**Main Memory**

Memory consists of a large array of bytes, each with its own address.

**Basic Hardware – (registers, stalls, cache, protection)**

The CPU can only directly access main memory and registers built into each processing core for general-purpose storage. While machine instructions can take memory addresses as arguments, there are none that take disk addresses directly. This means that any instructions being executed and any data required by those instructions must reside in either main memory or the registers. If the required data is not in main memory, it must be moved there before the CPU can operate on it.

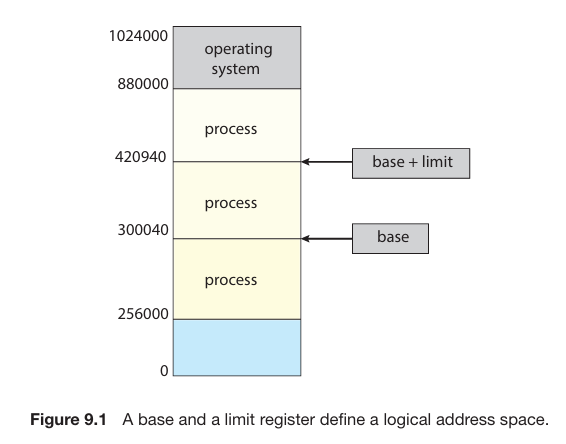
Registers built into each CPU core are typically accessible within one cycle of the CPU clock, allowing for fast operations. Some CPU cores can even decode instructions and perform simple operations on register contents at a rate of one or more operations per clock tick. However, accessing main memory is slower as it requires transactions on the memory bus, which may take many cycles of the CPU clock to complete. When the CPU lacks the required data in memory to complete an instruction, it often needs to stall, which is problematic due to the frequency of memory accesses.

To mitigate this issue, fast memory, known as cache, is added between the CPU and main memory, often integrated onto the CPU chip for quick access. This cache automatically speeds up memory access without the need for operating system control. This management of cache built into the CPU hardware helps improve overall system performance.

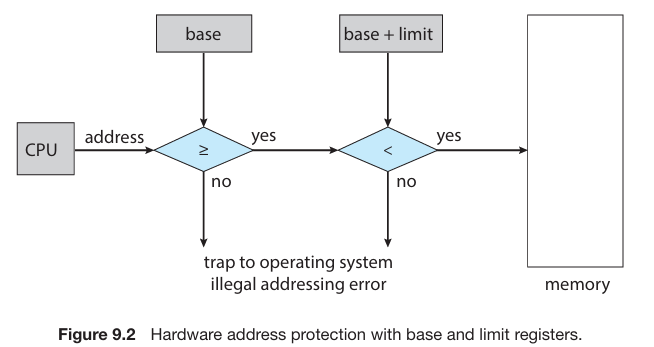
In addition to concerns about memory access speed, ensuring correct operation of a system involves protecting the operating system from user processes and preventing interference between user processes. Hardware is responsible for providing this protection, as the operating system typically does not intervene directly in CPU-memory interactions due to performance considerations

One way hardware achieves this is through the use of base and limit registers to define separate memory spaces for each process. This separation is crucial for preventing processes from accessing each other's memory and allows for the concurrent execution of multiple processes.

These registers establish the range of legal addresses that a process can access. The ***base*** register holds the smallest legal physical memory address, while the ***limit*** register specifies the size of the range. For instance, if the base register holds 300040 and the limit register is set to 120900, the program can legally access all addresses from 300040 through 420939 (inclusive). This mechanism ensures that processes can only access memory within their designated range, enhancing system security and stability.



Memory space protection is achieved by having the CPU hardware compare every address generated in user mode with the base and limit registers. Any attempt by a program running in user mode to access memory outside of its designated range, including operating system memory or other users' memory, triggers a trap to the operating system. The operating system treats such attempts as fatal errors, preventing user programs from inadvertently or intentionally modifying the code or data structures of the operating system or other users.



The base and limit registers can only be loaded by the operating system using a special privileged instruction. Privileged instructions can only be executed in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This setup allows the operating system to modify the values of these registers as needed but prevents user programs from altering their contents, ensuring the integrity of memory protection mechanisms.

In kernel mode, the operating system has unrestricted access to both operating-system memory and users' memory. This privilege enables the operating system to perform various essential tasks, including:

1. Loading users' programs into their designated memory spaces.

2. Dumping out programs in case of errors or crashes.

3. Accessing and modifying parameters of system calls, which are requests for services made by user programs.

4. Performing input/output (I/O) operations to and from user memory.

5. Providing a multitude of other services crucial for system functionality and user interactions.

For example, in a multiprocessing system, the operating system is responsible for executing context switches. This involves saving the state of one process from the CPU registers into main memory before loading the context of the next process from main memory back into the registers. This seamless transition between processes allows for efficient multitasking and utilization of system resources.

**Address Binding – (Compile time, load time, run time)**

In most computer systems, a program initially exists on a disk as a binary executable file. To run, the program must be loaded into memory and associated with a process, where it becomes eligible for execution on an available CPU. As the process executes, it accesses instructions and data from memory, and upon termination, its memory is released for use by other processes.

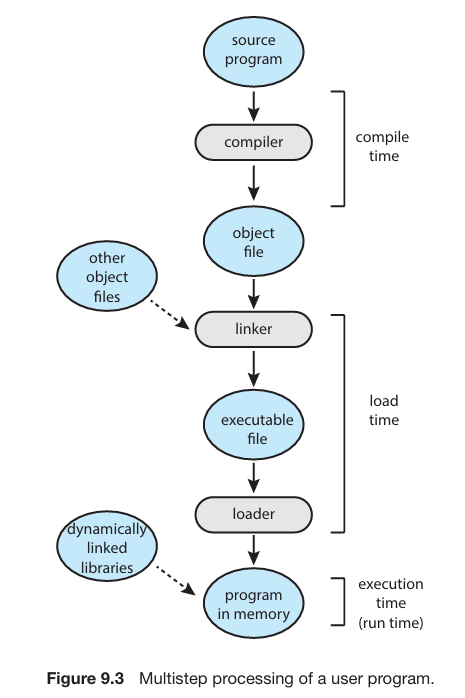
In many systems, a user process can reside in any part of physical memory, so its starting address need not necessarily be 00000. The operating system handles the placement of processes in physical memory.

