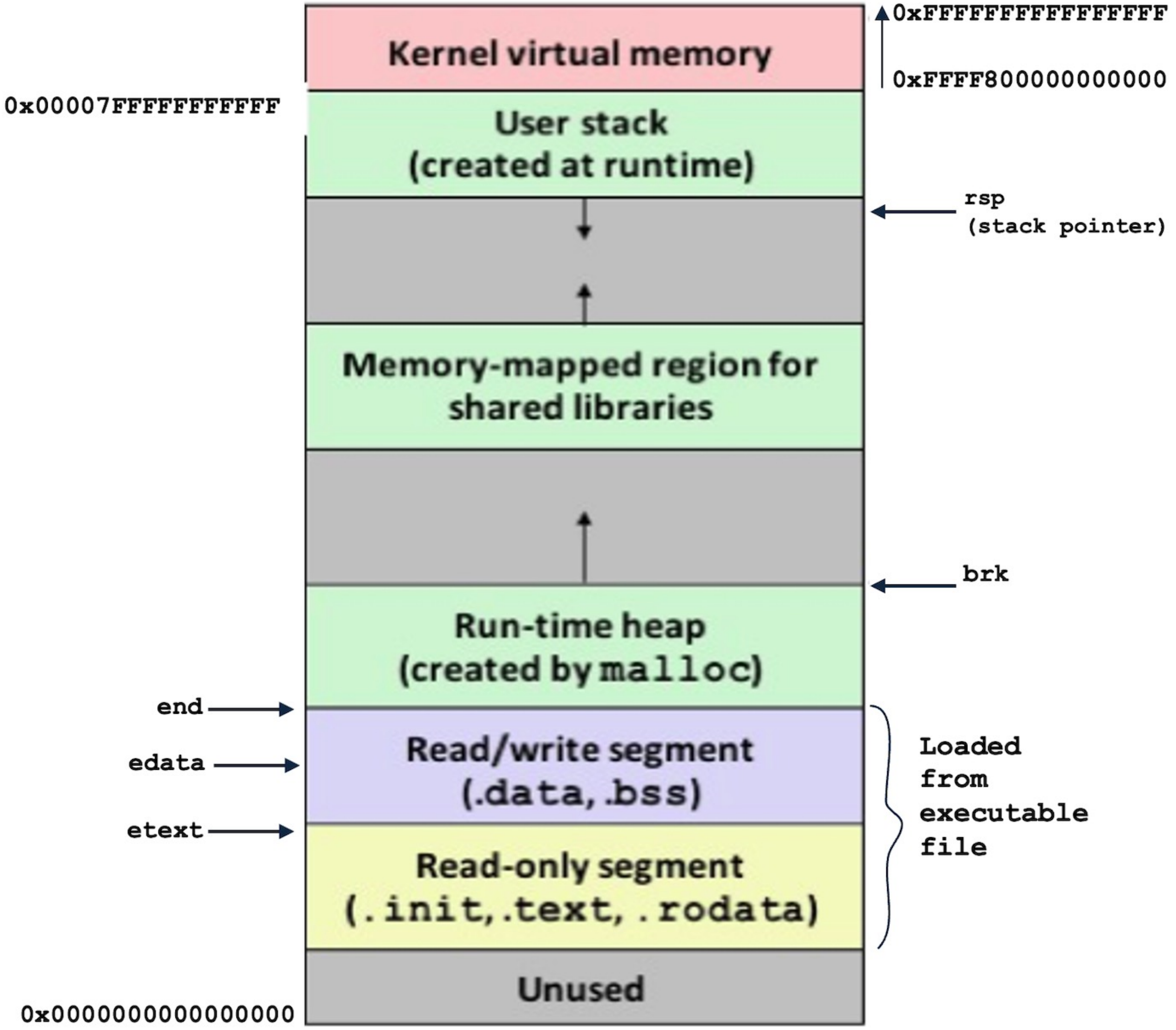


Address Binding

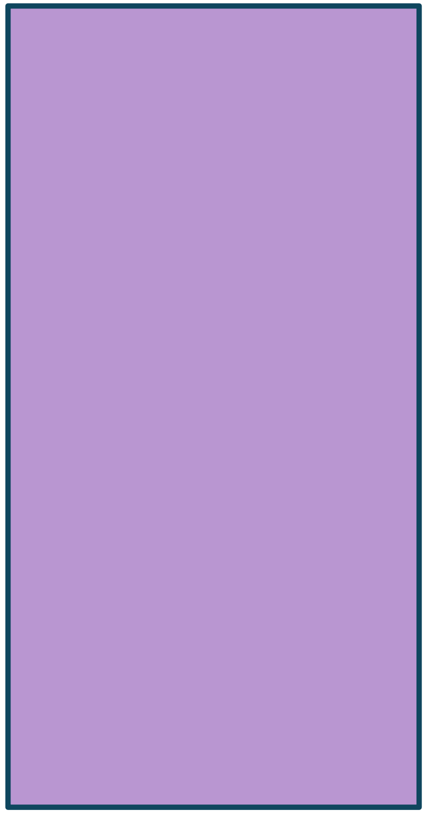
Loading a Binary to Physical Memory

Program Loading is a process of copying a program from disk to main memory in order to make it a process

Executable Object File
ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)

