**Chapter 08: Main Memory**

Table of Contents

[8.1 Introduction 2](#_Toc83032673)

[Basic Hardware 2](#_Toc83032674)

[Address Binding 4](#_Toc83032675)

[Logical and Physical Address Spaces 5](#_Toc83032676)

[Dynamic Loading 5](#_Toc83032677)

[Dynamic Linking and Shared Libraries 5](#_Toc83032678)

[8.2 Swapping 7](#_Toc83032679)

[Standard Swapping 7](#_Toc83032680)

[8.3 Contiguous Memory Allocation 9](#_Toc83032681)

[Memory Allocation 9](#_Toc83032682)

[Fragmentation 10](#_Toc83032683)

[8.4 Segmentation 11](#_Toc83032684)

[Segmentation Hardware 11](#_Toc83032685)

[8.5 Paging 13](#_Toc83032686)

[Internal Fragmentation 14](#_Toc83032687)

[Translation Lookaside Buffer 15](#_Toc83032688)

## 8.1 Introduction

We know that **main memory** is a large **array of bytes**, with each element of the array having its **own address**. The value of the **program counter** stores one of these addresses, and based on that address, **instructions** are fetched. This involves some LOAD and STORE instructions.

### Basic Hardware

**Main memory** or **RAM** and some built-in **registers** are the only memory types that the processor can **directly access**. Code that we write in higher level languages are converted to **machine code**, which uses **machine instructions**. These instructions take **memory addresses** as arguments, but never disk addresses. This means that for the CPU to be able to operate on some data, it **must be in memory**.

**Register contents** can typically be accessed within a **single clock cycle**. This allows the CPU to decode instructions and perform simple operations on the register contents at a rate of 1 or more per clock tick. However, **RAM** is not so fast. The **memory bus** gets involved and **multiple CPU cycles** might be used to access RAM. This is unacceptable due to how frequently memory is accessed.

To deal with this, **cache** is built into the CPU. Modern CPUs have at least **three levels** of cache. Instructions that come after the last loaded one are stored on the cache so that they can be loaded directly from there.

Memory is typically **divided** between different processes and even different users. The OS does not intervene between the CPU and its memory accesses, so the hardware must be responsible for **protecting** one section of data from others. One possible way to do this is with the help of **base registers** and **limit registers**.



The **base register** holds the **memory address** of the **starting point** of the address space for a particular process. The **limit register** holds the **length** of the address space. Thus, the last legal address is the address space is found by adding the value of the limit register to the value of the base registers (and subtracting 1).

When the CPU generates an address in the **user mode**, we check if the address is within the legal address space. If it is not, a **trap** is generated, which is a software generated interrupt.

The base and limit registers can only be loaded using a special **privileged instruction**, which of course requires execution in the **kernel mode**.

### Address Binding

When we code in a higher-level language, we typically use **symbolic addresses** for variables. We do not specify a particular memory location in which to store a value.

The compiler binds the symbolic addresses to **relocatable addresses**, i.e. a particular variable may be 14 bytes from the start of the module. However, this is still not a memory location. We are essentially using a **relative address** at this point. The base address is generated from hardware and we will be looking into how that is done later.

The actual memory location, the **absolute address**, is bound by the linker during **load time**.

Physical addresses can be generated at one of three times:

1. **Compile Time** - If we know where the source code will reside in memory, we can directly use the compiler to generate the absolute addresses. If the starting location changes, the code will need to be recompiled.
2. **Load Time** – If it is not possible to know the location of the source code in memory beforehand, then the compiler generates relocatable code, as described above. The physical addresses are bound when the code is being loaded into memory. If the starting address changes, we just reload the code.
3. **Execution Time** – Programs can shift from one memory segment to another, for example if we move from the CPU to the I/O queue, then to the ready queue and back to the CPU. In these cases, the addresses will change, which means we cannot bind physical addresses even during load time. They must be bound during runtime, using special hardware.

### Logical and Physical Address Spaces

An address generated by the CPU is a **Logical Address**, whereas an address in memory is a **Physical Address**. For Compile Time and Load Time binding, the Logical and Physical Addresses are the same, but for Execution Time binding, they are different. In this case, the Logical Address is also called the **Virtual Address**.