Before execution, a user program typically undergoes several steps, some of which may be optional. Addresses are represented differently during these steps: symbolic addresses in the source program, which the compiler binds to relocatable addresses, and then the linker or loader binds these relocatable addresses to absolute addresses. Each binding is a mapping from one address space to another.

The binding of instructions and data to memory addresses can occur at different stages:

* **Compile time**: If the process's memory location is known at compile time, absolute code can be generated. Recompilation is necessary if the starting location changes later.
* **Load time**: If the process's memory location is unknown at compile time, the compiler generates relocatable code. The final binding occurs at load time, and reloading the code is sufficient if the starting address changes.
* **Execution time**: If the process can be moved during execution from one memory segment to another, binding must be delayed until runtime. This method requires special hardware, as discussed later.

Most operating systems employ the execution time method, where binding is deferred until runtime and facilitated by hardware support.



**Logical versus Physical address Space**

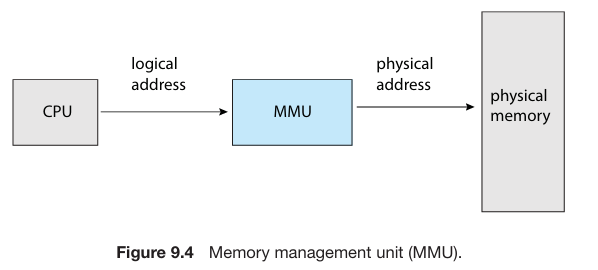
**Logical address** – Address generated by CPU is commonly referred to as *logical* address.

**Physical address** – Address seen by the memory unit is called *physical* address.

When addresses are bound at compile or load time, both logical and physical addresses are identical. However, in the execution-time address-binding scheme, there's a distinction between logical and physical addresses.

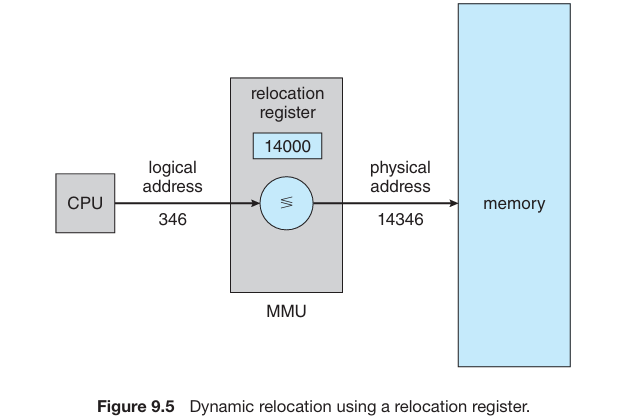
In summary, in the execution-time address-binding scheme, the logical address space and the physical address space differ. The set of all logical addresses generated by a program constitutes the logical address space, while the set of all physical addresses corresponding to these logical addresses constitutes the physical address space. Thus the logical and physical address spaces differ.

The mapping from virtual to physical addresses at runtime is facilitated by a hardware device known as the **Memory-Management Unit (MMU)**.



One straightforward MMU scheme, which serves as a generalization of the base register scheme described earlier, involves a relocation register. In this scheme, the relocation register, similar to the base register, holds a value that is added to every address generated by a user process before being sent to memory.

For instance, if the relocation register's value is 14000 and a user attempts to access location 0, the address is dynamically relocated to location 14000. Similarly, an attempt to access location 346 is mapped to physical location 14346 by adding the relocation register's value to the virtual address. This mechanism ensures that each address generated by a user process is correctly mapped to its corresponding physical address in memory.



In the execution-time binding scheme, the user program operates solely with logical addresses, never directly accessing physical addresses. For instance, the program can manipulate and compare a pointer to location 346 without considering its physical counterpart. However, when used as a memory address, such as in an indirect load or store operation, it undergoes relocation relative to the base register by memory mapping hardware. This hardware translates logical addresses into physical addresses.

We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R +0 to R +max for a base value R). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.

**Dynamic Loading**

**Dynamic loading** is a strategy used to optimize memory utilization in computer programs. Instead of loading the entire program and all its data into memory at once, dynamic loading loads routines only when they are needed. These routines are stored on disk in a relocatable load format. When a routine needs to be executed, it is loaded into memory on-demand, and the program's address tables are updated accordingly.

The main *advantage* of dynamic loading is its ability to conserve memory by loading only the portions of the program that are actively used. This is particularly beneficial for handling infrequently occurring cases, such as error routines, where large amounts of code may be needed but only a small portion is used during normal execution.

Dynamic loading does not require special support from the operating system; rather, it is the responsibility of the programmers to design their programs to take advantage of this method. However, operating systems can provide library routines to facilitate dynamic loading and help programmers implement it effectively.

**Dynamic Linking and sharing Libraries**

**Dynamic Link Libraries (DLLs)** are system libraries that are linked to user programs at runtime, unlike static linking where libraries are combined into the binary program image by the loader. Dynamic linking postpones linking until execution time and is commonly used with system libraries like the standard C language library. Without dynamic linking, each program would need its own copy of the language library, increasing executable size and possibly wasting memory.

