Memory management

Programs expand to fill the memory available to hold them. Consequently, most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, volatile main memory (RAM), and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage. It is the job of the OS to coordinate how these memories are used. The memory-management algorithms vary from a primitive bare-machine approach to **paging** and **segmentation** strategies.

1. Main memory

- Memory is central to the operation of a modern computer system. The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and deallocate it when they are done,
  - to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory consists of a large array of words or bytes, each with its own address.
  - **Malloc** library call
    - used to allocate memory,
    - finds sufficient contiguous memory,
    - reserves that memory,
    - returns the address of the first byte of the memory.
  - **free** library call
    - give address of the first byte of memory to free,
    - memory becomes available for reallocation.
- Both **malloc** and **free** are implemented using the **brk** system call.
- The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.
- The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.
- Memory management systems can be divided into two classes:
  - Those that move processes back and forth between main memory and disk during execution (swapping and paging), (Memory Abstraction)
  - Those that do not. Simpler. (No Memory Abstraction)

1.1 **Basic Hardware**

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses.

- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete (processor stalls).
- The remedy is to add fast memory between the CPU and main memory (**cache** memory).
- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure correct operation has to **protect** the OS from access by user processes and, in addition, to **protect** user processes from one another.

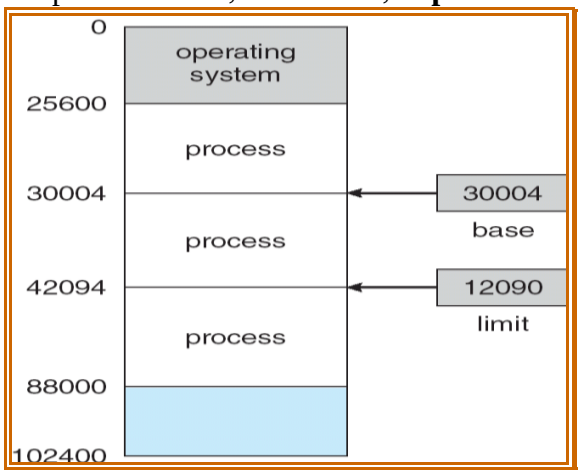


Fig. A base and a limit register define a logical address space

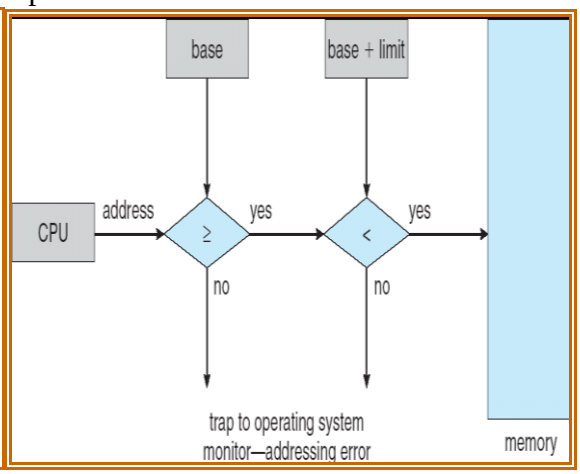
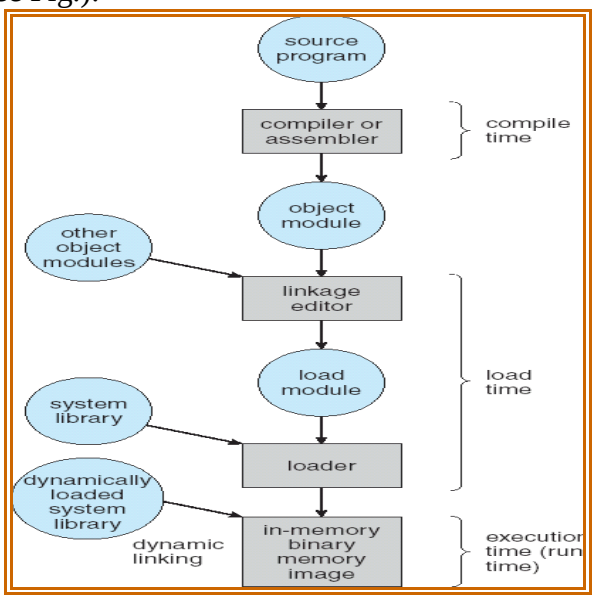


Fig. HW address protection with base and limit registers

- This protection must be provided by the hardware. We first need to make sure that each process has a separate memory space.
- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated in Fig.
  - The **base register** holds the smallest legal physical memory address;
  - The **limit register** specifies the size of the range.
  - For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the OS, which treats the attempt as a fatal error (see Fig.).
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.

