[redacted] (Virtual Memory Management 1)

[redacted]

1. Background

[redacted]

Let us first understand the working of a hardware component that plays a central role in overall memory management. This unit, part of the CPU, is the memory management unit (MMU) whose primary role is to correctly translate virtual addresses into physical addresses.

To begin the discussion of memory management unit, first we reminded ourselves of the basic property of a computer that for a computer to work on a program the code and data portions of the process must be brought from the hard disk into the main memory and placed within a process for it to be run. CPU only has direct access to the main memory and registers.

Registers are the fastest memory in computers that can be accessed in one CPU clock (or less), but DRAM (as a kind of main memory) can take many cycles. Level 1 and Level 2 caches sit between CPU registers and main memory. These memory resources are shared between processes. Protection of reserved memory for each process is required to prevent access by other processes and ensure correct operation.

The way we virtualized the register file was based on the ability to save the contents of registers for a process somewhere in memory when it was being context switched out and then relying on virtual memory management of the OS and the hardware MMU to ensure that these contents are kept safe (i.e., untouched by other processes). That is, virtualization of CPU's registers itself depends on correct memory isolation for which the OS memory manager + the MMU are responsible.

Virtualizing these registers within the CPU rely on protecting the memory contents of one process from another, next we will learn how memory content of one process is protected from another. Our highest level goal is to take the address spaces of multiple processes and feed them in a given amount of DRAM or main memory space such that they are protected from each other.

Virtual memory management is essential to be able to have multiple threads and feed them by limited number of memory resources when their address spaces are protected from each other.

2. Logical Address vs. Physical Address

Next we focused on the distinction between virtual or logical addresses and physical addresses; the addresses we were talking about so far were virtual or logical addresses. Next we should concern maintaining the relationship between logical versus physical addresses.

The notion of the logical address space that is bound to a separate **physical address space** is central to proper memory management.

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit

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

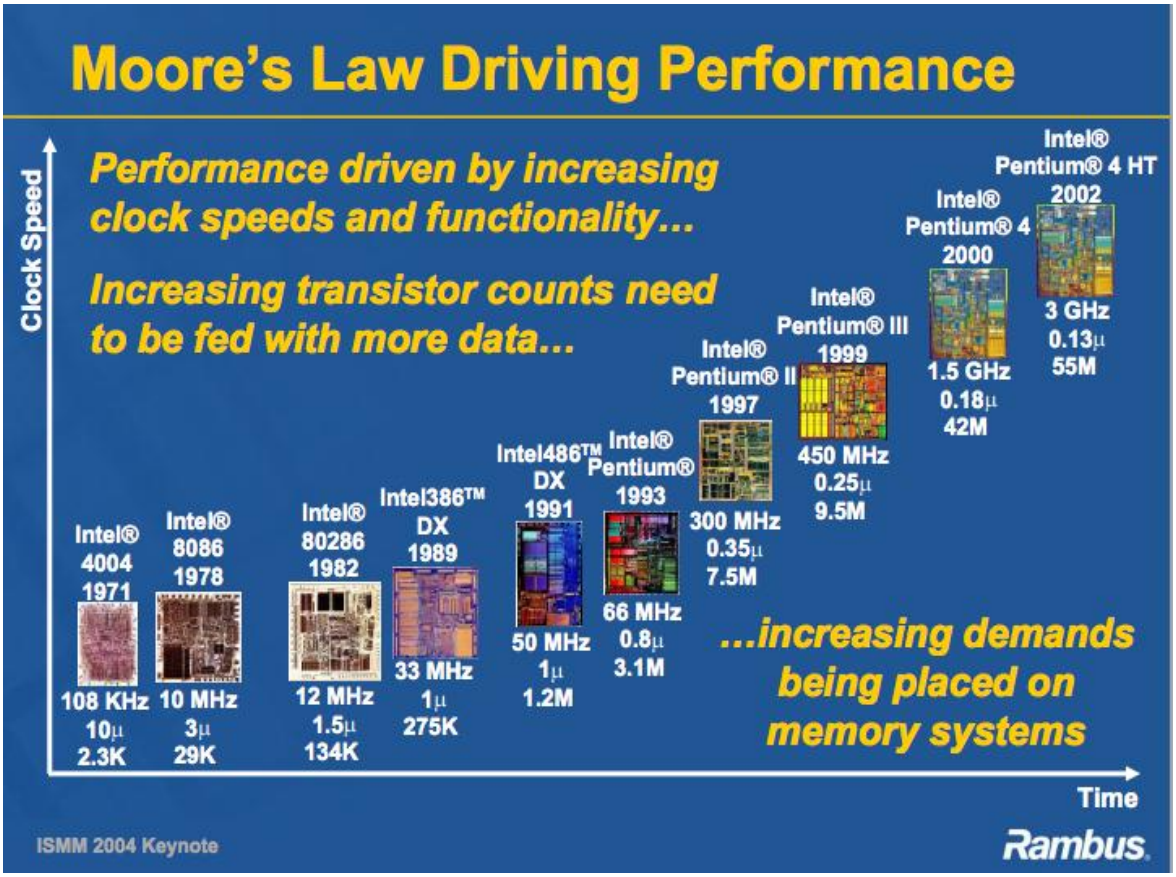
We want to know the relation between physical and virtual address. What is the task of the OS in memory management?