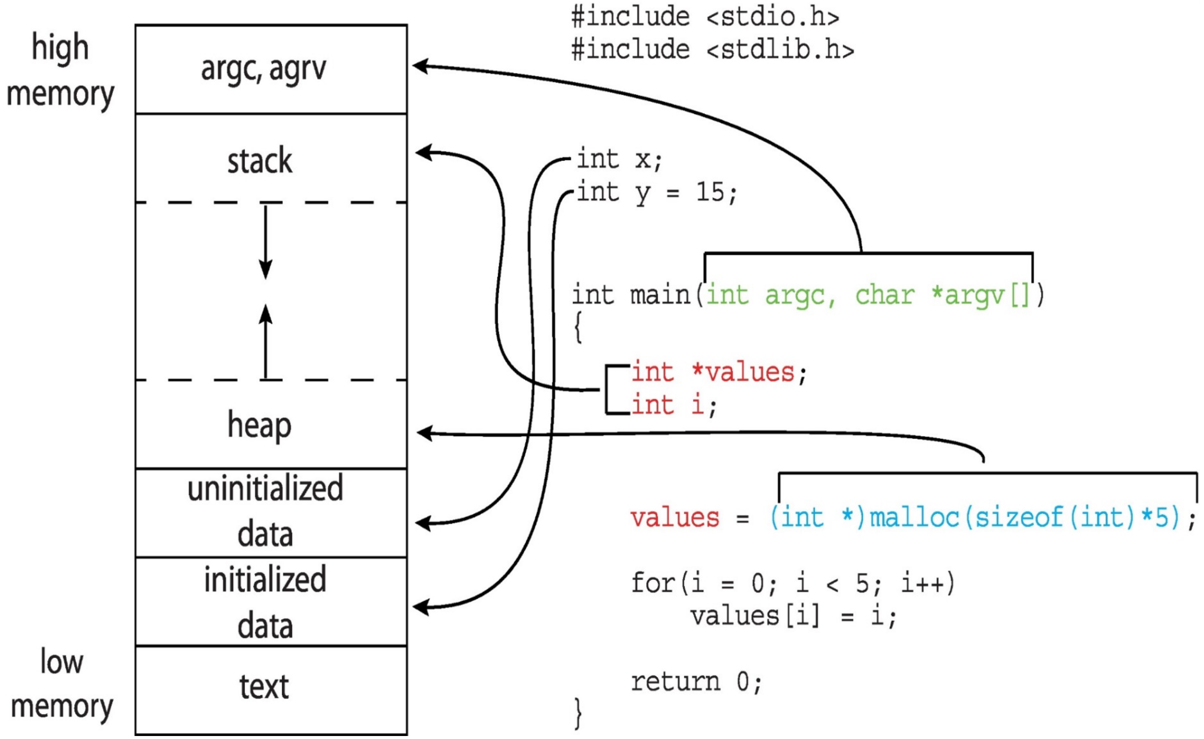


Physical Memory



Core Components of Process Address Space

- We have seen as how a source file written in C is transformed to a binary in elf format and how that binary is loaded into memory.
- The logical address space is shown in the opposite image having different sections of the process address space.
- Remember the addresses of a process logical address space starts from zero on wards.



```
#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;

    return 0;
}
```

Components	Description	Purpose	Memory Characteristics
Program Code	Executable instructions	Store compiled program logic	Read-only, shared across instances
Global/Static Data	Variables with program lifetime	Store persistent application state	Read-write, fixed size at compile time
Stack	Function calls & local variables	Manage execution context	LIFO structure, grows/shrinks dynamically
Heap	Dynamic memory allocation	Runtime memory allocation	Grows upward, managed by allocator

Overview of Address Binding

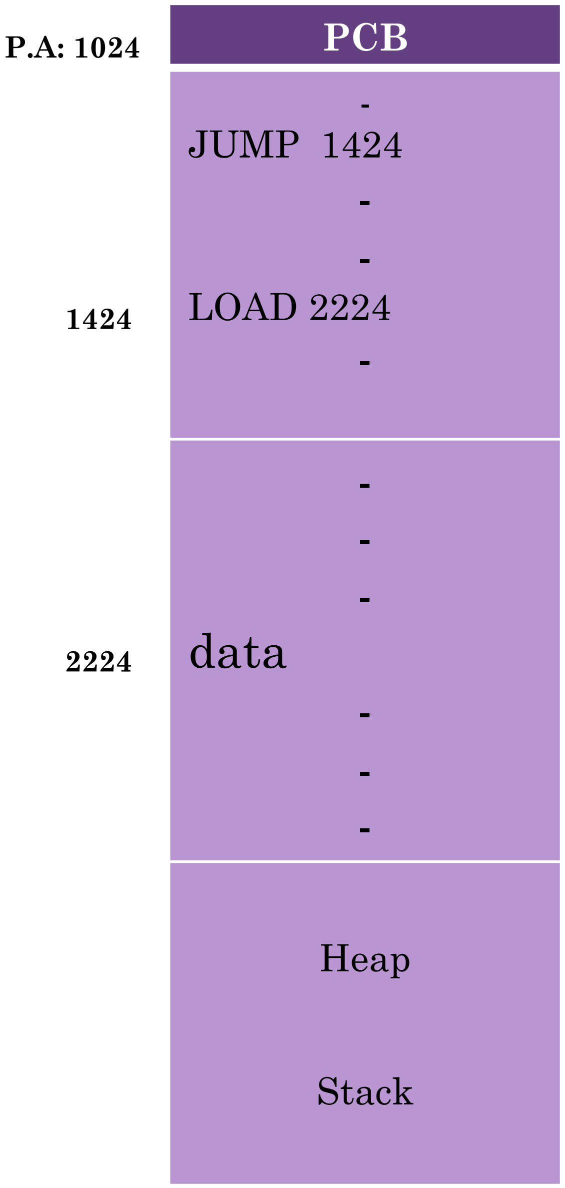
- Address binding is the process of mapping a program's logical (or virtual) addresses, generated by the CPU, to actual physical addresses in main memory.
- Since programs are typically written and compiled without knowing their exact memory location, the operating system and supporting hardware must decide *when* and *how* this mapping occurs.
- Binding can happen at different stages:
  - Program Time
  - Compile Time
  - Load Time
  - Execution/Run Time

Program-Time Address Binding

All actual physical addresses are directly specified by the programmer in the program itself

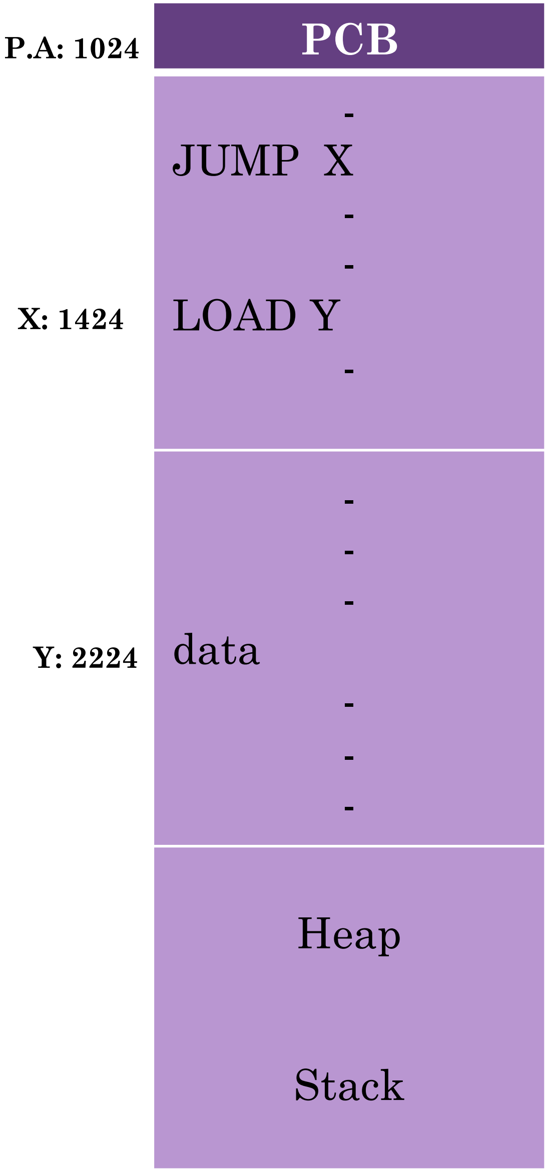
Limitation:

- Very difficult for programmers to specify address
- Used normally in uni-process programming environment



Compile-Time Address Binding

- At compile time, all symbolic and relative addresses in the program are resolved into fixed absolute physical addresses, generating an executable that assumes a known and static load location.
- If you know at compile time where the process will reside in memory, the absolute code can be generated by the compiler.
- This approach eliminates address translation overhead at runtime, making it efficient
- However, it lacks flexibility. 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
- This method is generally found in simple or embedded systems, or legacy environments like DOS `.COM` programs