1.2 Address Binding

- The process of associating program instructions and data to physical memory addresses is called address binding, or relocation.
- A user program will go through several steps -some of which may be optional-before being executed (see Fig.).

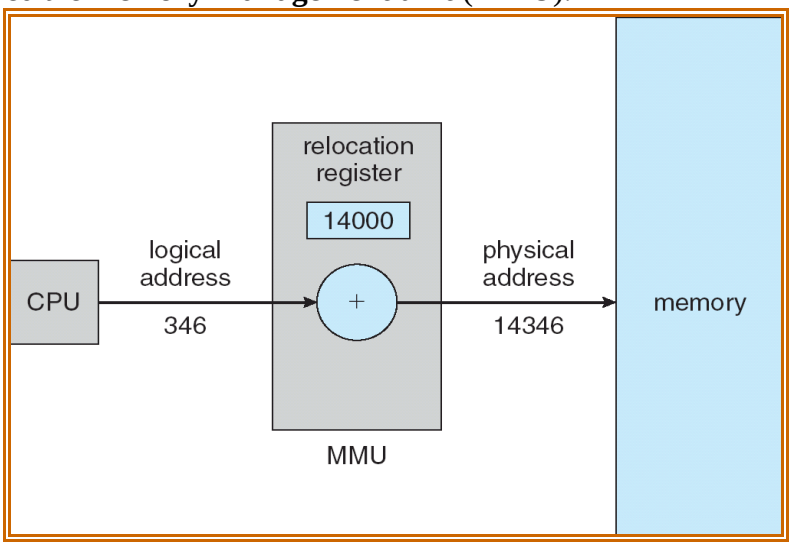


Multi step Processing of a User Program

- Addresses may be represented in different ways during these steps.
  - Addresses in the source program are generally symbolic.
  - A compiler will typically bind these symbolic addresses to **relocatable addresses**.
  - The linkage editor or loader will in turn bind the **relocatable addresses** to **absolute addresses**.
  - Each binding is a mapping from one address space to another.
  - Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
    - **Compile time**. The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute code can be generated (Static).
    - **Load time**. The compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code** (Static).
    - **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. The absolute addresses are generated by hardware. Most general-purpose OSs use this method (Dynamic).
  - **Static**-new locations are determined before execution. **Dynamic**-new locations are determined during execution.

1.3 **Logical Versus Physical Address Space**

- An address generated by the CPU is commonly referred to as a **logical address**, whereas 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**.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- However the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (MMU).



Dynamic relocation using relocation register

- For the time being, we illustrate this mapping with a simple MMU scheme, which is a generalization of the base-register scheme (see Fig.).
  - The base register is now called a **relocation register**.
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses -all as the number 346.
- The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

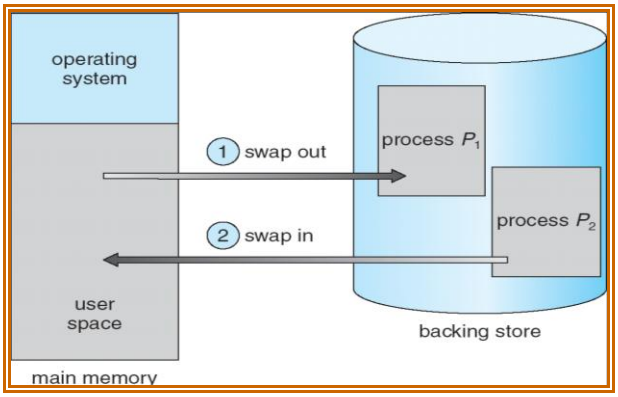
We now have two different types of address: logical address (in the range 0 to max) and physical address (in the range R + 0 to R + max for a base value R). The user generates only logical address and thinks that the process runs in location 0 to max. However, these logical addresses must be mapped to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

1.4 Dynamic Loading

- To obtain better memory-space utilization, 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.
  - The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether 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.
  - Then control is passed to the newly loaded routine.
- The advantage of dynamic loading is that an unused routine is never loaded.
- Dynamic loading does not require special support from the OS. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

2. Swapping

- A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a backing store (disk) and then brought back into memory for continued execution.
- A round-robin CPU-scheduling algorithm; when a quantum expires (see Fig.),
  - The memory manager will start to swap out the process that just finished
  - and to swap another process into the memory space that has been freed.
  - In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
  - When each process finishes its quantum, it will be swapped with another process.