Logical Addresses are **mapped** to Physical ones by the **Memory Management Unit**. One possible way to perform this mapping is with the help of a **base register**, called the **relocation register**. Essentially, the Logical Addresses are relative to the address in the base register.

### Dynamic Loading

The **limitation** to all of the above is that programs cannot be larger than the **physical memory space**. To solve this, we use **dynamic loading**. Essentially, a routine is not loaded until it is called. For example, one class might refer to a procedure from another class, but that procedure is not loaded into memory until that line of code is reached during execution.

### Dynamic Linking and Shared Libraries

**Dynamically linked libraries** are **pre-compiled libraries** that are linked to the user programs when the programs are run. This is different from **static libraries**, which we include as **header files**. This is slightly different from dynamic loading, in that the actual **linking** takes place during runtime. This results in the libraries not having to be included with the source code, which allows several different processes to use the dynamically linked libraries as **shared libraries**.

Instead of including the libraries itself, the source code includes a **stub**. This stub is a small piece of code that can **locate** the library in memory or **load** it into memory if it is not already present. The stub **replaces itself** with the **address of the routine** and executes it. This same loaded routine can then be used by other processes, which removes the cost of dynamic linking.

If the library is **updated**, all programs that reference the library will automatically use the updated version. Without dynamic linking, the programs would need to be **relinked**. **Version information** is also included in the programs and the libraries so that programs use the correct version of the library if multiple are available.

## 8.2 Swapping

### Standard Swapping

We know that the CPU executes processes by taking them from the **ready queue** and when a process requires some I/O device, it is moved to the **I/O Queue**. A process that is being moved out of CPU execution and into the I/O queue is said to be **swapped out** and a process that is being moved into CPU execution is said to be **swapped in**.

Suppose we have two processes in the **ready queue**, and . is **swapped in**, and after some time, it is **swapped out** because it needs an I/O device and is **swapped in**. At the moment, say some other process, , is already in the **I/O queue**, so that must wait.

When swapping a process, we need to ensure that the **process is idle**. Pending I/O operations are particularly concerning. For the above scenario, where is waiting for an I/O device, if we try to move it, the I/O device may end up trying to access memory locations that no longer belong to .

There are two ways in which we can handle this:

1. Never swap a process with **pending I/O operations**
2. Perform I/O operations in the **OS buffer**

For the second case, when the proper process is using the CPU again, it will **communicate with the OS buffer** to retrieve the output of the I/O operations. There is a **double buffer** here, extra memory being used, which is obviously an overhead.

This sort of swapping is called **standard swapping**. This is not directly used in modern computers. Instead, modifications are done to the process and the modified versions are used.

## 8.3 Contiguous Memory Allocation

We know that processes can belong to either the OS or to user programs. The **main memory** needs to handle data from both types of processes in the most **efficient** manner possible. In this section, we will discuss an early method of doing this, called **contiguous memory allocation**.

Consider that we have a bunch of **device drivers**, some of which we use frequently and some of which we use very rarely. However, all of them are taking up **memory space** in the part of the main memory which the OS occupies. Say we take the **rarely used** device drivers, place them on **disk** and give the freed memory to **user processes**. This will increase efficiency. We can reload the rarely used device drivers when we need them.

Essentially, we have made the part of the memory space occupied by the OS **variable**. The code we are keeping sometimes in memory and sometimes on disk is called **transient code**. Contiguous memory allocation allows us to do this.

### Memory Allocation

One of the simplest methods of memory allocation is to use **partitions**. Instead of using the entire memory as a whole, we divide it into parts. The numbers of partitions we can make directly corresponds to the number of **processes** we can have running at the same time. Spaces in memory that are unused are called **holes**.

Suppose we have three processes, , and , which require , and bytes of memory respectively. However, we only have bytes available. If we allocate memory to and , will have to wait.

Now, we might not have the freedom to allocate memory to processes side by side. We might have to deal with **holes**. There are several schemes we can use to decide which location of memory to place a process in based on this.

1. **First Fit** – Find the first hole and place the process there. Searching can begin at the start or where the last search ended.
2. **Best Fit** – Find the smallest hole that is big enough to hold the process.
3. **Worst Fit** – Find the largest hole available. This is useful for processes that might dynamically increase in size.