3. Trends

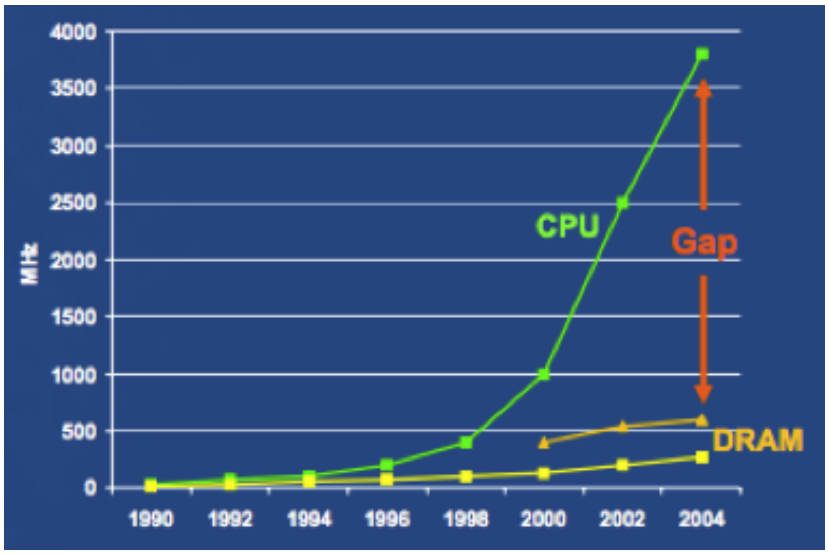
Before beginning our discussion of memory management we started some background of the so-called "memory wall" and Moore's law.

Understanding these trends will help us to understand what operating system's goal is in managing memory and what kind of support it must have from hardware? **Moore's law** says that every two years the number of transistors able to fit on CPU doubles, as does the speed of the computers.

This chart depicts the implication of Moore's law on how fast CPU circuits have become over past few years. We have been able to increase clock speeds of the CPU at a rate that is incredible. But the performance that the program sees depends not just on how fast the CPU runs the instructions, but also on how fast other resources are cooperating with the CPU.