Schematic View of Swapping

- The quantum must be large enough to allow reasonable amounts of computing to be done between swaps.
- A variant of this swapping policy is used for priority-based scheduling algorithms. This variant of swapping is sometimes called **roll out, roll in**.
- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.

- This restriction is dictated by the method of address binding.
  - If binding is done at assembly or load time, then the process cannot be easily moved to a different location.
  - If execution-time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.
- Context-switch time; to get an idea of the context-switch time,
  - Let us assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40 MB per second.
  - The actual transfer of the 10-MB process to or from main memory takes

```
10000 KB/40000 KB per second = 1/4 second
= 250 milliseconds.
```

- Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds.
- Since we must both swap out and swap in, the total swap time is about 516 milliseconds.
- For efficient CPU utilization, we want the execution time for each process to be long relative to the swap time. Thus, the time quantum should be substantially larger than 0.516 seconds.
- Notice that the major part of the swap time is transfer time. Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible.
- Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle.
- Currently, standard swapping is used in few systems. A modification of swapping is used in many versions of UNIX.
  - Swapping is normally disabled but will start if many processes are running and are using a threshold amount of memory.
  - Swapping is again halted when the load on the system is reduced.

2.1 **Constraint of swapping**

- Context switch time in such system is fairly high
- Major part of the swap time is transfer time and the total transfer time is directly proportional to the amount of memory swapped.
- A process may wait for I/O operation when we want to swap the process to free up memory. If the I/O is asynchronously accessing the user memory for I/O buffer, then the process cannot be swapped. Solution to this problem is to :
  - Never swap a process with pending I/O
  - Execute I/O operation only into OS buffers. Transfer between OS buffer and process memory then occur only when the process is swapped in.

3. Contiguous Memory Allocation

Contiguous memory allocation is one of the efficient ways of allocating main memory to the processes. The memory is divided into two partitions. One for the Operating System and another for the user processes. Operating System is placed in low or high memory depending on the interrupt vector placed. In contiguous memory allocation each process is contained in a single contiguous section of memory.

3.1 Memory Mapping and Protection

Memory protection is required to protect Operating System from the user processes and user processes from one another. A relocation register contains the value of the smallest physical address for example say 100040. The limit register contains the range of logical address for example say 74600. Each logical address must be less than limit register. If a logical address is greater than the limit register, then there is an addressing error and it is trapped. The limit register hence offers memory protection. The MMU, that is, Memory Management Unit maps the logical address dynamically, that is at run time, by adding the logical address to the value in relocation register. This added value is the physical memory address which is sent to the memory.

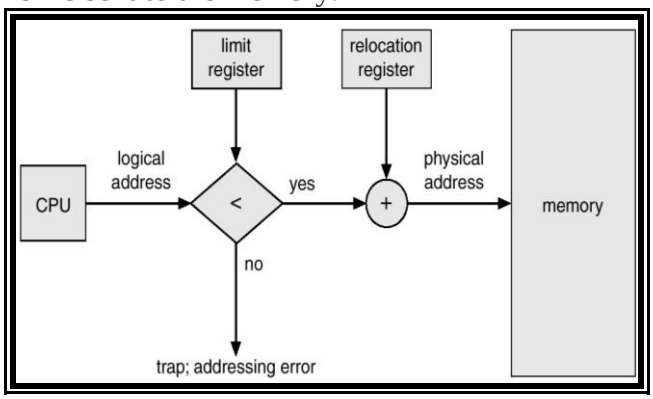


Fig. Hardware Support for Relocation and Limit Registers

The CPU scheduler selects a process for execution and a dispatcher loads the limit and relocation registers with correct values. The advantage of relocation register is that it provides an efficient way to allow the Operating System size to change dynamically.