### Fragmentation

The **first-fit** and **best-fit** strategies both suffer from **external fragmentation**. As processes are loaded and removed repeatedly, the holes get broken into **small pieces**. Eventually, we will have a situation where there is enough space to hold a process, but the holes are **not contiguous**, which prevents us from using them. The small holes are called **external fragments**.

The **worst-fit** strategy on the other hand, suffers from **internal fragmentation**. This is when a process is assigned to a hole that is **too large**, which leaves unused memory internal to the partition.

When we **defragment** memory, things are moved around to bring the holes together, thus improving memory utilization. This is also called **compaction**.

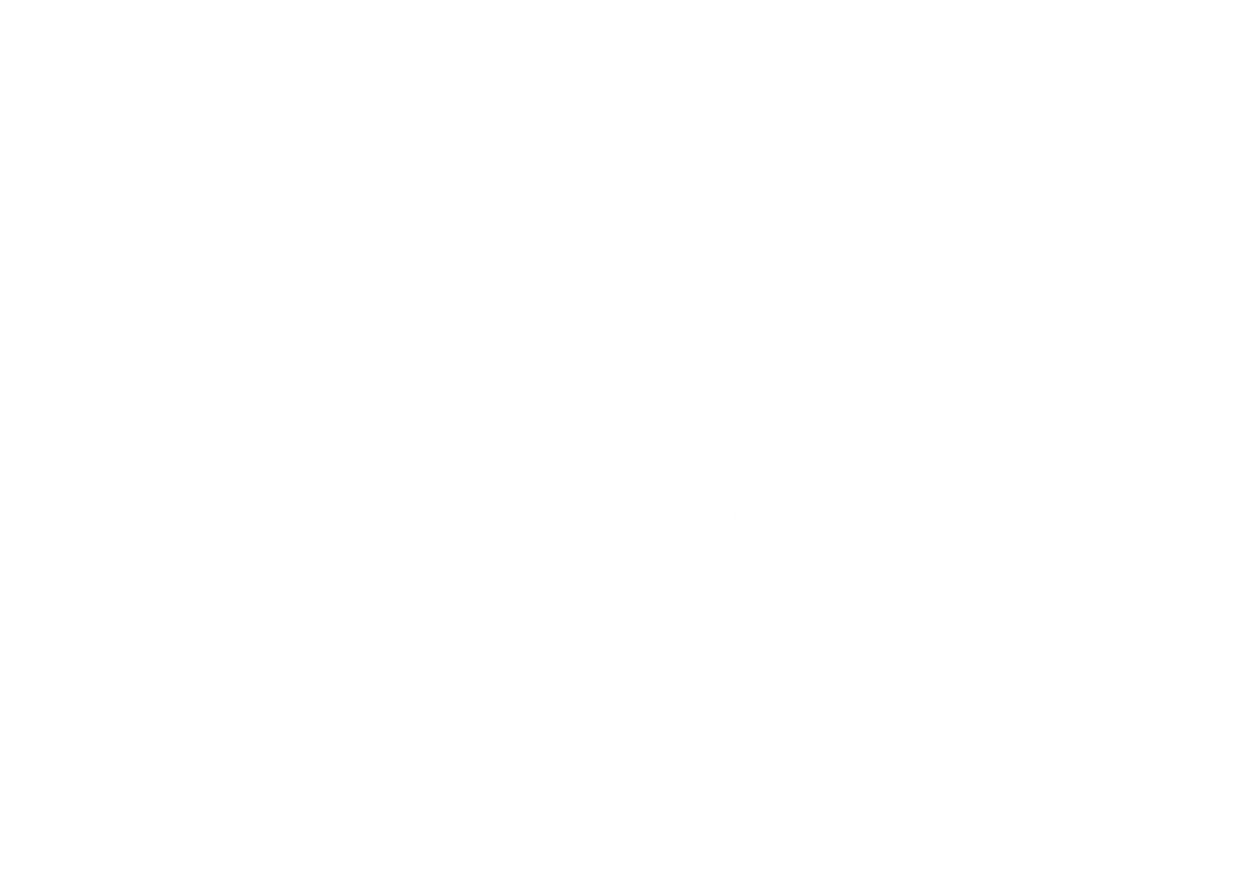
## 8.4 Segmentation

We already know that **logical address** do not correspond to **physical addresses**. If we create an array, the actual storage of values may not be one after another like we imagine. If we have a bunch of functions in a program, each function may be stored in a different memory location.

