

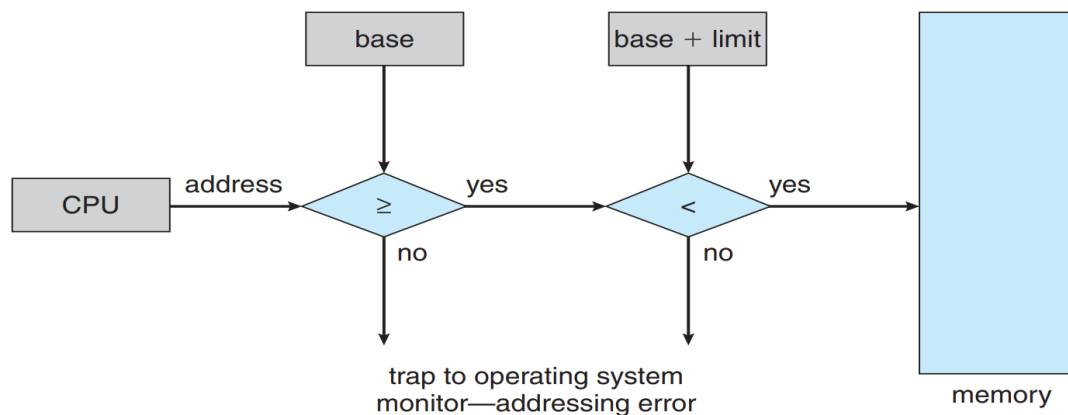
## Main Memory

### Background Information

- A **program** must be **read into memory** from **secondary storage** (for example a disk) and placed within a **process** in order to allow the program to **execute**.
- **Main memory** and **registers** are storage that is **directly accessible by the CPU only**.
- **Register access** only required a maximum of **one CPU clock cycle**.
- **Main memory** can take **several clock cycles**, and may cause a stall.
- To prevent the **main memory from stalling**, we use **cache**; cache sits inbetween the main memory, and the registers.
- **Protection** may be required to **ensure correct operation** (for example if a processor has two or more cores which share a single cache).

### Base and Limit Registers

- A pair of **base and limit registers** are used to define the **logical address space** for a process.
- The **CPU** must check every **memory access generated in user mode** to ensure it is within the **users base and limit**.



### Address Binding

- **Addresses** are represented in **different ways** at **different stages** of a **program's life**.
  - **Source code addresses** typically contain a **symbolic reference** (eg variables).
  - **Compiled code addresses** bind to **relocatable addresses** (eg 14 bytes after the start of this module).
  - The **linker or loader** will **bind relocatable addresses** to **absolute addresses** (the actual address, not an offset).
- **Address binding of instructions and data to memory addresses** can happen at three different stages:
  1. **Compile Time** — If the memory location is known prior, absolute code can be generated. This is not optimal, as it needs to be recompiled if the address changes.
  2. **Load Time** — Final binding is delayed until load time. If the starting address changes, we only reload the user code to incorporate this changed value.

3. **Execution Time** — Binding is delayed until runtime. This is done if the process can be moved during its execution from one memory segment to another. This requires hardware support for address maps (eg base and limit registers).

## Static vs Dynamic Linking

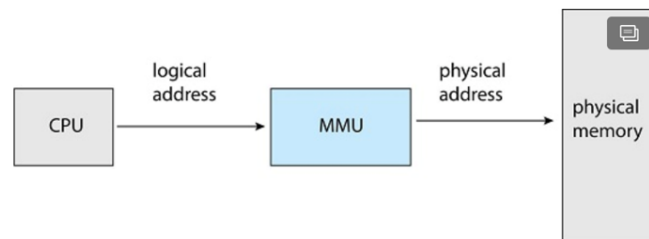
- **Static linking** is when the **code** for all **routines** called by your program become **a part of the executable file**.
- **Dynamic linking** is when the **code** for some **external routines** are **located when the program runs**.
  - **Dynamic linking** is useful for **shared libraries**.
- With **dynamic linking**, a **code stub** is used to **locate the appropriate memory-resident library routine**. The stub is replaced with the address of the routine, and executes the routine.
- If a **routine** that is being **dynamically linked** is not in the **process' address space** the operating system will **add it** to the address space.
- Not all operating systems support dynamic linking.

## Logical vs Physical Address Space

- The concept that **logical address space** is bound to **physical address space**, is essential for **proper memory management**.
- **Logical Addresses** are addresses that are **generated by the CPU** (also known as virtual addresses).
- **Physical Addresses** are addresses that are **managed by the memory unit**.
- **Logical (virtual)** and **physical** addresses have a different **execution-time address-binding** scheme.

## Memory-Management Unit (MMU)

- A **Memory-Management Unit (MMU)** is a **hardware device** that **maps virtual addresses to physical addresses at runtime**.



- A **partition table** is used to store the **allocated, and available physical memory**. Along with the **process id** of **memory that is allocated**.
- The **dynamic storage-allocation problem** refers to the problem that arises when we need to dynamically allocate memory in a partition table. The following algorithms can be used:
  1. **First-Fit** — Allocate the first hole that is big enough.

2. **Best-Fit** — Allocate the smallest hole that is big enough; must search the entire table, unless the table is ordered by size.
  3. **Worst-Fit** — Allocate the largest hole; must search the entire table, unless the table is sorted.
- Best fit is slower than first fit, but more efficient with memory use.

## Memory Fragmentation

- **External fragmentation** occurs when the **total memory space exists** to satisfy a request, but the **memory space is not contiguous**.
  - The memory is said to be **fragmented** (broken into separate parts).
- To **fix fragmentation** we use **compaction**; a very **expensive operation**.
- **Compaction** consists of **shuffling memory contents** to place **all free memory together** in one large block.
- **Compaction** is only possible if **relocation is dynamic**, and done at **execution time**.
- First-Fit, Best-Fit, and Worst-Fit all **cause fragmentation**.
- Modern operating systems instead use **virtual memory** and **segmentation** to manage memory, avoiding fragmentation.

## Virtual Memory

- **Virtual memory** is achieved by **splitting the main memory** into **fixed-size partitions** known as **page frames**.
- **Page frames are typically 4KB**.
- **Processes** are then split into **blocks of equal size** (block size = page frame size).
  - The blocks do not have to be contiguous.
- A page table is then used to map each **block** in **virtual memory** to a distinct **page** in **physical memory**.
- This **removes fragmentation**, as each virtual memory block appears to be **contiguous** to the CPU.
- When using **virtual memory**, you **do not need to load all of the program** into memory at the **start**, they can be loaded later on.
  - This is known as **demand paging**; pages are **loaded into the main memory** when they are first used.