For example, the OS contains code and buffer space for device drivers.

- If a device driver (or other OS service) is not commonly used, we do not want to keep the code and data in memory.
- Such code is sometimes called **transient** OS code; it comes and goes as needed.
- Thus, using this code changes the size of the OS during program execution.

3.2 Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this **multiple-partition** method,

  - When a partition is free, a process is selected from the input queue and is loaded into the free partition.
  - When the process terminates, the partition becomes available for another process.

- This was used by IBM OS/360 called MFT. MFT stands for Multiprogramming with a Fixed number of Tasks. This method is no longer in use.
- The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. In the fixed-partition scheme,
  - The OS 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, a hole.

    - When a process arrives and needs memory, we search for a hole large enough for this process.
    - If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- At any given time, we have a list of available block sizes and the input queue. The OS can order the input queue according to a scheduling algorithm.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- This procedure is a particular instance of the general **dynamic storage-allocation** problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem.
    - **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
    - **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
    - **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is 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 have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
- Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Disadvantage of Contagious memory allocation

Contagious memory allocation suffer from fragmentation

Fragmentation

In computer storage, **fragmentation** is a phenomenon in which storage space is used inefficiently, reducing storage capacity and in most cases reducing the performance.

- Both the first-fit 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; storage is fragmented into a large number of small holes.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5N$ blocks will be lost to fragmentation.
- That is, one-third of memory may be unusable! This property is known as the 50 percent rule.
- Memory fragmentation can be **internal** as well as external.
    - Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.
    - Suppose that the next process requests 18,462 bytes.
    - If we allocate exactly the requested block, we are left with a hole of 2 bytes.

    - The difference between these two numbers is internal fragmentation; memory that is internal to a partition but is not being used.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

Solution to Fragmentation problem:

- One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
    - Compaction is not always possible. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible only if relocation is dynamic and is done at execution time.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be **non-contiguous**, thus allowing a process to be allocated physical memory wherever the latter is available.

4. Non-contiguous memory allocation

Non-contiguous memory allocation allow a process to be allocated physical memory wherever the latter is available

Two complementary techniques achieve this solution:

- paging
- segmentation

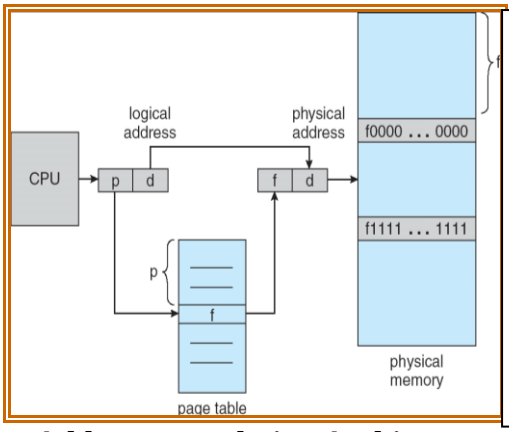
4.1 **Paging**

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous which avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store. Paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and OS, especially on 64-bit microprocessors.

4.2 Basic Method of Paging

Paging is a non contiguous memory allocation method in which physical memory is divided into fixed sized blocks called frames of size in the power of 2, ranging from 512 to 8192 bytes. Logical memory is also divided into same size blocks called pages. For a program of size *n* pages to be executed, *n* free frames are needed to load the program.

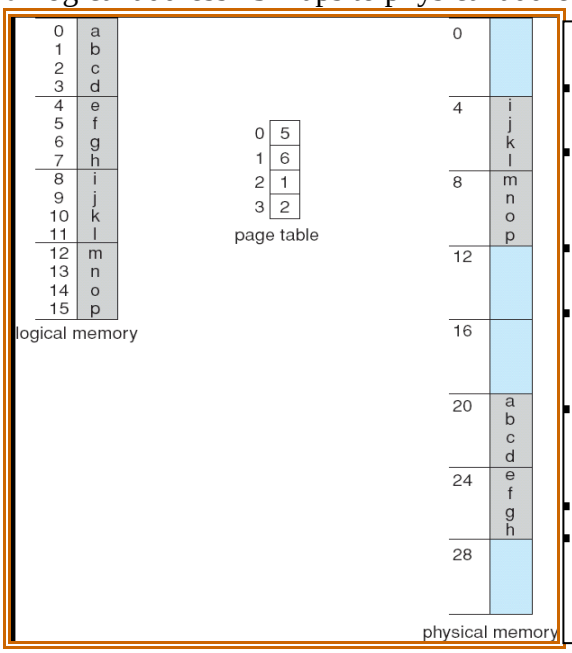
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.