DLLs offer several advantages:

1. **Memory efficiency**: They can be shared among multiple processes, reducing memory usage as only one instance of the DLL is loaded into memory.

2. **Flexibility**: DLLs can be updated with bug fixes or new versions, and all programs referencing the library will automatically use the new version without needing to be relinked.

3. **Version control**: Version information is included in both the program and the library, allowing different versions to coexist in memory. Programs decide which version to use based on their version information.

Dynamic linking and shared libraries usually require support from the operating system, especially in protected memory environments where the OS manages memory access between processes. The OS handles tasks such as locating and loading DLLs into memory, adjusting addresses that reference functions in the library, and ensuring that multiple processes can access the same memory addresses when necessary.

**Contiguous Memory Allocation**

Contiguous memory allocation is a method used to allocate memory for both the operating system and user processes. In this approach, memory is divided into two partitions: one for the operating system and the other for user processes. The operating system can be placed either in low or high memory addresses, with many modern operating systems like Linux and Windows choosing to place it in high memory.

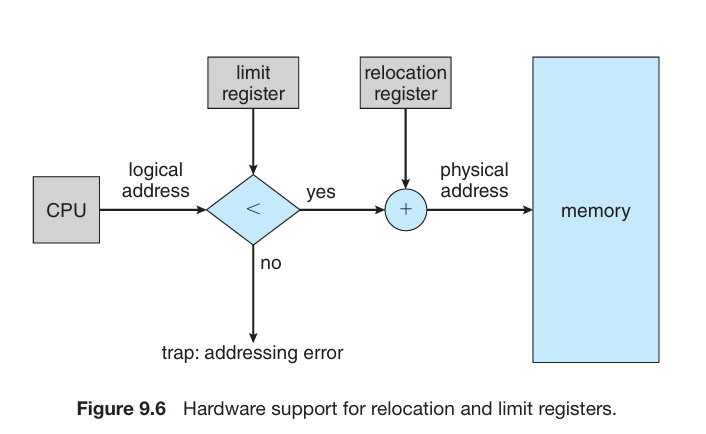
The goal is to efficiently allocate memory for multiple user processes. In contiguous memory allocation, each process is allocated a single contiguous section of memory next to the section containing the next process. This allows for efficient memory management and access.

**Memory Protection**

To prevent a process from accessing memory it doesn't own, a combination of two previously discussed concepts can be used: the relocation register and the limit register. The relocation register holds the smallest physical address, while the limit register defines the range of logical addresses accessible to the process.

When a process is selected for execution, the CPU dispatcher updates the relocation and limit registers as part of the context switch. The Memory Management Unit (MMU) dynamically maps logical addresses by adding the relocation value, ensuring that all addresses generated by the CPU fall within the specified range.

This scheme not only protects the operating system and other user programs from unauthorized access but also allows for dynamic resizing of the operating system. For instance, device drivers (OS keeps separate buffer and code for drivers) and other OS components can be loaded into memory only when needed, and removed when no longer required, providing flexibility in memory management.

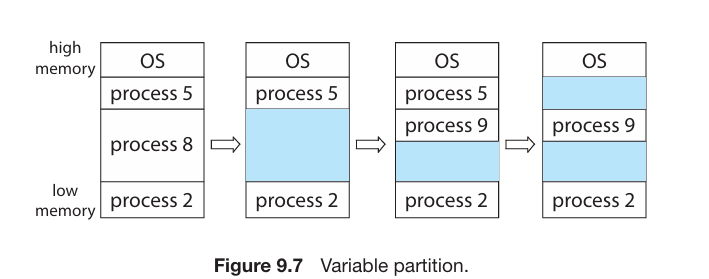


**Memory Allocation**

The **variable partition** scheme for memory allocation involves assigning processes to variably sized partitions in memory, where each partition can contain exactly one process. Initially, all memory is available for user processes and is considered one large block of available memory, called a hole. As processes enter and leave the system, memory holes of various sizes are formed.

When a new process arrives, the operating system considers the memory requirements of the process and the available memory space to determine allocation. If there isn't sufficient memory to satisfy the demands of a process, the system can either reject the process or place it in a wait queue until memory becomes available.

To allocate memory, the system searches for a hole that is large enough for the arriving process. If the hole is too large, it is split into two parts: one allocated to the process and the other returned to the set of holes. When a process terminates, its memory block is released and returned to the set of holes. If the new hole is adjacent to other holes, they are merged to form a larger hole.



This memory allocation procedure is a specific instance of the general **dynamic storage allocation problem**, which deals with satisfying requests of varying sizes (n) from a list of free holes. Common strategies for selecting a free hole include first fit, best fit, and worst fit.

* **First fit**: Allocate the first hole that is big enough. The search can start from the beginning of the list or where the previous search ended.
* **Best fit**: Allocate the smallest hole that is big enough. This strategy produces the smallest leftover hole and may require searching the entire list unless it's ordered by size.
* **Worst fit**: Allocate the largest hole. Like best fit, it may require searching the entire list unless it's sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations suggest that both first fit and best fit are generally better than worst fit in terms of decreasing time and storage utilization. While neither first fit nor best fit is clearly superior in terms of storage utilization, first fit tends to be faster.

**Fragmentation**

Memory allocation strategies like first-fit and best-fit suffer from **external fragmentation**, where although there may be enough total memory to satisfy a request, the available spaces are not contiguous, leading to fragmentation into small holes. This fragmentation problem can be severe, potentially resulting in wasted memory between every two processes.

The choice between first-fit and best-fit strategies can affect the amount of fragmentation, with different systems favoring one over the other. However, regardless of the strategy used, external fragmentation remains a challenge. Statistical analysis suggests that with first fit; approximately half of the memory may be lost to fragmentation, known as the **50-percent rule**.

