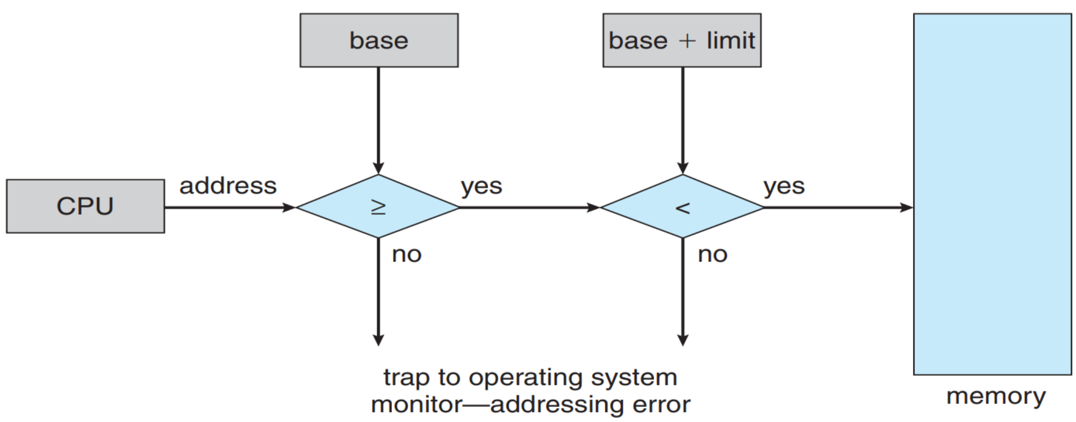
Main Memory

Background Information

- A **program** must be **read into memory** from **secondary storage** (for example a disk) and placed within a **process** inorder to allow the program to **execute**.
- **Main memory** and **registers** are storage that is **directly accessable 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 addreses** to **absolute adresses** (the actual address, not an offset).
- **Address binding** of **instructions and data** to **memory addresses** can happen a three different stages:
  1. **Compile Time** — If the memory location is known prior, absolute code can be generated. This is not optomal, 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 it's execution from one memory segment to another. This requires hardware support for address maps (eg base and limit registers).

Static vs Dynamic Linking

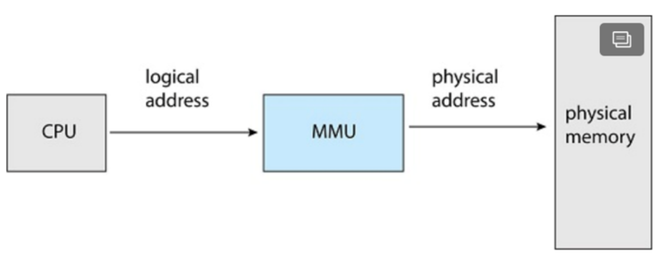
- **Static linking** is when the **code** for all **routines** called by your program become **a part of the executable file**.
- **Dynamic linking** is when the **code** for some **external routines** are **located when the program runs**.
  - **Dynamic linking** is useful for **shared libraries**.
- 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 managment**.
- **Logical Addresses** are addresses that are **generated by the CPU** (also know as virtual addresses).
- **Physical Adresses** are adresses 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 adresses** 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 partation 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 seperate 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 acheived 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** of **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.
  – If the **primary memory is full** when a **page** needs to be **loaded**, the operation system will perform a **swap**; the operating system will save a block already in ram, load the one currently needed, and when it is done the original one will be place back into memory. **Secondary memory** usually has a **swap partition** for this scenerio.
  – There are different swap replacement policies: first in first out, least recently used, least frequently used, etc.

Shared Paging

- **Common library routines** can be shared throughout **all processes** running on the operating system.
- This can be acheived with **shared pages**.

Copy on Write (COW)

- All **parent pages** are initallly marked as **shared**.
- When **data** in any of the **shared pages change** (by the parent or any child), the **OS intercepts** and makes a **copy of the change**.
- The **parent and all children** will have the **same copy of unmodified pages**.
- This is done to save memory.

Address Translation Scheme

Abstract

The **page table** consits of four columns: block, page frame, present bit, dirty bit.

The **page table** has a **page-table base register (PTBR)**.