- The hardware support for paging is illustrated in Fig.
  - Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d).
  - The page number is used as an index into a page table.
  - The page table contains the base address of each page in physical memory.
  - This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

Address Translation Architecture

- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. If the size of the logical address space is 2^m, and a page size is 2^n address unit (bytes or word), then the high-order m-n bits of a logical address designate the page number and n lower bit designate the page offset.
- Consider the memory in Fig. Using a page size of 4 bytes (2^n) with logical address space 16 bytes (2^m) generate 4 pages and a physical memory of 32 bytes (8 frames). It is shown that how the user's view of memory can be mapped into physical memory.
  - Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0).
  - Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3).
  - Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0).
  - Logical address 13 maps to physical address 9.



Paging Example

Advantages of paging

- **Address translation**: each task has the same virtual address
- **Address translation**: turns fragmented physical addresses into contiguous virtual addresses
- **Memory protection** (buggy or malicious tasks can't harm each other or the kernel)
- **Shared memory** between tasks (a fast type of IPC, also conserves memory when used for DLLs)
- **Demand loading** (prevents big load on CPU when a task first starts running, conserves memory)
- **Memory mapped files**
- **Virtual memory swapping** (lets system degrade gracefully when memory required exceeds RAM size

Disadvantage:

Paging cannot solve the problem of internal fragmentation. For example: Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example,

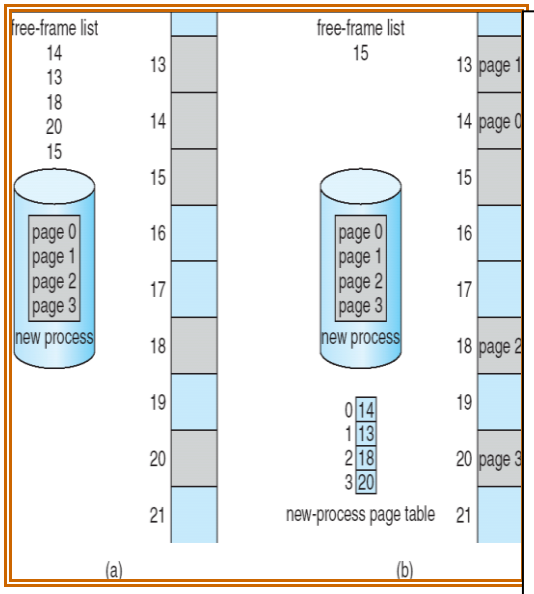
- If page size is 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes.
- It would be allocated 36 frames, resulting in an internal fragmentation of 2,048 - 1,086 = 962 bytes.
- In the worst case, a process would need n pages plus 1 byte. It would be allocated (n+ 1) frame, resulting in an internal fragmentation of almost an entire frame.

- If process size is independent of page size, we expect internal fragmentation to average one-half page per process.
- It is difficult share program/Data between processes.

One probable solution to minimize internal fragmentation and its disadvantage:

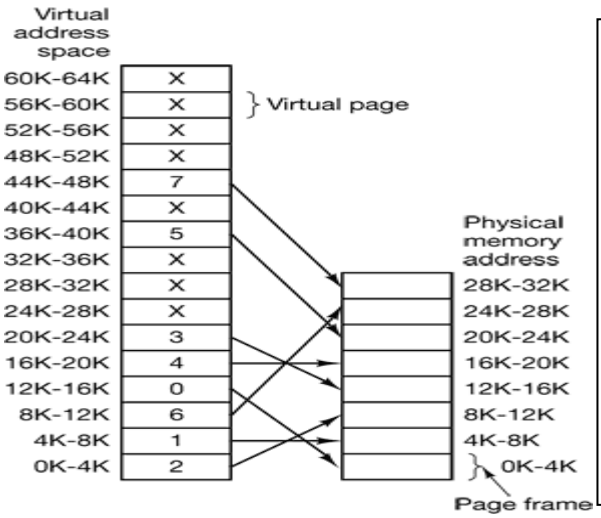
- This consideration suggests that small page sizes are desirable.
- However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the number of data being transferred is larger.
- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
- Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes.
- Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames.
- If frame size is 4 KB, then a system with 4-byte entries can address 2^{4}(4KB *2^{32}) bytes (or 16 TB) of physical memory