The **logical address space** generated by the CPU is a collection of **segments**, each of which has a number and length. The different logical addresses specify the segment number and the **offset** within the segment, i.e. <segment-number, offset>.

### Segmentation Hardware

To convert the segment number and offset of a logical address to a physical address, we need **segmentation hardware**.



The **segment number** is used to locate a segment within the **segment table**. This table consists of records that contain the **limit** and **base** of different segments. Once this data is located, we check whether the **offset** provided is within the limit or not. If it is not, then we throw an **error**. If it is, we add it to the **base address**, which gives us the physical address.

## 8.5 Paging

**Paging** gives us the same benefits as segmentation, but without the issues presented by **external fragmentation**. Here, **logical memory** is divided into **pages**.

In paging, we divide both the logical memory and the physical memory into **blocks** of the same size. For the logical memory, the blocks are called **pages**. For physical memory, the blocks are called **frames**.

A specific logical address consists of a **page number** () and an **offset** (). The page number is used to retrieve the **base address** of the page from a **page table**.

A page number cannot be defined arbitrarily. There are some rules.

Say the **page size** we are dealing with is . Having the size as a **power of**  has lots of benefits. The total **logical memory** that we have can be represented as , meaning there are bits.

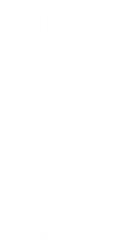
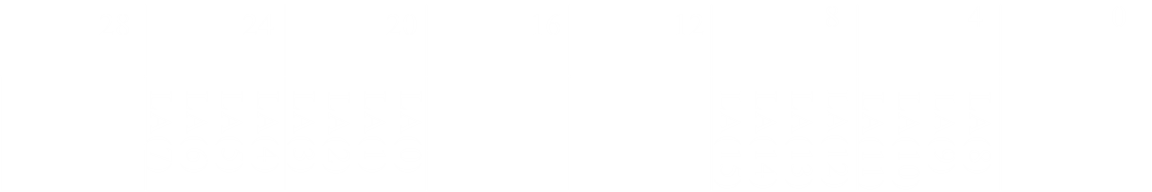
The **left-most** bits from the logical address create the **page number**, while the **right-most** bits create the **page offset**. This should be clearer with an example.

Consider that and , so that we have a **page size** of **4 bytes** and a **logical address space** of **16 bytes**. Say the **physical memory** is of **32 bytes**, so that it can hold a total of **8 pages**.

Since , and we are using the left-most two bits for **page numbers**, the only valid page numbers are , , and . From this, we can use the different **logical addresses** to calculate the page number and the offset.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| LA(0) | 00 | 00 |
| LA(3) | 00 | 11 |
| LA(4) | 01 | 00 |
| LA(13) | 11 | 01 |

In the **page table**, the indices for each page number has a **frame number**. We use this to locate the frame in the physical memory.

Finally, we can calculate the **physical address** as . For example, LA(3) would be stored in memory location .

### Internal Fragmentation

Paging does not suffer from external fragmentation, but it does suffer from **internal fragmentation**. If we have pages of bytes for example, and a process of bytes, the process will need 35 pages and would leave bytes left. In the worst case, we could have bytes left, with just 1 byte being used by a process.

To avoid this, the **page sizes** need to be **optimally defined**, not so small that there is too much overhead and not so large that there is a lot of internal fragmentation.

### Translation Lookaside Buffer

The **page table** we are using is itself stored on the physical memory. Thus, we need to access the physical memory once to get the frame number and once more to get the data. **Two accesses** will cost us twice the time. If a single access takes , we will need for a single access.

To avoid this, the **translation lookaside buffer** (TLB) is used. This stores all the recently used entries from the page table on the **main memory**. The TLB is more commonly called the **L1 or L2 cache**.

If we cannot find a page we need in the TLB, it still needs to be retrieved from the page table. This is called a **TLB miss**. The ratio of the number of times we find a page on the TLB to the number of times we have a TLB miss is called the **hit ratio**.

This directly corresponds to the **average access time** we have. If we have a 80% hit ratio, we will need to access our data 80% of the time and to access our data 20% of the time. Thus, the average access time is . The higher the hit ratio, the lower the average access time.