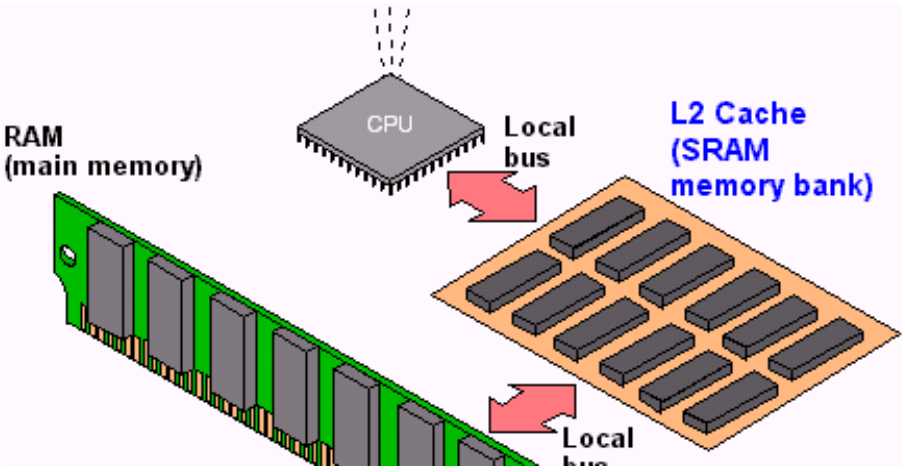
Interesting to know but not related to our discussion here (added by the scribes): Amdahl's law says that the minimum time required to execute programs in parallel is at least the time required to execute the sequential part of the program that cannot be parallelized.



Memory Wall:

The large gap between DRAM and CPU performance (both bytes/sec and latency) is called as the "memory wall" and is a significant determinant of the effective speed of a computer. A useful example to think about to appreciate the memory wall and its implications: Assume that 30% of a program code contains Load/Store instructions, and the other instructions do not access memory. What is the big difference between how this program runs today versus how it ran 30 years ago?

So if we were to execute such a program today, 70% of instructions will execute much faster than in the past (for example 70% percent of instructions that used to run in 10 seconds will be run in picoseconds today), but the 30% related to Load/Store would not have as much of a performance increase. The load and store instructions have become faster but they have become faster by the factor of two or three. So the overall program runs much faster because of the instructions that are only concerned with the CPU.



When we look at new computers we will realize that despite all the fancy CPU managements, despite the fast context switching that we might have created, because the memory unit is so slow, the CPU is possibly spending most of his time for the memory unit to respond. The CPU has become very fast, but when it is given a load and store instruction it has to wait for many cycles. This phenomenon is called memory wall. There was a huge gap between memory and CPU in the past, but this gap is significantly larger today.

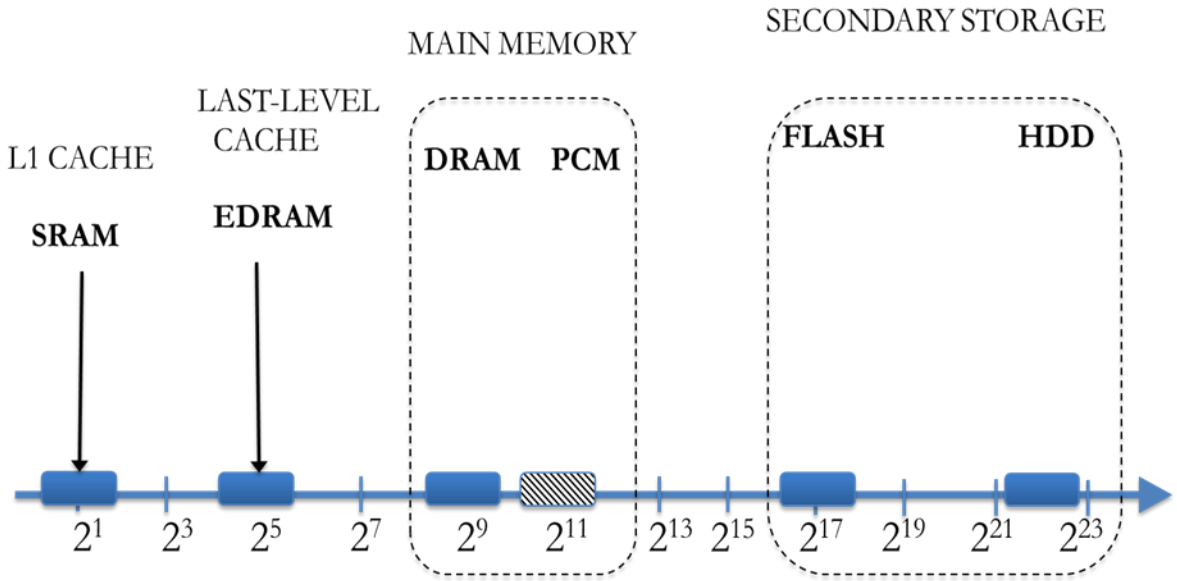
Latency is the amount of time it takes a program to respond. There is a huge gap between latency bandwidth of CPU and memory. The improvement in bandwidth is higher than the improvement in latency. But how is this gap filled?