4.2 General process of page loading



Before allocation After allocation

- When a process arrives in the system to be executed,
  - Its size, expressed in pages, is examined. Each page of the process needs one frame.
  - Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process.
  - The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.
  - The next page is loaded into another frame, and its frame number is put into the page table, and so on



The pages and page frames are always the same size. In this example they are 4 KB, but page sizes from 512 bytes to 64 KB have been used in real systems.

For Example: With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames. Transfers between RAM and disk are always in units of a page.

Figure : The relation between virtual addresses and physical memory addresses is given by the page table.

When the program tries to access address 0, for example, using the instruction

MOV REG, 0

virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, an instruction

MOV REG, 8192

is effectively transformed into

MOV REG, 24576

because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only eight physical page frames, only eight of the virtual pages in Fig. 4-10 are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

What happens if the program tries to use an unmapped page, for example, by using the instruction

MOV REG, 32780

which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the page just

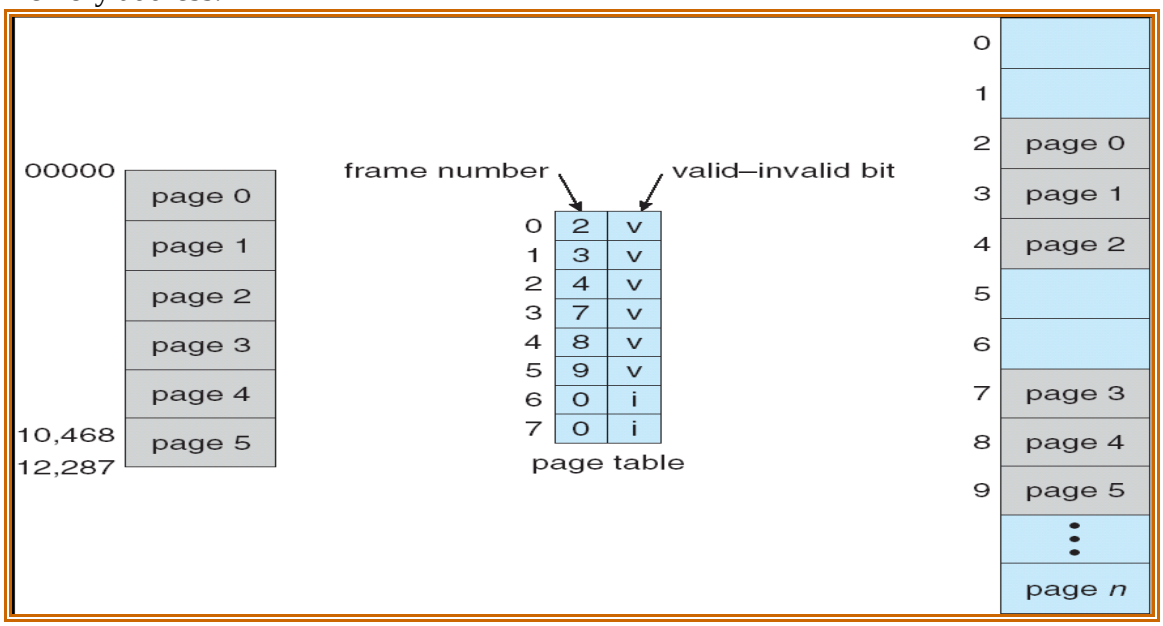
referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

For example, if the operating system decided to evict page frame 1, it would load virtual page 8 at physical address 4K and make two changes to the MMU map. First, it would mark virtual page 1's entry as unmapped, to trap any future accesses to virtual addresses between 4K and 8K, Then it would replace the cross in virtual page 8's entry with a 1, so that when the trapped instruction is re-executed, it will map virtual address 32780 onto physical address 4108.