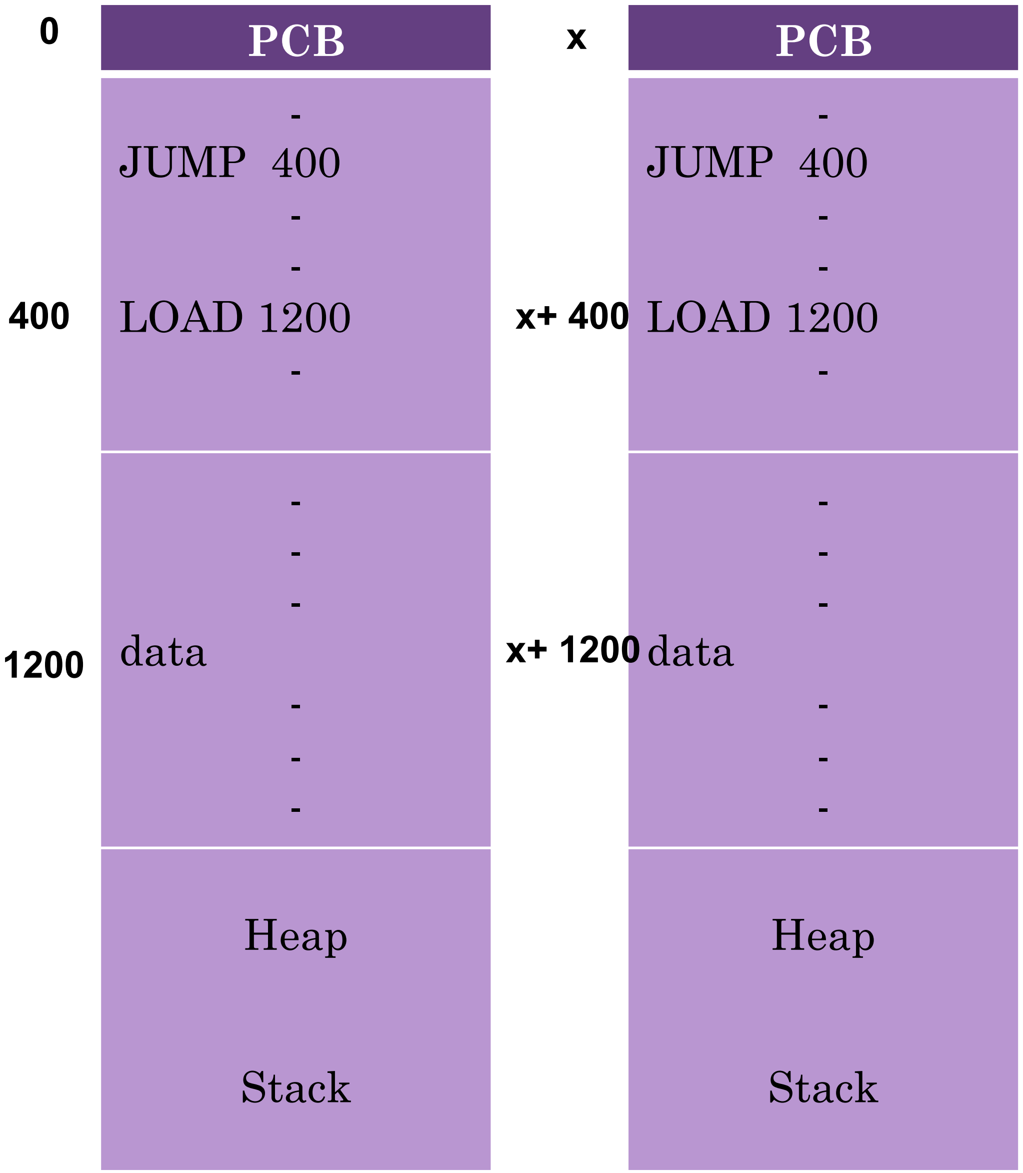


So in program time as well as in compile time address binding, we can load a process in memory if and only if the absolute addresses for instructions and data are free inside the memory

Load-Time Address Binding

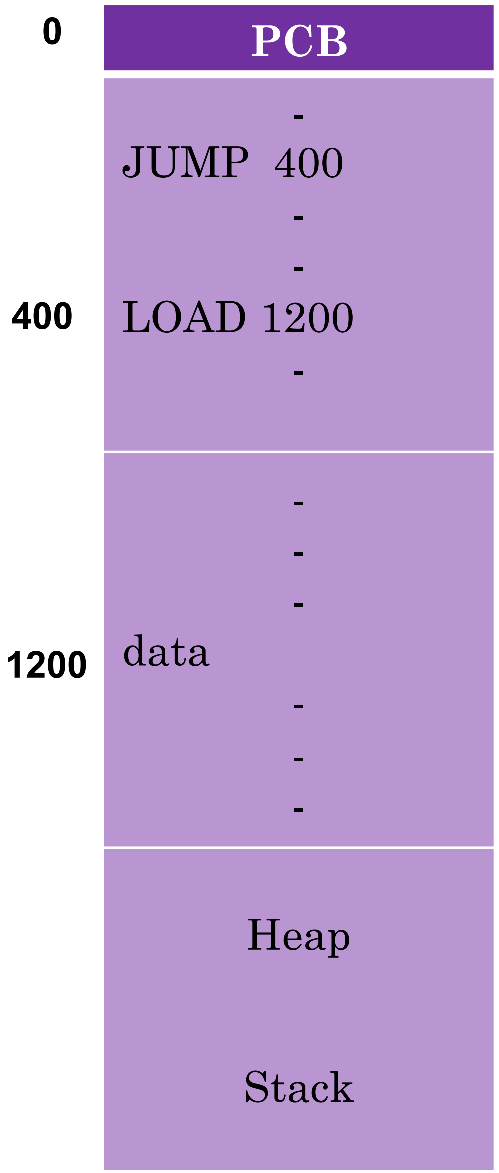
- With load-time binding, the compiler produces relocatable (relative) addresses, not fixed physical ones.
- Initially addresses within the process are relative to start address and final binding is delayed until load time.
- When the program is loaded into memory by the OS loader, it selects a free memory region and adds the appropriate base value each logical address

If the process is swapped out and then swapped in, it has to be loaded in the same memory region.