1. Use faster memory technologies. DDR (Double Data Rate, also known as DDR 1, 2, 3 or 4) represents the technique where in the design of memory systems data is transferred on both falling and raising edges of the clock signal. DDR3 (64MB/s) is the most recent technology regarding main memory access.

2. Use hardware caches by assuming locality (temporal/spatial) exist, the basic idea is that in most programs certain parts of their address spaces are accessed more frequently, this is known as locality. Assuming locality exist it becomes possible to identify the most popular content and place it in fastest circuits closest to CPU. So the probabilistic latency is improved by reducing the probability of miss rates.

3 Rewrite the program to execute Load/Store once in the pipeline without stalls.

4. Access Latencies



Access latencies of different technologies are shown by the number of CPU cycles as 2^n. It brings out huge ranges of access latencies and the gaps between them. The registers are the fastest hardware holding memory and are not shown in the above picture. The next fastest is the L1 Cache made of SRAM. Main memory technology uses DRAM, which is considerably slower than SRAM (DRAM has a

latency of about 100 cycles). Another kind of memory technology known as Embedded DRAM whose access time (30-40 cycles) is between SRAM and DRAM. These memories reside in the motherboard.

SRAM and DRAM are volatile, but HDD and flash technologies remain persistent. This means they do not lose their information when disconnected from a power source. HDD technology is very slow despite some new flash technologies like SSD. Flash storage is faster than HDD but also more expensive. Any technology that fills the gap between them could be useful in the hierarchical architecture of memories in computer.

PCM (Phase-change Memory) is a main memory technology that is emerging and different from the prior types. Phase-change memory relies on a certain kind of material which can exist in two phases based on the temperature that is applied to it. It consumes less power than DRAM and is persistent like magnetic memory (HDD). Hybrid combinations of DRAM and PCM can be imagined when we want to save to PCM the pointer of a memory location residing in DRAM, as opposed to saving both of them in DRAM. Imagine a system that where we are using both DRAM and PCM to construct the main memory. Let's first start with our traditional computer that only has DRAM, and imagine a data structure with one element which points to another, both of them in DRAM, both of them is lost. But if we have a hybrid system, one of them will be gone and one will stay. We will possibly come back to this issue again. There are some tradeoffs among memory technologies like cost, speed, persistence, and size:

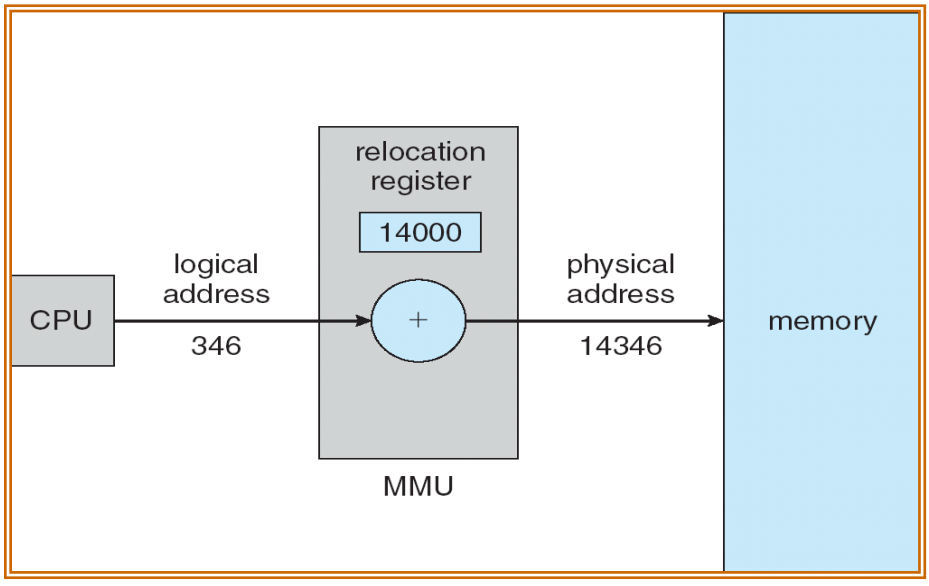
With time these technologies have become cheaper.

