Justin Ciocoi

Dec. 16, 2023

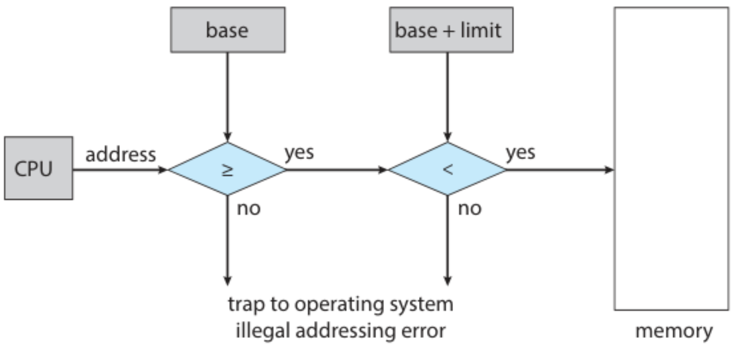
CSCI 375 Textbook Notes

Chapter 9: Main Memory

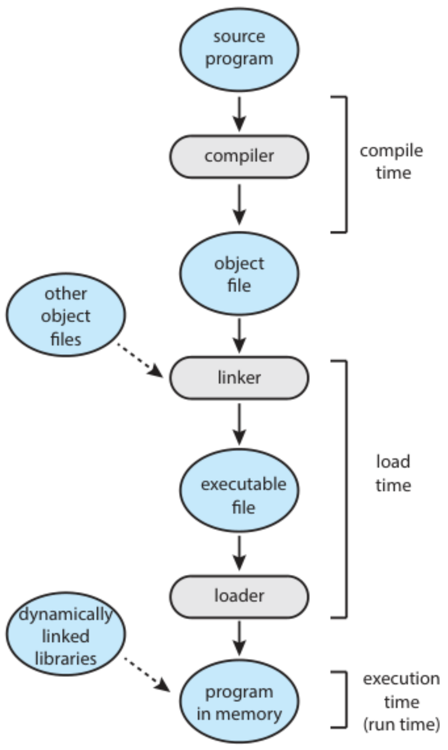
9.1: Background

- Main memory consists of a large array of bytes each with its own address
- A typical instruction0execution cycle will first fetch an instruction from memory, then decode the instruction, which may cause operands to be fetched from memory, then execute and possibly store results back in memory
- We can ignore how a program generates memory addresses, as from the point of view of a memory unit, we see only a continuous stream of memory addresses and are not necessarily interested in their methods of origin
- **9.1.1: Basic Hardware**
    - Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly
    - Therefore, if data is not in memory or in a CPU register, it must first be moved into memory before the CPU can execute on it
    - CPU registers are generally accessible once per every tick of the CPU clock, but completing memory accesses may take many cycles of the CPU clock, and in these cases, the processor must stall
        - In order to ease the issue of stalling, caches were introduced for CPUs such that more frequently used or likely to be used memory will be held in fast memory between the CPU and main memory
    - Speed is not the only consideration in the accessing of physical memory, as we must also ensure correct operation
    - We have to make sure that each process has a separate memory space

        - This protects processes from each other and is fundamental to having multiple processes loaded into memory for concurrent execution
    - To do this, we must first have a method by which to find the range f legal memory addresses which processes may enter
    - We can provide this function by using two registers, namely the *base register* and *limit register*
        - The base register will hold the smallest legal physical memory address
        - The limit register specifies the size of the range
    - Protection of the memory space is accomplished by having CPU hardware compare every address generated in user mode with the registers
    - This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users
    - The base and limit registers can be loaded only by the operating system, which prevents programs from changing the registers' contents