Run-Time Address Binding

- Run-time address binding defers address mapping until the program runs. Here, logical addresses are translated into physical ones dynamically, via hardware (e.g., using a base register or modern MMU mechanisms), allowing the OS to move the process in memory during execution.
- This supports advanced features like relocation, paging, swapping, and memory protection—making it the default for most modern operating systems



Logical addresses are translated dynamically to Physical addresses during execution using hardware support like base-limit registers or an MMU. More on this later ☺

Memory Allocation Techniques

Memory Allocation Techniques

Memory allocation schemes used by operating systems can be broadly divided into two main categories:

Contiguous Memory Allocation:

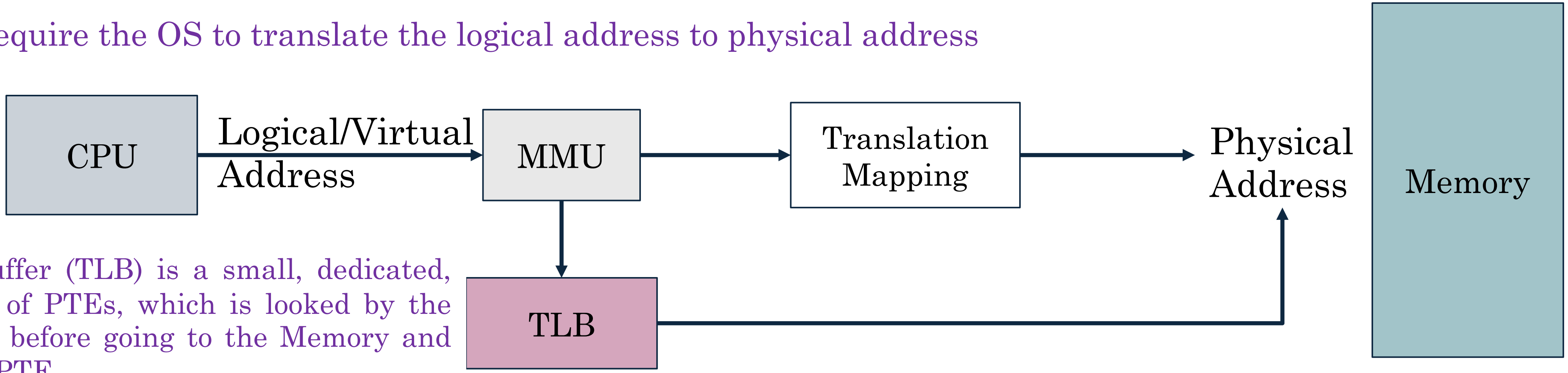
- Multiprogramming with Fixed Tasks (MFT)
- Multiprogramming with Variable Tasks (MVT)
- Buddy Partitioning

Non Contiguous Memory Allocation:

- Segmentation
- Paging
- Paged Segmentation

The Memory Management Unit (MMU) is a **hardware unit** (not part of OS) built into the CPU (in modern x86-64 processors) It's silicon circuitry that performs address translation (LA to PA) at hardware speeds.

All these techniques require the OS to translate the logical address to physical address



Translation Look-aside Buffer (TLB) is a small, dedicated, superfast hardware cache of PTEs, which is looked by the MMU to translate the LA before going to the Memory and look for the corresponding PTE.

Memory Allocation Techniques

Allocation Techniques	Description	Advantages	Disadvantages
MFT	Memory divided into fixed-sized partitions, each assigned to a process.	Simple implementation, predictable partitioning.	Internal fragmentation (unused space within partitions), inflexible, multiprogramming limited.
MVT	Partitions dynamically sized to process needs; allocated at runtime.	More efficient memory use compared to fixed partitions	External fragmentation, complex allocation and compaction required.
Buddy System	Divides memory into power-of-two blocks; splits and coalesces “buddy” blocks as needed.	Reduces external fragmentation, fast allocation/deallocation, relatively easy to implement	Internal fragmentation due to rounding up size; still some memory waste; limited to power-of-two sizes.
Segmentation	Memory divided into variable-sized logical segments (e.g. code, data, stack).	Logical program organization, protection by segment, dynamic allocation	External fragmentation, complex address translation, harder to manage.
Paging	Divides both memory and processes into fixed-size pages and frames; uses page tables.	No external fragmentation, straightforward allocation, isolation between processes	Internal fragmentation (unused space in pages), page table overhead, address translation cost.
Paged Segmentation	A hybrid: segments (logical units) that are further divided into pages.	Logical grouping (like segmentation) + fragmentation reduction and protection (like paging)	High complexity, two-level translation overhead, management complexity.

Address Translation
in
Contiguous Memory Allocation

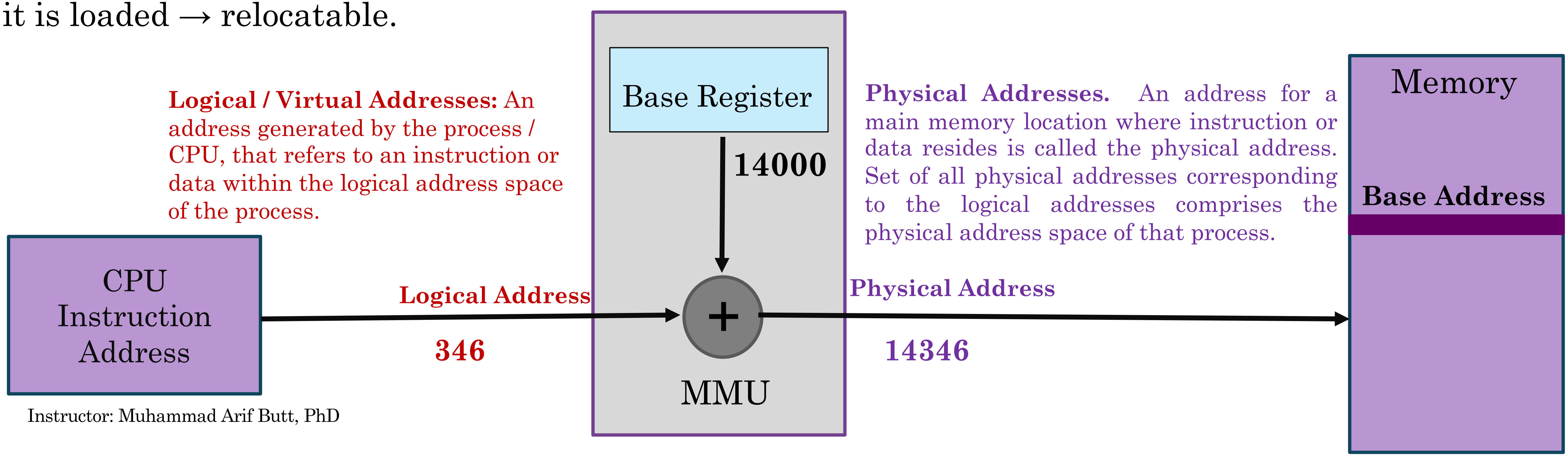
Contiguous Memory Allocation

A memory management technique in which the entire process is loaded into a single continuous block of physical memory. This approach simplifies address translation and improves access speed but can lead to fragmentation and inefficient memory utilization.

Address Translation using Base Register

Base and limit registers are provided by the CPU/MMU for **relocation** and **protection**. They don't dictate *how* memory is divided among processes, rather are used to translate the logical address to its corresponding physical address in contiguous memory allocation schemes.

Relocation: A process is compiled assuming it will start at address 0. But in reality, the OS can load it anywhere in physical memory. This problem is resolved using the Base Register that holds the starting physical address of the process's allocated memory block. The CPU adds this base value to every logical address generated by the process. This way, the same program can run no matter where it is loaded → relocatable.



Logical / Virtual Addresses: An address generated by the process / CPU, that refers to an instruction or data within the logical address space of the process.