5. Memory Management Unit (MMU)

This is the primitive unit of memory management that is necessary for OS efficiency. **Memory management unit (MMU)** is within the CPU, and is an example of primitive job. Memory management unit is a piece of hardware in CPU whose job is to map virtual addresses to physical addresses. This translation must be done in a way so that can protect the contents of one program in a process from the content of another.

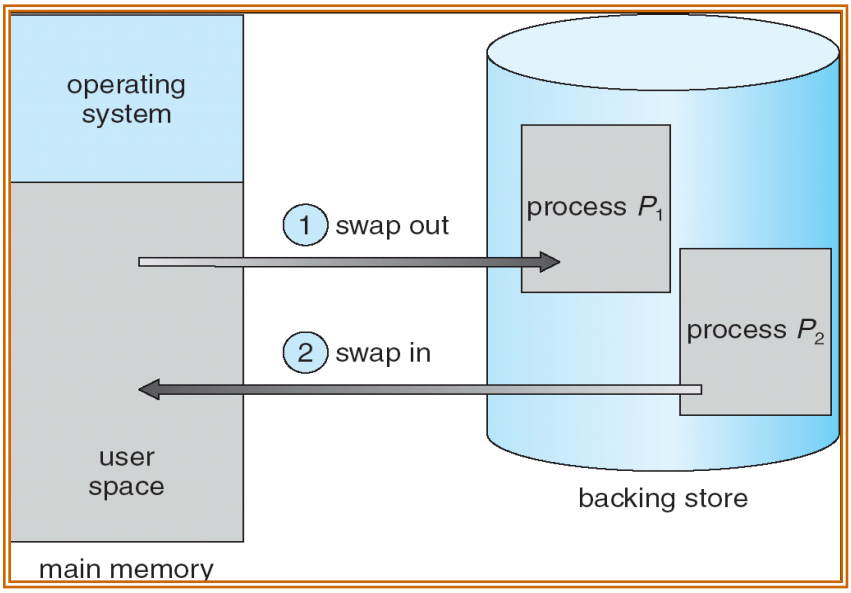
- Hardware device that translates or maps virtual (logical) to physical address by adding the offset to the logical address number.
- In the MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

The problem the MMU deals with is fitting the content of address spaces of all programs in the memory. MMU can implement **dynamic relocation using a relocation register**. The MMU increases all addresses of a program by some integer (14000 for the example shown in the figure).