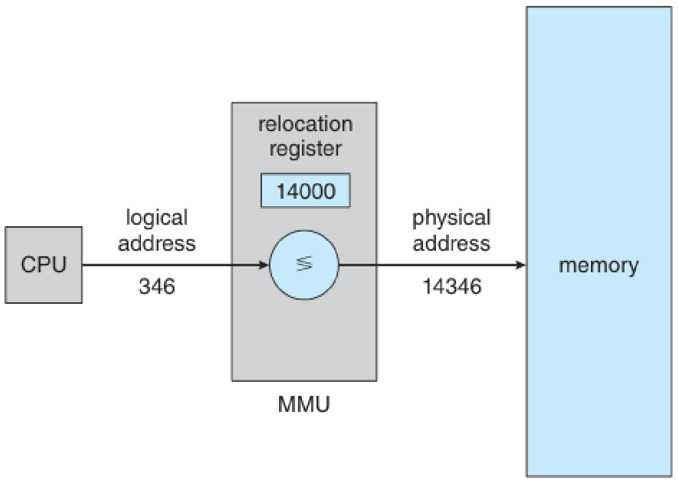


- **9.1.2: Address Binding**
    - In most cases, a program resides on a disk as a binary executable file
    - In order to run a program, the program must be brought into memory and placed within the context of a process where it becomes eligible for execution on an available CPU
    - As the process executes, it accesses instructions and data from memory, and once it eventually terminates its memory is reclaimed for use by other processes
    - Here we can see a visual representation of the multi-step processing of a user program



- **9.1.3: Logical Vs. Physical Address Space**
  - An address generated by the CPU is commonly referred to as a *logical address*
  - An address seen by the memory unit, that is, the one loaded into the memory-address register of the memory, is commonly referred to as a *physical address*
  - Binding addresses at either compile or load time generates identical logical and physical addresses
  - However, the execution-time address binding scheme results in differing logical and physical addresses
  - The set of all logical addresses generated by a program is called a logical address space, and the set of all physical addresses corresponding to these logical addresses is called a physical address space
  - The run-time mapping from virtual address to physical address is done by a hardware device called the *memory-management unit (MMU)*
  - Here, the base register is called a *relocation register* and the value in the relocation register is added to every address generated by a user process at the time the address is sent to memory
    - If a user program wants to access address location 5, and the relocation register is at 14000, the attempt to access is dynamically relocated to address location 14005

- So, we have two different types of addresses which can be thought of in the following manner
    - Logical addresses, in the range from 0 to *max*
    - Physical addresses, in the range from $R + 0$ to $R+$*max* for a base value R



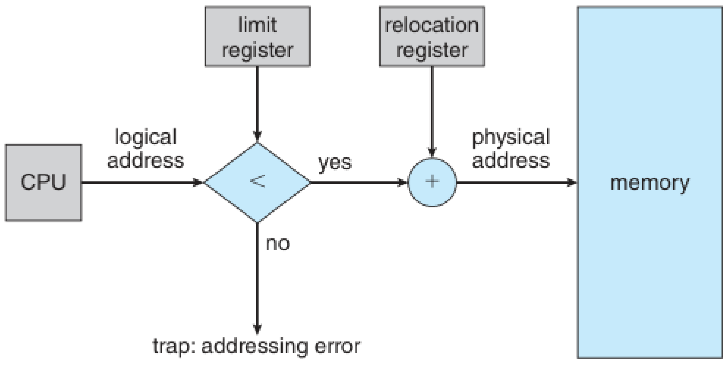
- **9.1.4: Dynamic Loading**
    - So far, we have assumed that it is necessary for an entire program and all data of a process to be in physical memory in order for the process to execute
    - Therefore, the size of a process will be limited to the size of physical memory
    - In order to obtain better utilization of the memory space, we can use *dynamic loading*
    - With dynamic loading, a routine is not loaded until it is called
    - All routines are kept on disk in a relocatable load format, and the main program is loaded into memory and executed
    - When a routine needs to call another routine, the calling routine first checks to see if the other routine has been loaded
    - If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change
    - This provides the advantage of not having to always load infrequently used routines that consist of large amounts of code
- **9.1.5: Dynamic Linking and Shared Libraries**

-
    - *Dynamically Linked Libraries (DLLs)* are system libraries that are linked to user programs when the programs are run
    - Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image
    - DLLs allow user programs to not need copies of certain system libraries such as the standard C language library
    - DLLs also can be shared among multiple processes such that only one copy of the DLL is in main memory
    - DLLs are used extensively in Windows and Linux systems
    - When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary
    - Dynamic linking and shared libraries generally require help from the operating system to function effectively, since processes in memory are protected from one another and would have trouble accessing shared memory