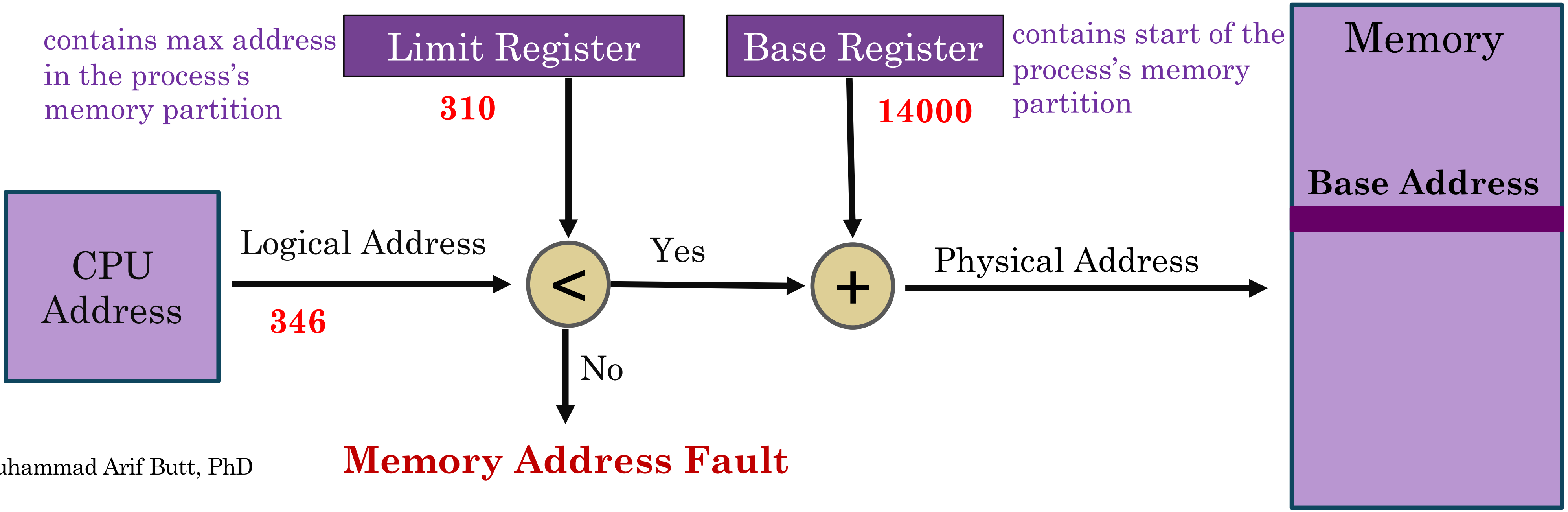
Physical Addresses. An address for a main memory location where instruction or data resides is called the physical address. Set of all physical addresses corresponding to the logical addresses comprises the physical address space of that process.

Translation using Base + Limit Register

Protection: A process should not access memory outside its allocated block, otherwise it might overwrite another process's data or OS memory. To ensure that a process stays inside its own allocated region, we use the limit register that stores the maximum address in the process's memory partition. Every logical address generated by the CPU is checked:

- If logical address < limit, it is valid.
- If logical address ≥ limit, it triggers a memory protection fault (segmentation fault).

In multi-programming scenarios, every process has its own value of base and limit register. So after a context switch, the appropriate values are loaded in these two registers by the OS.



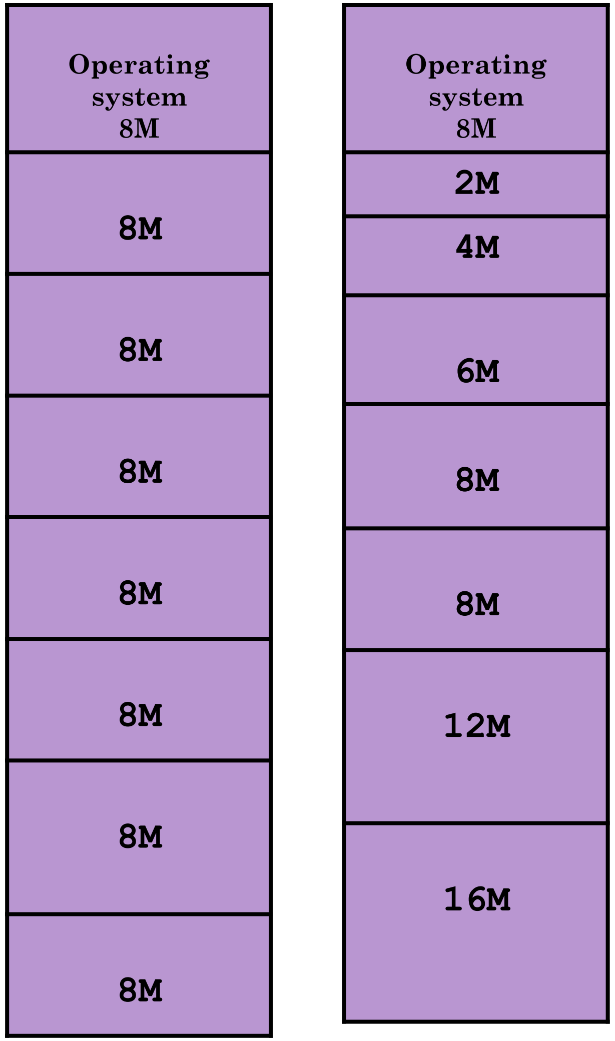
Multiple Partitioning with Fixed Tasks

Multiprogramming with Fixed Tasks (MFT)

- In **MFT** memory is divided into several fixed number of partitions (either uniform or varied) at system start-up. Each partition holds exactly one process, enabling simple multiprogramming. This is simple but leads to internal fragmentation (unused space inside partitions). Each process has a logical address space starting at 0, mapped to a contiguous physical address elsewhere in memory. When a partition becomes free, a process from the input queue is loaded into it. Upon termination, the partition is reused.

- **Equal-Size Partitions:** Memory is divided into uniform partitions. Any process ≤ partition size can be loaded into an available partition. If no process is ready/running, the OS may swap out a process to free space. Programs larger than partition size require overlay design by the programmer. Leads to internal fragmentation as small programs waste unused partition space.