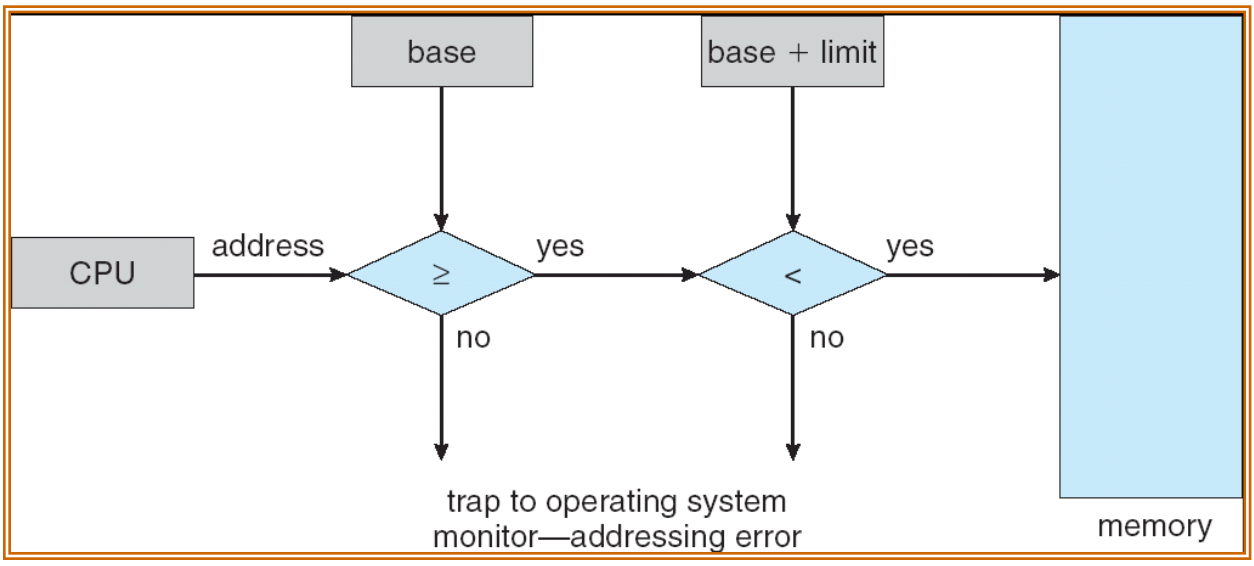


If we knew the limit on the address space of each process, by choosing the value of relocation space to be multiples of that and giving separate relocation value to each process we would have protection, but there are many downsides of this: each process has to be aware of the maximum for each process every time, even if it is not using it no one will use it. One way to eliminate this problem is to have two registers for each, base and limit registers. With the use of two hardware-registers called base and limit that the MMU maintains for each segment. The base register points to a segment's start address (usually 0) and the limit register points to its last address. A process can never access an address that is larger than its limit register. That fixes part of problem but there is still problem. When the process is context switched out the base and limit registers are stored in process control block.

A device called a **backing store** (or swap device) can be used to accommodate address space contents that do not fit within main memory. We will discuss this in detail soon. For now, let's assume that everything can fit within DRAM and focus on protecting contents of a process from being modified by other processes.

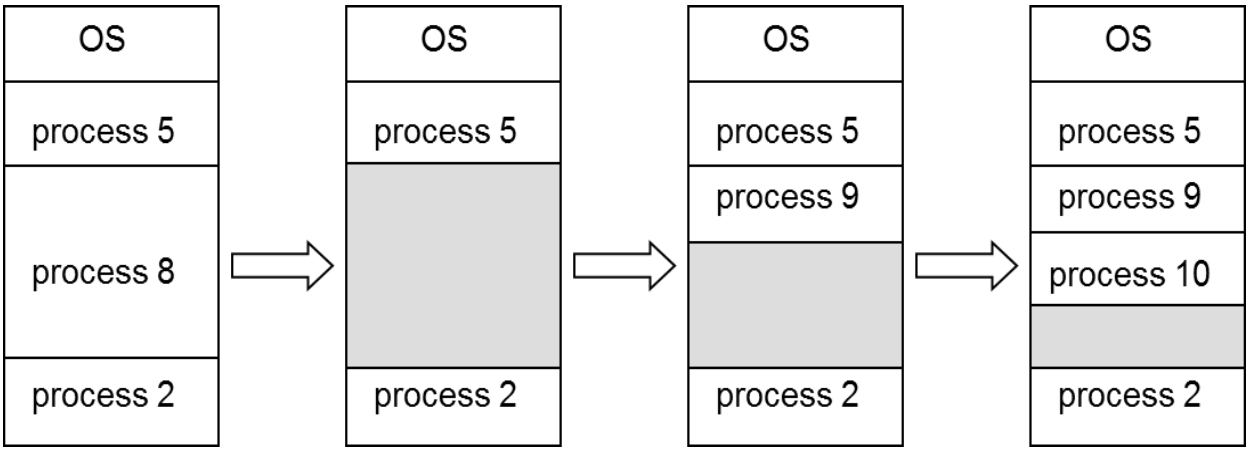


In **contiguous allocation** the MMU usually partitions main memory into two parts. The resident OS is often held in low memory with an interrupt vector and the user processes is held in higher addresses. Relocation registers are used to protect user processes from each other, as well as prevent changing OS code and data. A base register contains the value of the smallest physical address. A limit register contains the range of logical addresses, and each logical address must be less than the limit register. MMU maps logical address *dynamically.* MMU can use "base" and "base + limit" registers as shown in following figure.