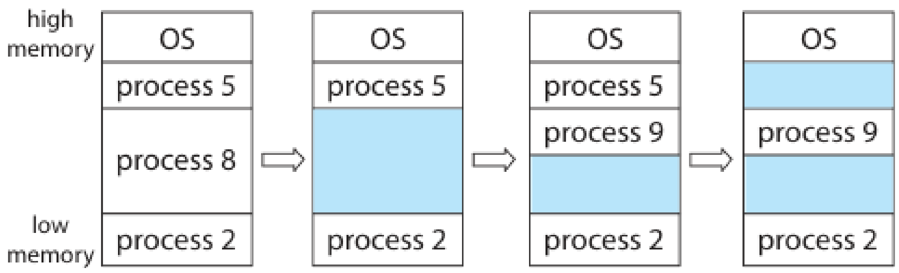
9.2: Contiguous Memory Allocation

- The main memory in a system must accommodate both the operating system and user processes
- Thus, we need to allocate main memory in the most efficient way possible
- Usually, memory is divided into two partitions, one for the operating system, and one for user processes, and usually operating systems are placed in high memory
- Generally, multiple user processes will reside in memory at the same time, so we therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory
- In *contiguous memory allocation*, each process is contained in a single section of memory which is contiguous to the section containing the next process
- **9.2.1: Memory Protection**
    - We must prevent a process from accessing memory it does not own by combining two previously discussed ideas

- In order to accomplish this goal, we can simply implement both a relocation register, and a limit register



- **9.2.2: Memory Allocation**
    - One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process
    - In this scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied
    - Initially, all memory is available for user processes and is considered one large block of available memory, or a *hole*, and eventually the memory will contain a set of holes of various sizes
    - Here, we can see this variable partition scheme



    - What happens when there isn't sufficient memory to satisfy the demands of an arriving process?
        - One option is to simply reject the process and provide an appropriate error message
        - Alternatively, we can place these processes into a wait queue where they will be placed into memory when it becomes available

    - When processes release their memory and form contiguous holes, they will merge to form one larger hole
    - How do we decide which hole in which to place an arriving process if multiple different holes satisfy the memory requirements
    - The *first-fit, best-fit, and worst-fit* strategies are the most commonly used
    - In the *first-fit* method, the first hole that is big enough to accommodate the process will be used
    - In the *best-fit* method, we will allocate the smallest hole which is large enough to accommodate the request
        - The entire list must be searched, unless the list is ordered by size
        - This strategy produces the smallest leftover hole
    - In the *worst-fit* method, the largest hole will be allocated
        - Again, the entire list must be searched unless it is ordered by size
        - This strategy produces the largest leftover hole, which may be more useful than the smallest leftover hole produced in the best-fit method
    - Simulations have shown that first-fit and best-fit are better than worst-fit in terms of decreasing time and storage utilization
    - Between the two, neither is clearly better in terms of storage utilization, but first fit is generally faster
- **9.2.3: Fragmentation**
    - Both the first- and best-fit strategies for memory allocation suffer from *external fragmentation*
    - External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous
    - No matter which memory allocation algorithm we use, external fragmentation will pose an issue
    - Statistical analysis of the first-fit method reveals that given N allocated blocks, $0.5 * N$ blocks will be lost to fragmentation

  - This idea, known as the *50-percent rule*, means that one third of memory may be unusable
  - Memory fragmentation can be internal as well as external
  - In *internal fragmentation* memory blocks are allocated only in a specific size, and are thus sometimes given to processes that require less than one full block
  - Here, the extra memory allocated to a process which will not be used is considered to be lost to internal fragmentation
  - One solution to the problem of external fragmentation is *compaction*, where the goal is to shuffle memory contents such that free memory is all placed together in one large block
  - Compaction is not always possible, since it requires relocation to be dynamic and be done at execution time
  - We can also allow processes to have a logical address space which is noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available
  - This is the general approach used in *paging*, which is the most common memory-management technique for computer systems