**Internal fragmentation** can also occur, where memory allocated to a process may be larger than the requested memory, resulting in unused memory within the partition.

One solution to external fragmentation is **compaction**, which involves rearranging memory to consolidate free memory into one large block. Compaction is only possible with dynamic relocation at execution time.

Another solution to external fragmentation is permitting noncontiguous logical address spaces for processes, as seen in **paging**, a common memory-management technique. Paging allows processes to be allocated physical memory wherever it's available, mitigating the impact of external fragmentation.

**Paging**

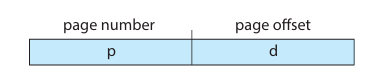
**Paging** is a memory management technique that allows a process's physical address space to be non-contiguous, thus avoiding the need for contiguous allocation.

This method effectively eliminates external fragmentation and the need for compaction, common issues with contiguous memory allocation. Paging offers numerous advantages and is widely used in operating systems across various platforms, from large servers to mobile devices. Its implementation involves cooperation between the operating system and the computer hardware.

**Basic Method**

Paging involves dividing physical memory into fixed-sized blocks called **frames** and logical memory into blocks of the same size called **pages**. When a process is executed, its pages are loaded into available memory frames from their source, which could be a file system or a **backing store**. The *backing store* is divided into fixed-sized blocks matching the size of memory frames or clusters of multiple frames. This approach separates the logical address space from the physical address space, allowing a process to have a logical address space larger than the physical memory available in the system, such as a process having a 64-bit logical address space even if the system has less than 2^64 bytes of physical memory.

In the paging model of memory management, every address generated by the CPU is divided into a page number (p) and a page offset (d).

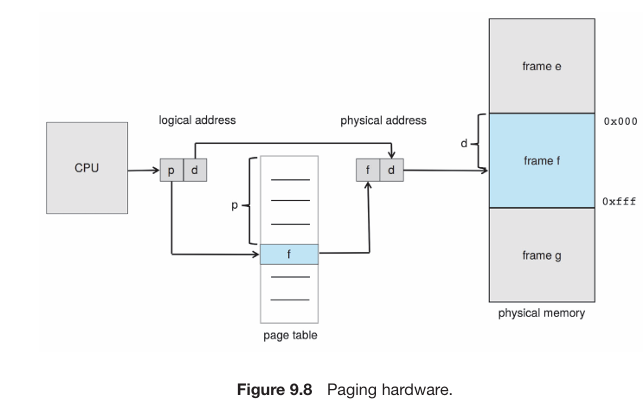


The page number serves as an index into a per-process page table, which contains the base address of each frame in physical memory. The page offset specifies the location within the frame being referenced. To translate a logical address to a physical address, the Memory Management Unit (MMU) follows these steps:

1. Extract the page number (p) and use it to index the page table.

2. Extract the corresponding frame number (f) from the page table.

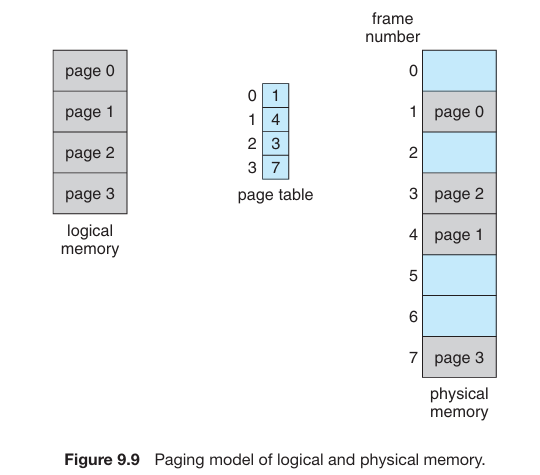
3. Replace the page number (p) in the logical address with the frame number (f), while keeping the offset (d) unchanged.

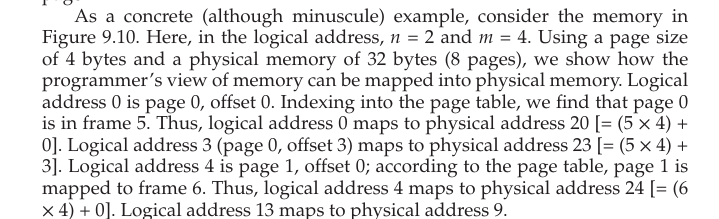


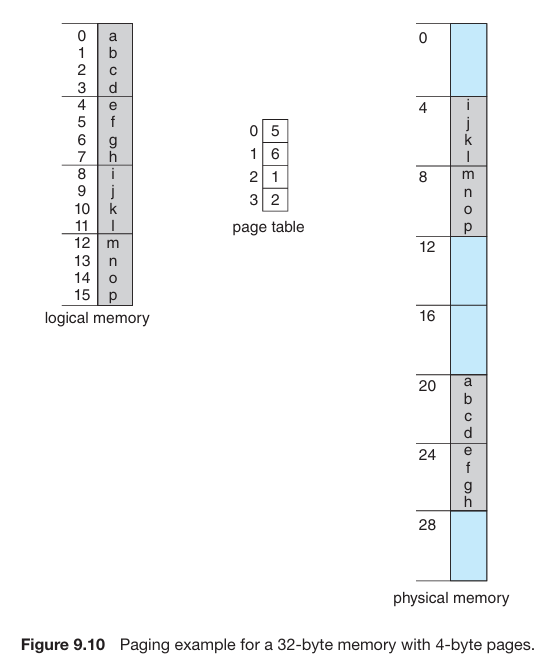
This combination of frame number and offset constitutes the physical address. The page size, like the frame size, is determined by hardware and is typically a power of 2, ranging from 4 KB to 1 GB per page depending on the computer architecture. The choice of a power of 2 as the page size simplifies the translation process, where the high-order bits (m – n) of a logical address designate the page number and the low-order (n) bits designate the page offset.