Now let us lock inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Fig. we see an example of a virtual address, 8196 (0010000000000100 in binary), being mapped using the MMU map of Fig. 4-10. The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset. With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

4.3 Protection

The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page. If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order 3 bits of the output register, along with the 12-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

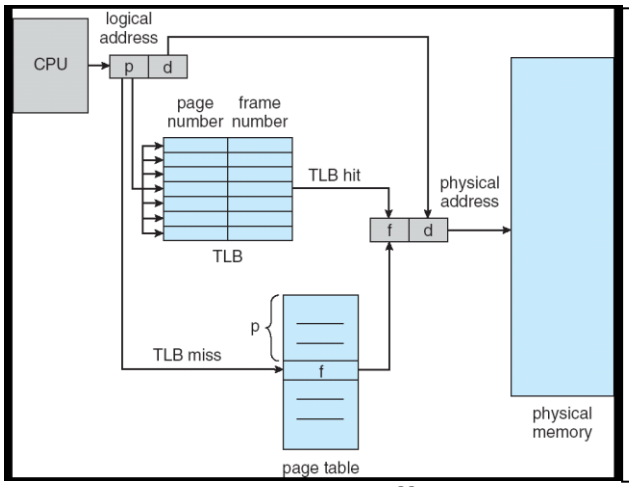


Valid (v or 1) or Invalid (I or 0) Bit in a Page Table

- Memory protection in a paged environment is accomplished by **protection bits** associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.

- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table: 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 (or valid) page.
  - When the bit is set to "invalid", the page is not in the process's logical address space.
- Illegal addresses are trapped by use of the valid-invalid bit. The OS sets this bit for each page to allow or disallow access to the page.

4.4 Paging hardware with TLB



- The percentage of times that a particular page number is found in the TLB is called **hit ratio**
- If it takes 20 nanosecond to search the TLB and 100 nanosecond to access memory
- If our hit ratio is 80%, the effective memory access time equal:

0.8*(100+20) + 0.2 *(100+100+20)=140

- If our hit ratio is 98%, the effective memory access time equal:

0.98*(100+20) + 0.02 *(100+100+20)=122

A **translation lookaside buffer** (**TLB**) is a CPU cache that memory management hardware uses to improve virtual address translation speed. All current desktop and server processors (such as x86) use a TLB to map virtual and physical address spaces, and it is ubiquitous in any hardware which utilizes virtual memory. A TLB has a fixed number of slots that contain page table entries, which map virtual addresses to physical addresses. The TLB references physical memory addresses in its table. It may reside between the CPU and the CPU cache, between the CPU cache and primary storage memory, or between levels of a multi-level cache. The placement determines whether the cache uses physical or virtual addressing. A common optimization for physically addressed caches is to perform the TLB lookup in parallel with the cache access.
With hardware TLB management, the CPU itself walks the page to see if there is a valid page table entry for the specified virtual address. If an entry exists, it is brought into the TLB and the TLB access is retried: this time the access will hit, and the program can proceed normally. If the CPU finds no valid entry for the virtual address in the page tables, it raises a page fault exception, which the operating system must handle. Handling page faults usually involves bringing the requested data into physical memory, setting up a page table entry to map the faulting virtual address to the correct physical address, and resuming the program.

5. Segmentation

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Users prefer to view memory as a

collection of variable-sized segments, with no necessary ordering among segments. When we are writing a program we think it as a main program with a set of methods, procedures, or functions. It also includes various data structure: objects, arrays, stacks, variables, and so on. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.

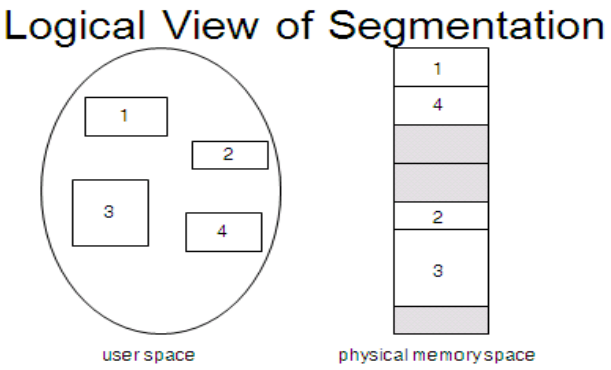
The user therefore specifies each address by two quantities:

- a segment name
- an offset

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.