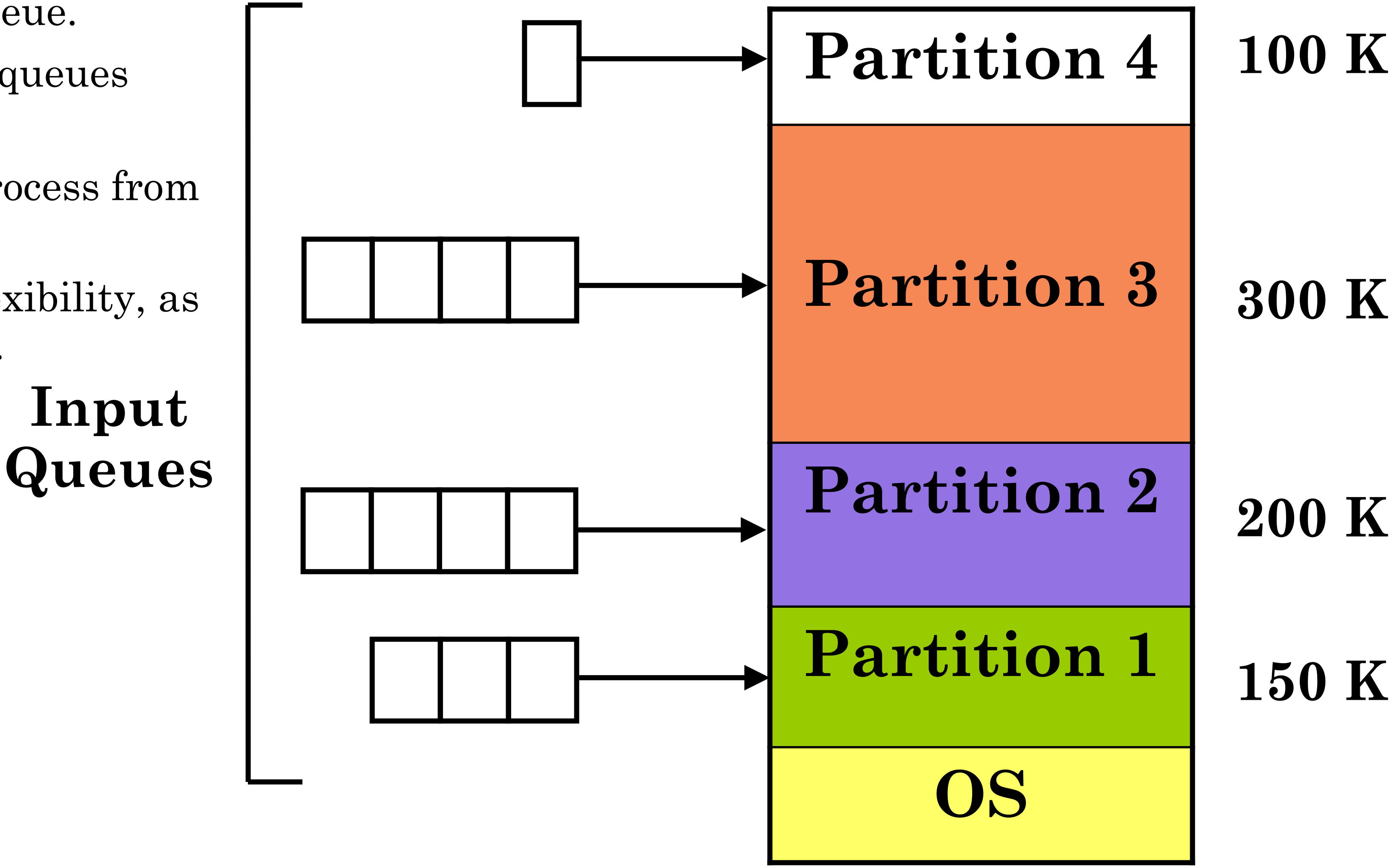
- **Unequal-Size Partitions:** Partitions are of varying sizes, tailored to typical program sizes. Reduces internal fragmentation by placing smaller programs in smaller partitions. In the figure, programs up to 16 MB can be accommodated without overlays.



Equal size partitions Unequal size partitions

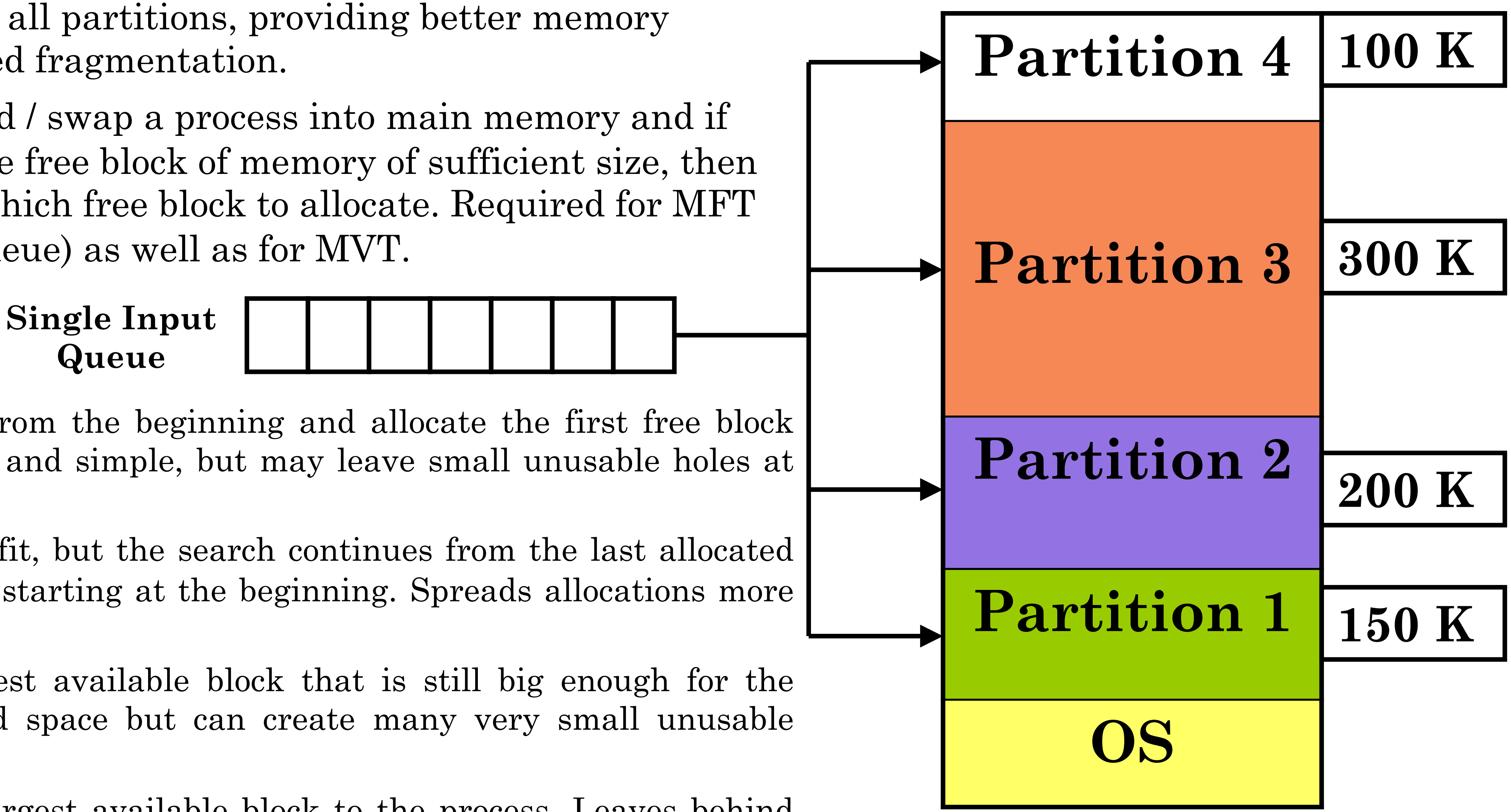
MFT with Multiple Input Queues

- Each partition has its own input queue.
- Incoming processes are placed into queues based on size compatibility.
- When a partition becomes free, a process from its dedicated queue is loaded.
- Offers fast allocation but limited flexibility, as no placement algorithm is required.