Where logical address space is 2^m and page size is 2^n and p is index to the page table and d is the displacement within the page.







4 \* 2 equals 8 pages.   
2^2 equals 4 Bytes of page size.  
Hence 8 \* 4 equals 32 Bytes of physical memory.

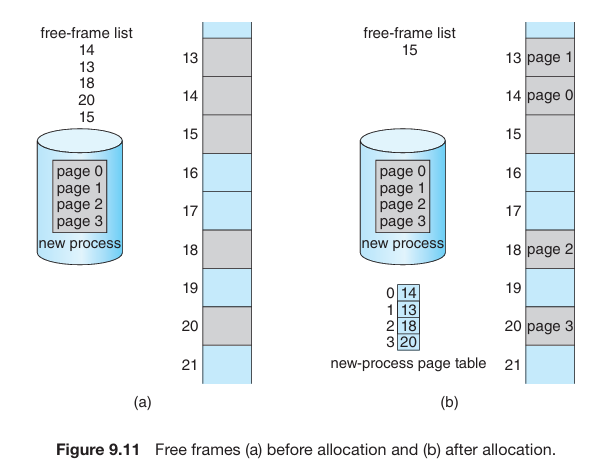
Paging in memory management serves as a dynamic relocation method, where every logical address gets bound by paging hardware to a physical address. This approach resembles using a table of base (or relocation) registers, with one for each frame of memory.

Paging eliminates external fragmentation as any free frame can be allocated to a process. However, internal fragmentation may occur since frames are allocated as units and may not perfectly align with the memory requirements of a process. For instance, if a process requires 72,766 bytes and the page size is 2,048 bytes, it would be allocated 36 frames, leaving internal fragmentation of 962 bytes. In the worst-case scenario, a process needing n pages plus 1 byte would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame.

When process size is independent of page size, internal fragmentation is expected to average around one-half page per process. This suggests that smaller page sizes are desirable. However, smaller page sizes incur overhead in each page table entry, which is reduced as page sizes increase. Additionally, larger page sizes contribute to more efficient disk I/O when transferring larger amounts of data. Over time, page sizes have generally increased to accommodate larger processes, data sets, and main memory sizes. Currently, typical page sizes range from 4KB to 8KB, with some systems supporting even larger sizes. Certain CPUs and operating systems, like x86-64 systems with Windows 10 and Linux, support multiple page sizes, such as 4KB and 2MB, or even larger architecture-dependent sizes referred to as huge pages.

On a 32-bit CPU, each page-table entry is commonly 4 bytes long, although this size can vary. A 32-bit entry can point to one of 2^32 physical page frames. If the frame size is 4KB (2^12), then a system with 4-byte entries can address 2^44 bytes (or 16TB) of physical memory. It's important to note that the size of physical memory in a paged memory system is typically different from the maximum logical size of a process.

When a process arrives in the system to be executed, its size, expressed in pages, is assessed. Each page of the process requires one frame in memory. Therefore, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to the arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is recorded in the page table for this process. Subsequently, each subsequent page is loaded into another frame, and its frame number is recorded in the page table accordingly. This process continues until all pages of the process are loaded into memory.



Paging enables a clear distinction between the programmer's conceptual view of memory and the physical reality. From the programmer's standpoint, memory is perceived as a unified space containing only their program. However, in physical memory, user programs are dispersed alongside others. Address translation hardware bridges this disparity by converting logical addresses to physical ones, a process hidden from the programmer and managed by the operating system. Importantly, user processes are confined to accessing only the memory they own, unable to address memory beyond their page table, which exclusively includes their allocated pages.

The operating system manages physical memory through a system-wide data structure known as a **frame table**, which records allocation details such as which frames are free or allocated, and to which process's page they are allocated if applicable.

User processes operate in user space, necessitating the mapping of logical addresses to physical addresses. The operating system maintains a copy of the page table for each process, akin to how it keeps track of the instruction counter and registers contents. This copy facilitates the translation of logical addresses to physical addresses when needed, such as during system calls. It is also utilized by the CPU dispatcher to establish the hardware page table when a process is allocated the CPU. However, paging increases context-switch time due to the additional operations involved.

**Hardware Support**

Page tables, which are per-process data structures, are stored in the process control block along with other register values like the instruction pointer. When the CPU scheduler selects a process for execution, it reloads the user registers and the appropriate hardware page-table values from the stored user page table.

The hardware implementation of the page table can vary. In a simple setup, the page table is implemented using dedicated high-speed hardware registers, ensuring efficient page-address translation. However, this approach increases context-switch time because each register must be exchanged during a context switch.

For contemporary CPUs that support larger page tables, storing the page table in fast registers is not feasible. Instead, the page table is kept in main memory, and a page-table base register (PTBR) points to it. Changing page tables then only requires modifying this single register, significantly reducing context-switch time.

**Translation Look Aside Buffer**

Storing the page table in main memory can lead to slower memory access times despite faster context switches. When accessing a location i, the system must first index into the page table using the value in the page-table base register (PTBR) offset by the page number for i, requiring one memory access. This provides the frame number, which is then combined with the page offset to generate the actual address, requiring a second memory access to access the desired data. Consequently, two memory accesses are needed to access data, effectively slowing down memory access by a factor of 2, which is often considered intolerable in most circumstances.

The standard solution to the problem of slower memory access due to storing the page table in main memory is to employ a translation look-aside buffer (TLB). The TLB is a special, small, fast-lookup hardware cache that facilitates quick translation of logical addresses to physical addresses. It is associative, high-speed memory, with each entry containing a key (or tag) and a corresponding value. When presented with an item, the TLB compares it with all keys simultaneously, and if a match is found, the corresponding value is returned.