9.3: Paging

- So, far memory management discussions have centered around schemes that require the physical address space of a process to be contiguous
- Now, we introduce *paging*, which is a scheme by which we allow a process's physical address space to be non contiguous
- **9.3.1: Basic Method**
  - The basic method for paging implementation involves breaking physical memory into fixed-size blocks called *frames*, and breaking logical memory into blocks of the same size called *pages*
  - When a process is executed, its pages are loaded into any available memory frames from their source
  - The source is also divided into fixed size blocks which are the size of individual memory frames or clusters of memory frames

- Every address which is generated by the CPU is divided into two parts, namely a *page number (p)* and a *page offset (d)*
- The page number is used as an index into a per-process *page table*
- The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced
- The following outlines the steps taken by the memory management unit in order to translate a logical address generated by the CPU to a physical address
    1. Extract the page number, p, and use it as an index into 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
- Like the frame size, the page size is determined by the hardware, and page sizes are typically powers of two ranging from 4 KB to 1 GB per page
- If the size of the logical address space is 2^m, and a page size is 2^n, then the high order $m - n$ bits of a logical address designate the page number, and the n low order bits designate the page offset
- Thus the logical address consists of p and d where p is an index into the page table and d is the displacement within the page
- If process size is completely independent of page size, we can expect internal fragmentation to average approximately one-half page per process
- This would suggest that smaller page sizes are more advantageous, but it is important also to note that a smaller page size will increase the total number of pages, and thus also increase system overhead
- Today, pages are generally either 4 KB or 8 KB in size, and some systems support larger page sizes
- Some systems even support multiple different page sizes
- When a process arrives in the system to be executed, its size, expressed in pages, is examined by the system
- If a process requires n pages, then at least n frames must be available in memory, and if they are, will be allocated to this arriving process

    - Since the operating system is responsible for managing physical memory, it must be aware of the physical memory's allocation details

    - This information is generally kept in a single, system-wide data structure called a *frame table*, which has one entry for each physical page frame, indicating whether the latter is free or allocated

- **9.3.2: Hardware Support**

    - As page tables are per-process data structures, a pointer to the page table is stored in the process control block (PCB) of each process
    - In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, making the page-address translation a very efficient operation
    - **Translation Look-Aside Buffer**
        - Storing the page table in main memory may yield faster context switch times, but it also will result in slower memory access times
        - It will effectively double the memory access time since each memory access requires an additional memory access for the page-address translation step
        - This delay is considered intolerable on many systems, and the standard solution is to use a special, small, fast-lookup hardware cache called a *Translation Look-Aside Buffer (TLB)*
        - Each entry in the TLB consists of two parts
            - A key
            - A value
        - When the associative memory is presented with an item, the item is compared with all keys simultaneously, and if the item is found, the corresponding value field will be returned
        - The TLB contains only a few of the page-table entries
        - When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB, and if it is, then the frame number is immediately available and is then used to access memory
        - If a TLB is full, then an existing entry must be selected for replacement
        - Policies for removal range from least-recently-used to round-robin or even random
        - Some CPU architectures also allow certain entries to be *wired down* meaning that they can not be removed from the TLB
        - The percentage of times that the page number of interest is found in the TLB is called the *hit ratio*

        - Effective memory access time can be calculated by utilizing the hit ratio, since hits will take one memory access time and misses will take two, thus allowing us to make a good approximation using the hit ratio

- **9.3.3: Protection**
    - Memory protection in a paged environment is accomplished by protection bits associated with each frame, which are normally kept in the page table
    - One bit can define a page to be read-write or read-only
    - When logical addresses are translated to physical addresses, these bits are also checked to verify that the protections are being followed
    - One additional bit is also generally attached to each entry in the page table, known as a *valid-invalid bit*
    - When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal page
    - Otherwise, the page is not in the process's logical address space
- **9.3.4: Shared Pages**
    - One advantage of paging is the possibility of sharing common code, something that is of particular importance in an environment with multiple processes