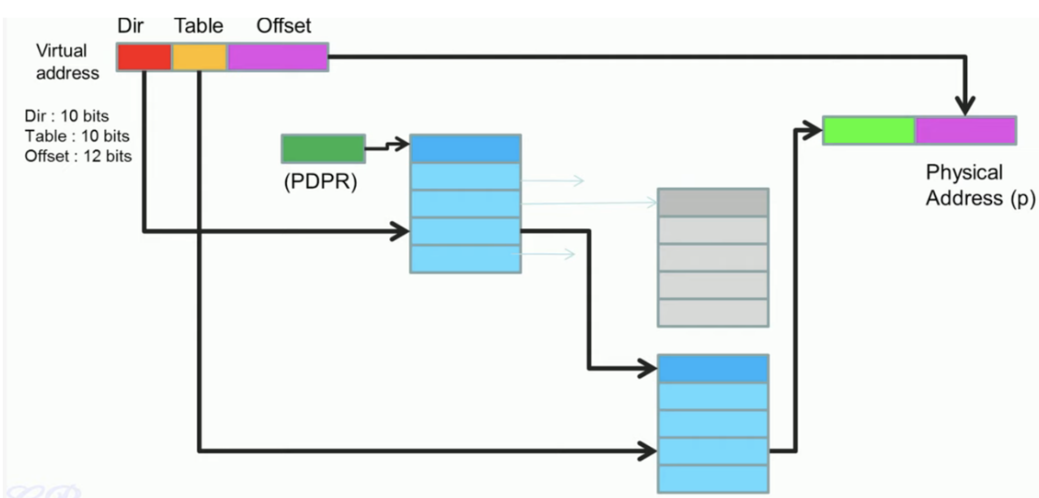
Each **virtual address** is composed of **a page number, and an offset**.

The page number refers to the virtual page number, and the offset is the number of bytes from the start of the page that the data resides.

On **32-bit systems** the **maximum process size is 4GB** (2^{32}).

Hierarchical Page Tables

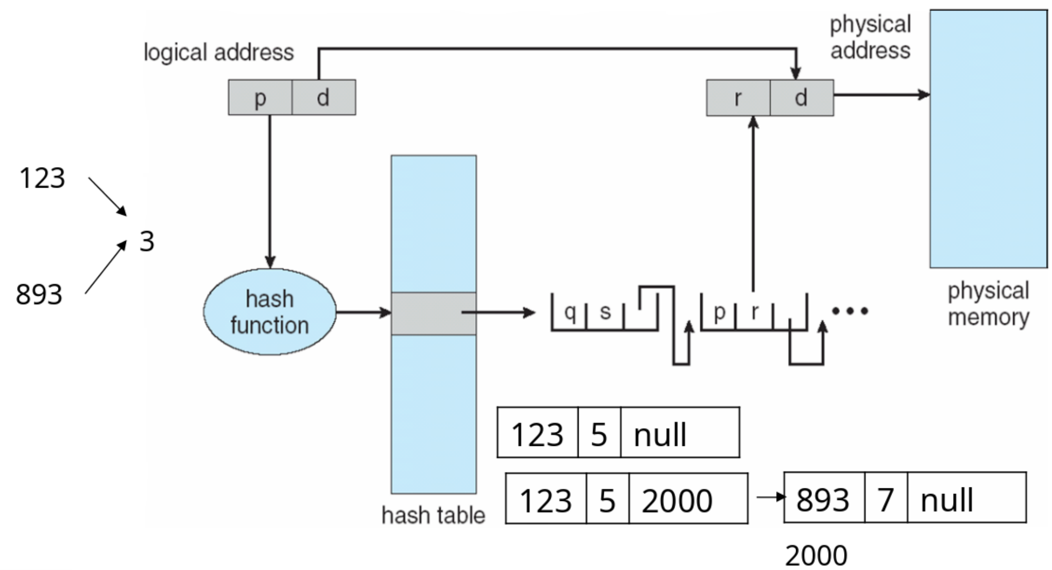
- **Hierarchical page tables** divide addresses into three sections:
  1. **The directory (Dir)** — First 10 bits.
  2. **The Table** — Next 10 bits.
  3. **The Offset** — Last 12 bits.
- The directory refers to a table that references other page tables. The table section of the address refers to the index of the referenced page table that we are interested in, the offset is the number of bytes from the start of the of the page.



- **Hierarchical page tables** allow us to store page tables in memory that is **not contiguous**. As page tables can get very large (32-bit systems need a 4MB page table per process).
- You can continue to add **more levels** to the hierachy for systems with more memory (eg 64-bit).

Hash Page Tables

- An efficient way to **map addresses** is with **hashed page tables**.
- You use a hashtable with a linkedlist to map values to adresses.



Inverted Page Tables

- A **modern** way to implement page tables is with **inverted page tables**.
- An **Inverted page table** is a page table that is shared with **all processes**, with addresses composed of process id's, block number, and offset.
- This **solves the problem** of **wasting a lot of memory** creating **page tables** for each process.