Modern TLB lookup operations are typically integrated into the instruction pipeline, incurring minimal performance overhead. However, to maintain this efficiency, TLBs must be kept small, typically ranging from 32 to 1,024 entries in size. Some CPUs implement separate TLBs for instruction and data addresses, effectively doubling the number of TLB entries available.

This development exemplifies the evolution of CPU technology, progressing from systems with no TLBs to those with multiple levels of TLBs, mirroring the presence of multiple levels of caches.

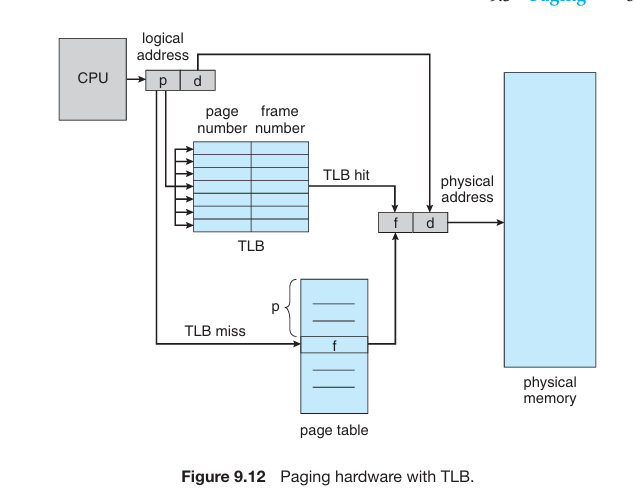
The TLB (Translation Look-aside Buffer) is utilized in conjunction with page tables to expedite address translation. When a logical address is generated by the CPU, the Memory Management Unit (MMU) first checks if its page number is present in the TLB. If found, then the corresponding frame number is readily available and used to access memory. This process, integrated into the instruction pipeline, incurs no performance penalty compared to systems without paging.

In the event of a TLB miss (the page number is not in the TLB), address translation follows standard steps where a memory reference to the page table is required. Once the frame number is obtained, it is used to access memory. Additionally, the page number and frame number are added to the TLB, enhancing future lookup efficiency.

TLBs (Translation Look-aside Buffers) employ replacement policies when they are full and need to select an existing entry for replacement. These policies can range from least recently used (LRU) to round-robin or random selection. Certain CPUs permit the operating system to participate in LRU entry replacement, while others manage it independently. Additionally, some TLBs allow certain entries to be "**wired down**," preventing their removal from the TLB. Typically, TLB entries for critical kernel code are wired down, ensuring their availability and minimizing translation overhead.

Some TLBs include **Address-Space Identifiers** (ASIDs) in each TLB entry. ASIDs uniquely identify each process and are used for address-space protection. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, it is treated as a TLB miss.

ASIDs allow the TLB to contain entries for several different processes simultaneously, enhancing efficiency. Without separate ASIDs, the TLB would need to be flushed (or erased) with each context switch to prevent the next executing process from using incorrect translation information leftover from the previous process. This flushing ensures that the TLB does not contain outdated entries with valid virtual addresses but incorrect or invalid physical addresses.



The hit ratio in a TLB (Translation Look-aside Buffer) refers to the percentage of times the desired page number is found in the TLB. For example, an 80-percent hit ratio indicates that the desired page number is found in the TLB 80 percent of the time. When the page number is found in the TLB, a mapped-memory access takes the memory access time, such as 10 nanoseconds in this example.

However, if the page number is not found in the TLB, additional memory accesses are required. First, memory must be accessed for the page table and frame number (10 nanoseconds), and then the desired byte in memory must be accessed (10 nanoseconds), totaling 20 nanoseconds.

To calculate the effective memory-access time, the weighted average of these cases is determined by their probabilities. For example, with an 80-percent hit ratio:

Effective access time = (0.80 \* 10) + (0.20 \* 20) = 12 nanoseconds.

For a more realistic hit ratio, such as 99 percent:

Effective access time = (0.99 \* 10) + (0.01 \* 20) = 10.1 nanoseconds.

This demonstrates that a higher hit rate in the TLB results in a minimal slowdown in average memory-access time.

**Memory Protection**

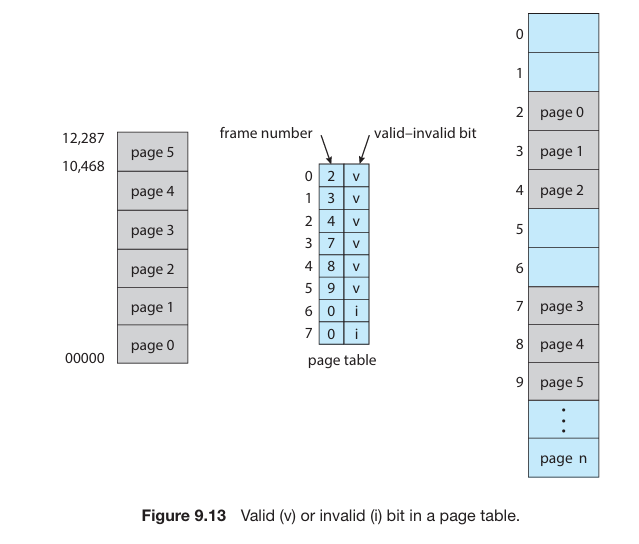
Memory protection in a paged environment is achieved through protection bits associated with each frame, typically kept in the page table. These bits define whether a page is read-write or read-only. When memory is accessed, the protection bits are checked to ensure that no writes are made to a read-only page. Any attempt to write to a read-only page results in a hardware trap to the operating system, signaling a memory protection violation.

This approach can be expanded to provide finer levels of protection, including read-only, read-write, or execute-only access, with separate protection bits for each type of access. Illegal attempts trigger traps to the operating system.