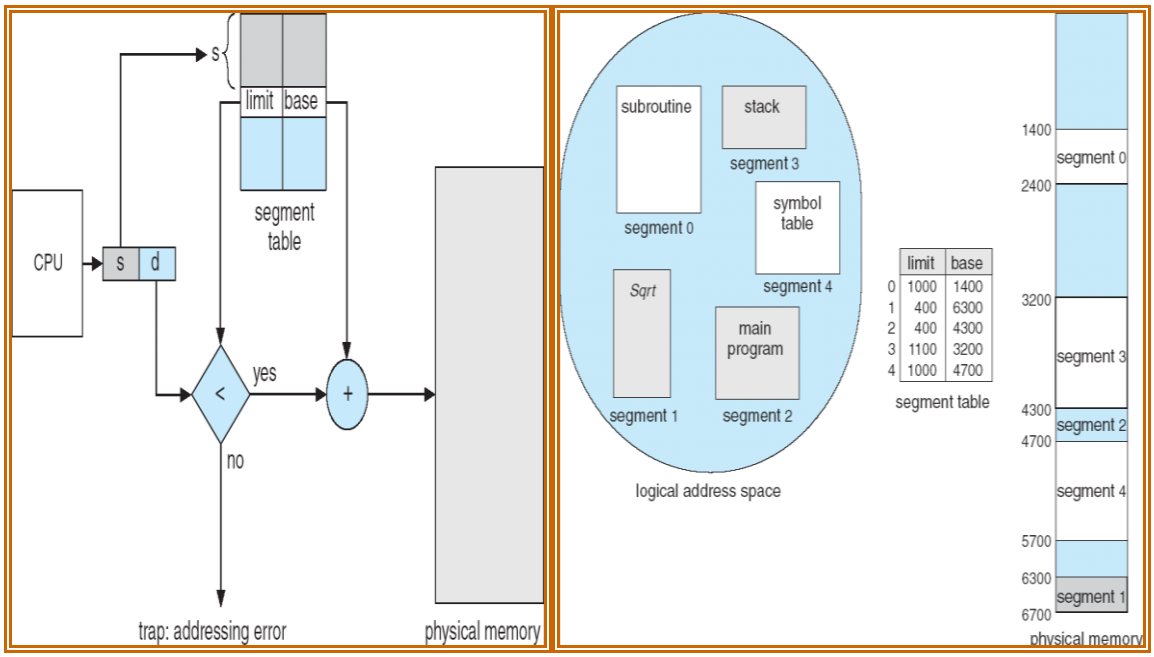
Thus, a logical address consists of a two tuple:

<segment-number, offset>



5.1 Hardware

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is affected by a **segment table**. Each entry in the segment table has a segment base and a segment limit.
  - The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment A logical address consists of two parts: a segment number, S, and an offset into that segment, d.
  - The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the OS (logical addressing attempt beyond end of segment).
  - When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.



Address Translation Architecture **Example of Segmentation**

We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment would result in a trap to the OS, as this segment is only 1,000 bytes long.

Difference between Segmentation and Paging

Paging	Segmentation
Computer memory is divided into small partitions that are all the same size and referred to as, page frames. Then when a process is loaded it gets divided into pages which are the same size as those previous frames. The process pages are then loaded into the frames.	Memory-management scheme that supports user view of memory. Computer memory is allocated in various sizes (segments) depending on the need for address space by the process.
Address generated by CPU is divided into: *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory. *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.	Logical address consists of a two tuple: <segment-number, offset>
Transparent to programmer (system allocates memory)	Involves programmer (allocates memory to specific function inside code)
No separate protection	Separate protection
No separate compiling	Separate compiling
No shared code	Share code

Advantage Segmentation

- Logical separation is performed along procedure/data structure boundaries blocks are of different sizes

- Protection is more natural at the logical level such as sharing of common procedures (Segmentation) Vs. sharing of fixed pages (Paging)

Drawbacks of segmentation

- Segment sizes can become enormous (leads to slow I/O transfer)
- More difficult for memory management
- Need to search for the proper hole in the main memory
- Management of free space in main memory
- External fragmentation (paging has the problem of internal fragmentation)