Register contents are stored in the PCB (process control block). When the OS context switches to the user process, the “base” and “base + limit” register contents are loaded from the corresponding PCB in memory.

Hardware addresses are protected with base and limit registers. Multiple-partition allocation is done by the MMU. Holes (block of available memory) of various sizes are scattered throughout memory. When a process arrives, it is allocated memory from a hole large enough to accommodate it. The OS maintains information about allocated partitions and free partitions (holes).



Now if process 9 is exited, and process 11 arrives with an address space a little greater more than process 9, what happens to MMU?

Operating Systems can use several approaches in solving this dynamic allocation problem. When a request is received for a given amount of memory, the process will be given memory from a hold according to first-fit, best-fit, or worst-fit techniques.

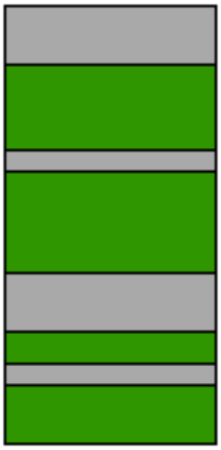
The **first-fit** method involves searching the available memory for a hole large enough to accommodate the request. The first hole that can satisfy the request is chosen to allocate memory from. This method is the most efficient in terms of speed.

The **best-fit** method is a process in which the operating system must search for all available holes that are large enough to fulfill the process’ request. After identifying all such holes, the operating system allocates memory from the smallest hole of adequate size. Best-fit allocation is most efficient in terms of storage utilization because it results in the smallest possible leftover hole.

Worst-fit consists of searching all the available holes as in the best-fit method. However, in this scheme the operating system allocates from the hole that is the largest. This results in the largest possible leftover hole, which has inferior efficiency both in terms of speed and storage utilization.

6. Fragmentation

One inherent problem associated with memory allocation is that of fragmentation. This appears in two different circumstances called internal and external fragmentation. Consider the following portion of memory:



In this picture, imagine that the green blocks are allocated portions of memory and the grey ones are not. This illustrates the problem of **external fragmentation**. As processes obtain and release memory, the portions allocated end up not remaining contiguous. It is easy to see that this can result in holes of memory that are not well sized for further allocation, resulting in wastage.

Using dynamic compaction can reduce external fragmentation. This process occurs at execution time and relocates blocks of memory into positions such that they are contiguous. This results in allocated memory residing in one large block and eliminates the problems of external fragmentation. However, compaction causes a problem of its own with I/O. This leads to latching jobs in memory when they are dealing with input or output. This will hold onto the data and means the operating system needs to control the buffers for I/O.

7. Paging

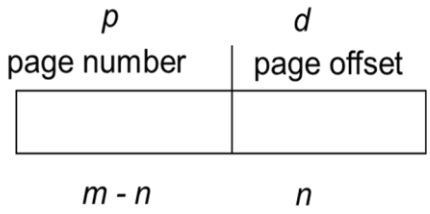
Paging is another approach to preventing external fragmentation. It revolves around the idea that the virtual addresses in a process do not need to map onto a single block in physical memory. The physical memory is divided into fixed-size blocks called **pages** that range between 512 and 8,192 bytes (powers of 2). In this scheme, the operating system keeps track of available pages rather than searching memory for open holes. A process simply requests a certain number of pages from the operating system and these are loaded for the process to run.