Each entry in the page table also includes a valid-invalid bit. When set to valid, the associated page is part of the process's logical address space and is legal. When set to invalid, the page is not part of the process's logical address space, and illegal addresses are trapped using this bit. The operating system sets this bit to control access to pages.

For example, in a system with a 14-bit address space and a program using addresses 0 to 10468 with a page size of 2KB, addresses beyond 10468 are invalid. The valid-invalid bit prevents access to these addresses, trapping attempts to access them.

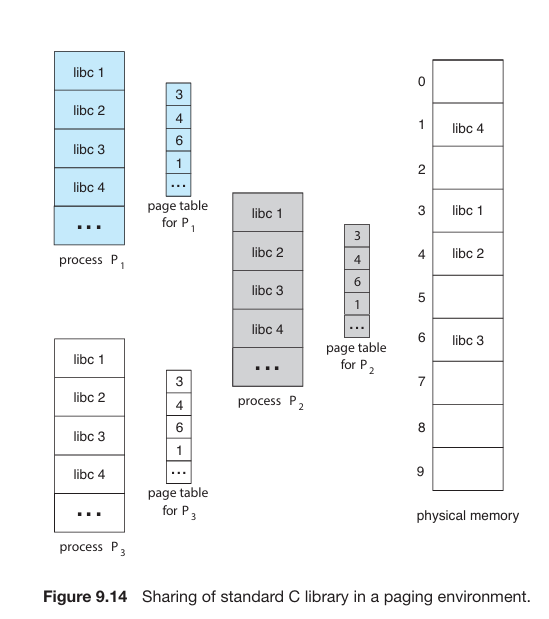
However, this scheme can lead to a problem where pages beyond the valid range of a program are still classified as valid due to the page table setup. For instance, in a system with a 14-bit address space and a program using addresses up to 10468 with a 2KB page size, addresses up to 12287 are erroneously deemed valid. This highlights the issue of internal fragmentation in paging, where memory allocations may not align perfectly with page boundaries, leading to inefficiencies in memory usage.



To optimize memory usage, systems may provide a page-table length register (PTLR) to indicate the size of the page table. This value is checked against every logical address to verify its validity. Failure of this test results in an error trap to the operating system.

**Shared Paging**

Paging offers the advantage of facilitating the sharing of common code, a crucial consideration in environments with multiple processes. For example, the standard C library (libc) is often shared among various processes in UNIX and Linux systems. Traditionally, each process would load its own copy of libc into memory, resulting in significant memory consumption, especially in systems with numerous processes.



However, if the code is reentrant—non-self-modifying and unchanging during execution—it can be shared among processes. By mapping the page table of each user process onto the same physical copy of libc, multiple processes can execute the same code concurrently, with each having its own set of registers and data storage. This sharing reduces memory requirements significantly, as only one copy of the library needs to be kept in physical memory, saving substantial space.

This sharing extends beyond runtime libraries like libc to encompass other frequently used programs such as compilers, window systems, and databases. Shared libraries, typically implemented with shared pages, require reentrant code to ensure correctness, which is enforced by the operating system.

The concept of sharing memory among processes parallels the sharing of address space by threads and the use of shared memory as a method of inter-process communication. Some operating systems implement shared memory using shared pages.

**Structure of a Page Table**

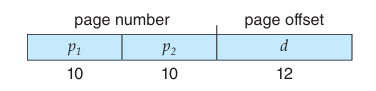
Three types:

* Hierarchal
* Hashed
* Inverted

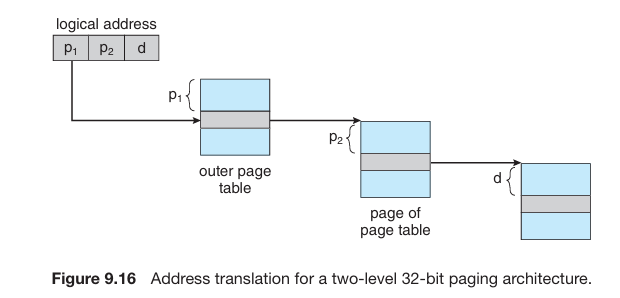
**Hierarchal Paging**

In systems with large logical address spaces, such as 32-bit systems (2^32 to 2^64), the page table can become excessively large. For instance, in a system with a 32-bit logical address space and a 4 KB page size, the page table may consist of over 1 million entries (220 = 2^32/2^12), requiring up to 4 MB of physical address space for the page table alone. To address this issue, the page table can be divided into smaller pieces, achieved through various methods.

One approach is to employ a two-level paging algorithm, where the page table itself is paged. For example, in a system with a 32-bit logical address space and a 4 KB page size, a logical address is divided into a 20-bit page number and a 12-bit page offset. By paging the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.



Where p1 is an index to the outer page table and p2 is the displacement within the page table. The translation is as follows:

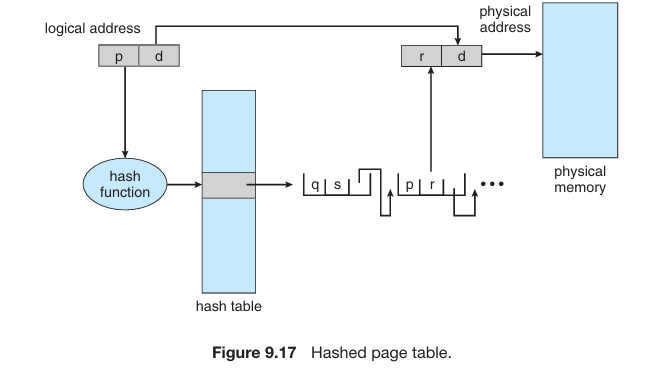


Advantages and disadvantages from slides.