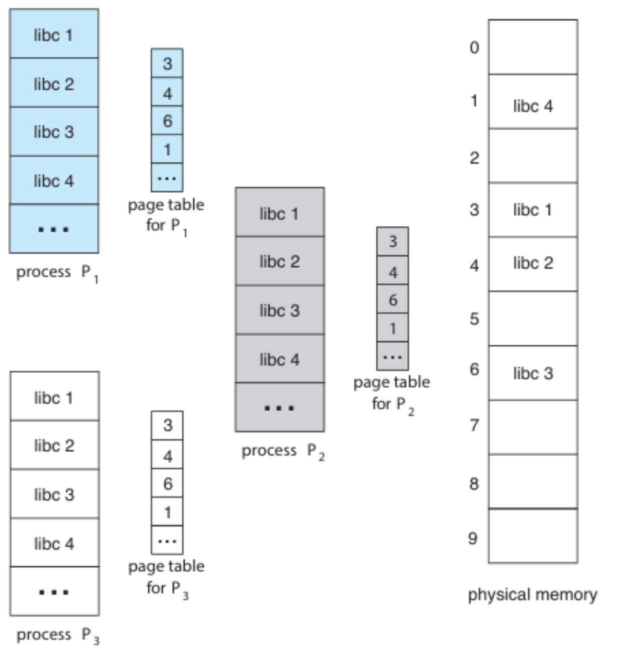


Figure 9.14 Sharing of standard C library in a paging environment.

9.4: Structure of the Page Table

- **9.4.1: Hierarchical Paging**

- In modern environments where systems support large logical address spaces, the page table itself will become excessively large
- In order to remedy this issue, we can implement a two-tiered paging system, where one page table pages a second page table, which is illustrated below

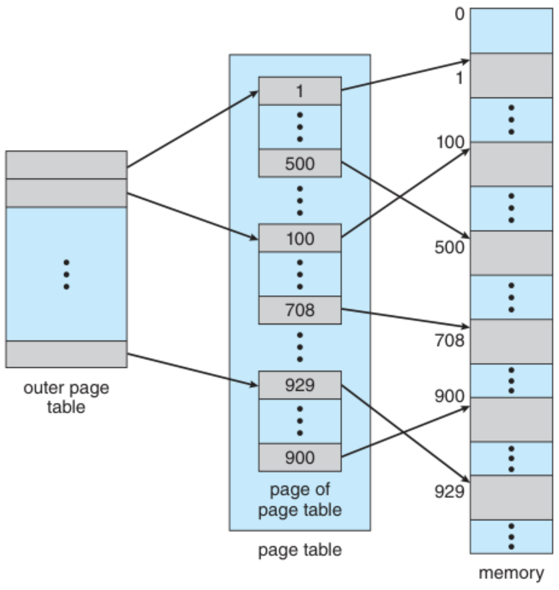
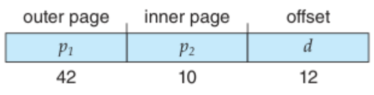
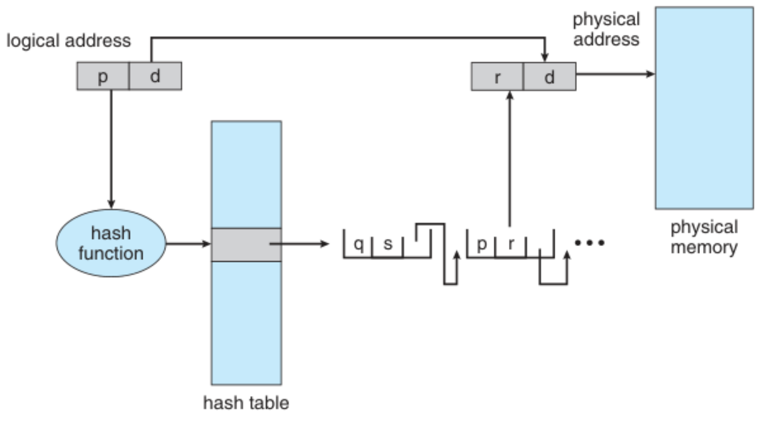


Figure 9.15 A two-level page-table scheme.



- **9.4.2: Hashed Page Tables**
  - Another approach for handling address spaces larger than 32 bits is to use a *hashed page table*



logical address
p
d
physical address
r
d
hash function
q
s
p
r
physical memory
hash table