MFT with Single Input Queue

- A single queue serves all partitions, providing better memory utilization and reduced fragmentation.
- 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. Required for MFT (using single input queue) as well as for MVT.



First Fit: Scan memory from the beginning and allocate the first free block that is large enough. Fast and simple, but may leave small unusable holes at the beginning.

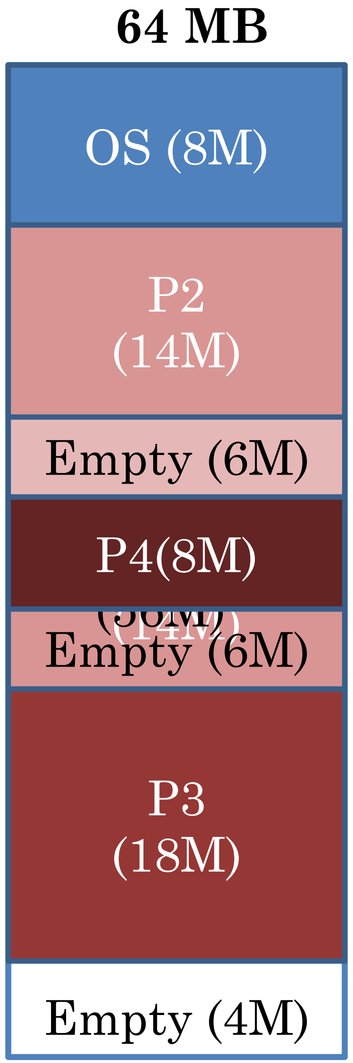
Next Fit: Similar to first fit, but the search continues from the last allocated position instead of always starting at the beginning. Spreads allocations more evenly across memory.

Best Fit: Find the smallest available block that is still big enough for the process. Minimizes wasted space but can create many very small unusable holes.

Worst Fit: Allocate the largest available block to the process. Leaves behind big free spaces that may be useful later, but can also waste memory.

Multiple Partitioning with Variable Tasks

Multiprogramming with Variable Tasks (MVT)



In MVT, memory partitions are created dynamically at runtime, sized exactly to fit each process. Both the number and size of partitions vary over time. Processes can be swapped out and reloaded into different partitions, offering flexibility. Minimizes internal fragmentation, though not entirely eliminated.

External Fragmentation

- Memory external to all processes is fragmented.
- Consider the memory snapshot, and suppose a new process of size 10 MB comes, it cannot be accommodated, although we do have 16 MB free memory, but that is not contiguous.
- Solution to External Fragmentation is **compaction**. OS moves all processes to the beginning of the memory, so that we have contiguous free space at the end. Compaction is time consuming and wastes CPU time.

Buddy System Of Partitioning

Buddy System of Partitioning

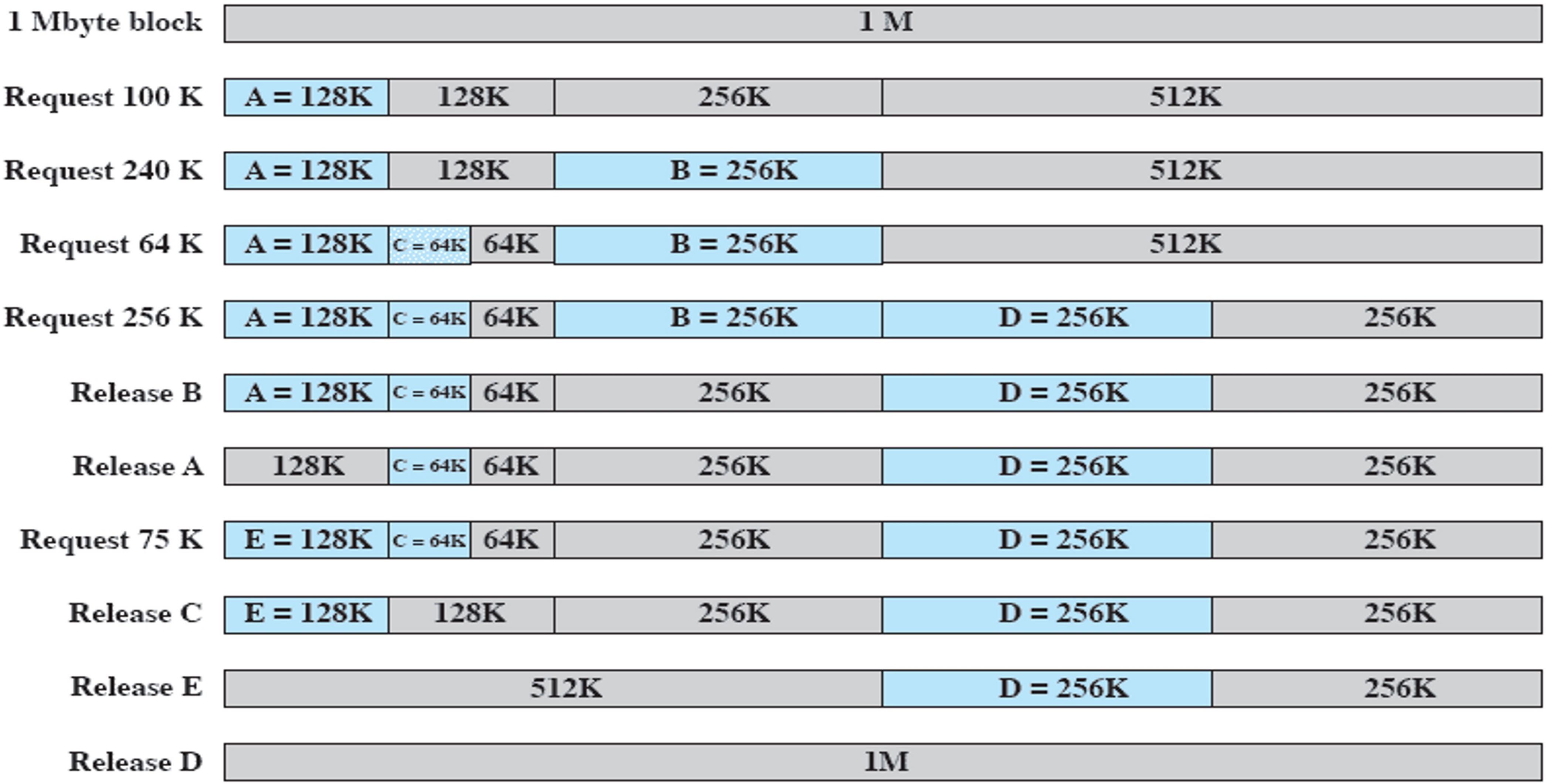
Multiprogramming with Fixed Tasks (MFT) suffers from **internal fragmentation**, while Multiprogramming with Variable Tasks (MVT) is prone to **external fragmentation**. To address these limitations, the **Buddy System** offers a dynamic partitioning approach based on memory blocks sized as powers of two.