**Hashed Paging**

For handling address spaces larger than 32 bits, one approach is to utilize a hashed page table, where the hash value corresponds to the virtual page number. Each entry in the hash table contains a linked list of elements, accommodating collisions. These elements comprise three fields: the virtual page number, the mapped page frame value, and a pointer to the next element in the linked list.

The algorithm proceeds as follows: the virtual page number in the virtual address undergoes hashing into the hash table. The virtual page number is then compared with the first element's virtual page number in the linked list. If a match is found, the corresponding page frame value is used to derive the desired physical address. In the absence of a match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme allows for efficient handling of large address spaces, as illustrated in Figure 9.17.



A proposed variation for managing large 64-bit address spaces involves clustered page tables. These are akin to hashed page tables but with a key difference: each entry in the hash table references multiple pages, typically around 16, instead of just one. Consequently, a single page-table entry can accommodate mappings for multiple physical-page frames.

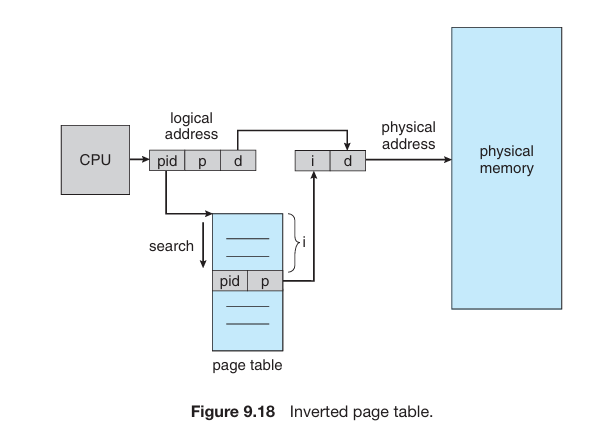
Clustered page tables offer particular advantages for sparse address spaces where memory references are noncontiguous and dispersed throughout the address space. By grouping multiple pages within a single entry, clustered page tables reduce the overhead associated with managing sparse address spaces, leading to more efficient memory utilization.

**Inverted Paging**

Typically, each process is associated with its own page table, which contains an entry for each page that the process is using, irrespective of the validity of the virtual address. This allows the operating system to translate virtual addresses into physical memory addresses efficiently. However, a drawback of this method is the potential for large memory consumption due to the millions of entries in each page table.

To address this issue, an inverted page table can be used. In an inverted page table, there is one entry for each real page (or frame) of memory. Each entry stores the virtual address of the page stored in that physical memory location, along with information about the process that owns the page. This approach reduces the memory overhead since there is only one page table in the system, with each entry corresponding to a page of physical memory.

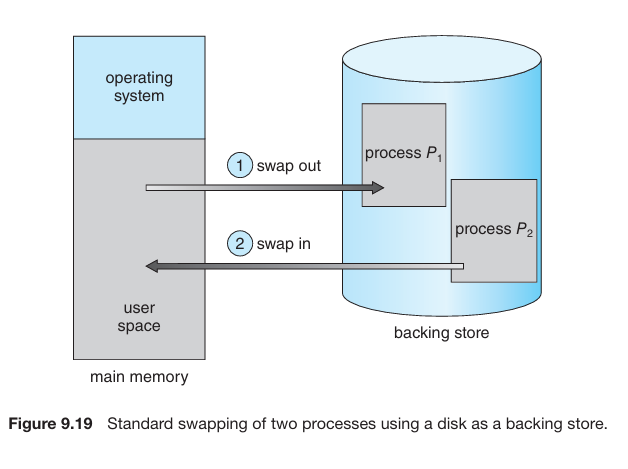
Inverted page tables may require storing an address-space identifier in each entry to distinguish between different address spaces mapping physical memory. This ensures that a logical page for a specific process is mapped to the corresponding physical page frame. Systems utilizing inverted page tables include the 64-bit UltraSPARC and PowerPC architectures.



**Swapping**

**Swapping** is a technique that allows a process or a portion of it to be temporarily moved out of memory to a *backing store*, such as disk, and later brought back into memory for continued execution.

This process enables the total physical address space of all processes to surpass the available physical memory in the system, thereby enhancing the degree of multiprogramming. Swapping effectively manages memory by allowing processes to utilize more memory than is physically available, thus optimizing system resources and supporting concurrent execution of multiple processes.



There are two types discussed in this chapter:

* Standard
* Swapping with paging

|  |  |
| --- | --- |
| **Standard Swapping** | **Swapping With Paging** |
| Standard swapping involves transferring entire processes between main memory and a backing store, typically a fast secondary storage medium. The backing store needs to be sufficiently large to hold the parts of processes being swapped and must offer direct access to these memory images. When a process or its part is swapped out, associated data structures, including those for multithreaded processes, are also written to the backing store. The operating system maintains metadata for swapped-out processes to facilitate their restoration when swapped back into memory.  The primary advantage of standard swapping is its ability to oversubscribe physical memory, enabling the system to accommodate more processes than the available physical memory can hold. Inactive or idle processes are commonly swapped out, freeing up memory for active processes. When an inactive process becomes active again, it must be swapped back into memory. | Traditional swapping, which involves moving entire processes between main memory and a backing store, is no longer commonly used in contemporary operating systems like Linux and Windows. Instead, a variation of swapping known as paging is employed, where individual pages of a process are swapped in and out of memory.  This approach allows for oversubscription of physical memory without incurring the prohibitive time cost associated with swapping entire processes. In paging, a "page out" operation moves a page from memory to the backing store, while the reverse process is called "page in". This distinction in terminology reflects the shift away from traditional swapping towards paging-based swapping, which is more efficient and practical in modern systems. |