A new type of fragmentation arises in the use of pages. Imagine a process that needs slightly more than a power of two amount of memory. The operating system will provide enough pages for the process, meaning the last page provided is only partially utilized. This inefficiency can be avoided by decreasing the size of pages. If pages are smaller, there is less unused space risked.

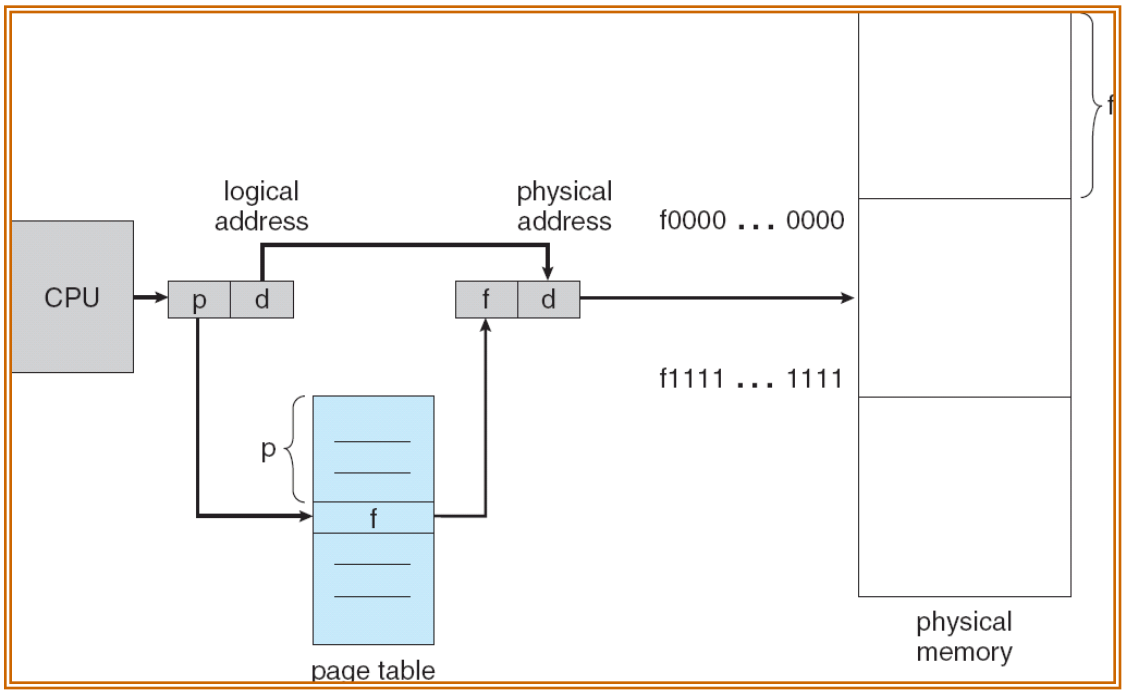
In order to keep track of which pages are allocated and to which process, the operating system holds a structure called the **page table**. The function of the page table is to translate logical addresses specific to a process into a physical address in the memory.

Recall that using smaller page sizes can reduce internal fragmentation. While this has benefits for storage utilization, it can drastically increase the size of the page table. Page tables consume memory, making a tradeoff emerge in terms of storage when decreasing the size of a page. More entries in the table also mean a loss of efficiency because the operating system will have to search a larger set for the translation. When determining the size to make pages these factors must be taken into account.

The process of translating an address begins when the CPU generates the virtual address. This address has two parts: the page number and page offset. The **page number** serves as an index to the page table which can access the physical memory addresses. The **page offset** defines how many addresses into a page the current address is. For example, an address that is 4 away from the base of the page in virtual memory is also 4 addresses away from the base of the page in physical memory. The page number and page offset can be visualized as follows.



The logical address space will have a range of 2^m addresses. The page size is 2^n, meaning the page offset field must be able to span the entire page size. The remaining bits identify the page number.



The following diagram illustrates how the hardware in address translation works.

This shows the CPU generating a virtual address with a page number and page offset. The page number is sent to the page table, which maps it onto a new value and outputs the physical address. The offset is appended to the new physical address and then the memory can be directly accessed.