In the Buddy System:

- Memory is allocated in blocks of size `2^i`.
- If a free block of the exact required size exists, it is allocated directly.
- If not, the system locates the next larger block, splits it into two equal halves (buddies), and allocates one half.
- These buddies are tracked in separate free lists for each block size.
- When both buddies of size `2^i` become free, they are merged (coalesced) into a single block of size `2^{i+1}`, reducing fragmentation.

This method ensures efficient memory utilization and simplifies block management through structured splitting and merging

Example



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

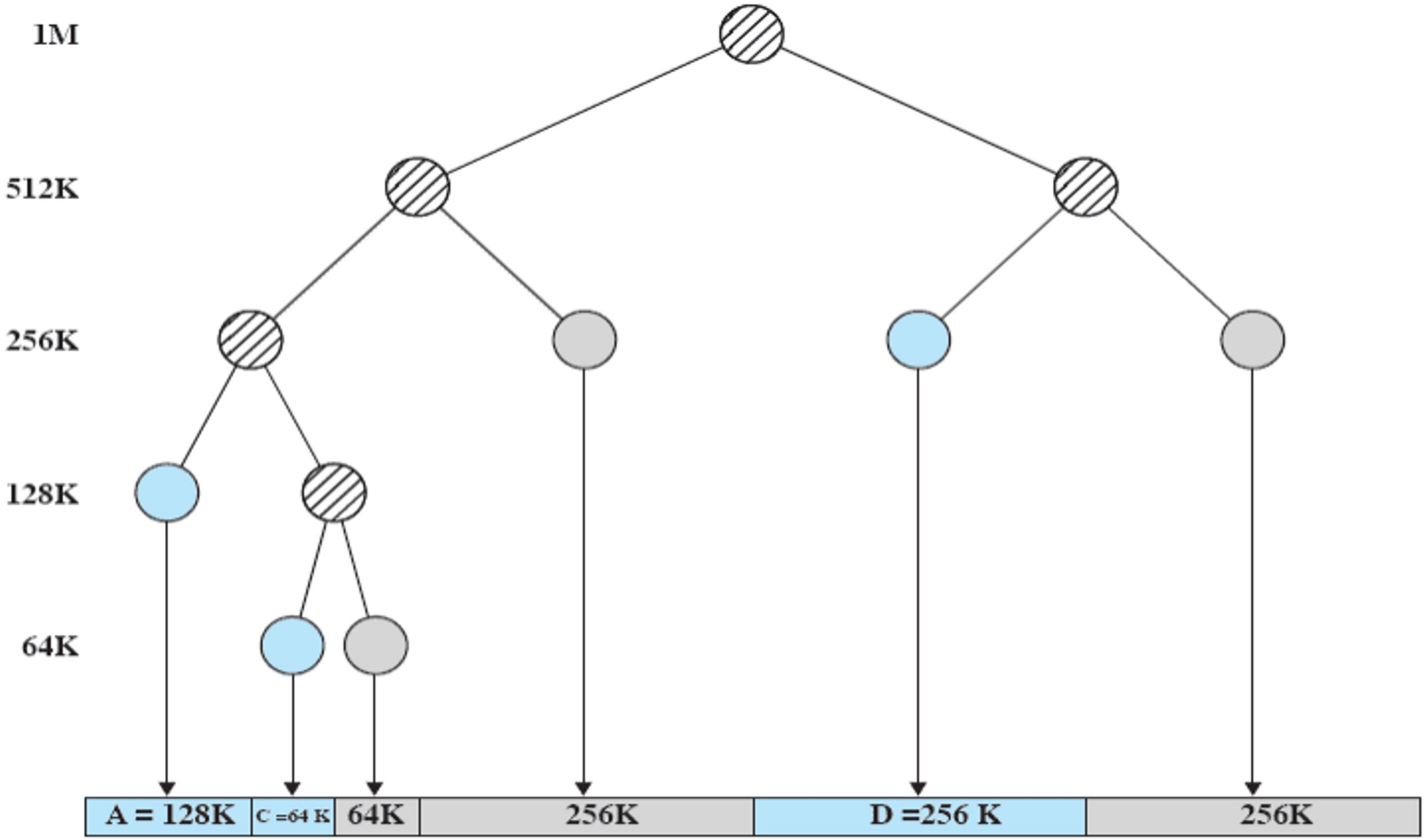
Request 75 K: E = 128K | C = 64K | 64K | 256K | D = 256K | 256K

Release C: E = 128K | 128K | 256K | D = 256K | 256K

Release E: 512K | D = 256K | 256K

Release D: 1M

Tree Representation



Sample Problems

Problem 1: A system has 256 KB memory, divided into 4 equal partitions. Show how jobs are allocated and calculate the internal fragmentation in each partition if a set of jobs arrive in sequence: `J1 (50 KB), J2 (60 KB), J3 (70 KB), J4 (120 KB).`

Problem 2: A system has 500 KB free memory. Show memory allocation step by step for First-Fit placement algorithm and calculate external fragmentation at the end. Jobs arrive in this order: `J1 (200 KB), J2 (100 KB), J3 (50 KB), J4 (175 KB)`. Jobs complete as follows: `J2 finishes first, then J3.`

Problem 3: Show how the available memory of 810 KiB will accommodate following job sequence with all the four placement algorithms using MVT.

```
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 4: Show how the available memory of 2560 KiB will accommodate following job sequence with all the four placement algorithms using MVT. OS Kernel takes 400 KiB.

```
 P1 (600K), P2 (1000K), P3 (300K), P2 terminates, P4 (700K), P5 (500K)
```

Sample Problems

Problem 5 Show how the available memory of 1MiB 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 6 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 KiBs

Which hole is taken for successive segment requests of

- 12 KiB
- 10 KiB
- 9 KiB

for First Fit?

Repeat the question for Best Fit, Worst Fit and Next Fit.

Problem 7 Show how the available memory of 1MiB 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
```

Sample Problems

Problem 8: Given five memory partitions of 100, 500, 200, 300 and 600 KiB (in order). How would each of the First Fit, Best Fit and Worst Fit algorithms place processes of 212 K, 417 K, 112 K, and 426 K (in order)? Which algorithm makes the most efficient use of memory?

Problem 9 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 MiB`? For simplicity, assume that word `0` is part of an empty slot and that the highest word in memory contains valid data.

Problem 10 A memory of 1024 KB is managed with Buddy System. Show how memory is allocated and split at each step, if process requests arrive in following order:

`P1 (120 KB), P2 (60 KB), P3 (370 KB), P4 (90 KB).`

To Do

- Carefully review all concepts discussed in class and go through the slides to build a clear understanding of contiguous memory allocation schemes.
- Review the overview of memory management concepts (swapping, overlaying, and address binding).
- Clearly differentiate between compile-time, load-time, and run-time binding, and identify when each is used.
- Understand in detail the working of MFT, MVT, and Buddy Partitioning Scheme with examples.
- Practice address translation problems in contiguous memory allocation using base and limit registers.
- Draw memory layout diagrams for MFT, MVT, and Buddy System to strengthen visualization.
- Prepare a list of possible exam-style short/long questions from today's content and try answering them without notes.
- Form small groups to discuss and cross-check answers for numerical and conceptual questions.

Coming to office hours does NOT mean that you are academically weak!