MEMORY MANAGEMENT

LOGICAL VS PHYSICAL ADDRESS SPACE

An address generated by the CPU is referred to as logical address. An address seen by the memory unit i.e. the one loaded into the memory address register of the memory is referred to as the physical address. The compile time and load time address binding methods generate identical logical and physical addresses. The execution time address binding scheme results in differing logical and physical addresses. Here, we refer to logical address as virtual address. The set of all logical addresses generated by the program is called logical address space and the set of all physical addresses corresponding to these logical addresses is called physical address space. A hardware device called the Memory Management Unit (MMU) does the run time mapping from virtual address to physical addresses. The user program supplies the logical addresses, these logical addresses must be mapped into physical addresses before they are used.

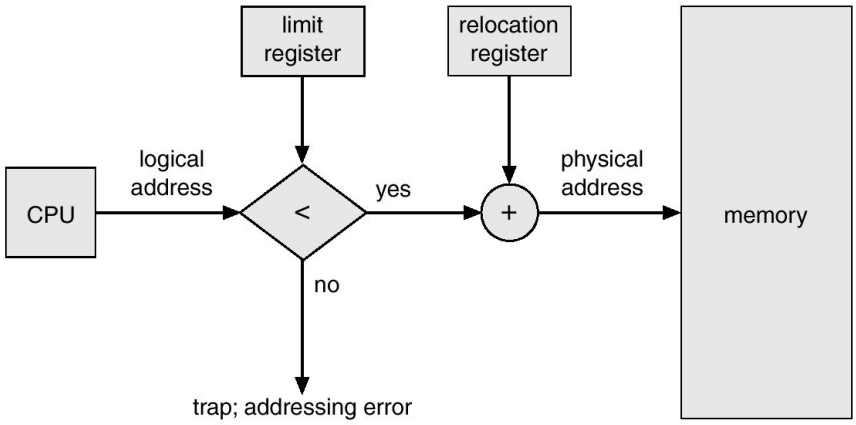

Figure: Dynamic relocation using relocation register

The base register is called as 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 MSDOS operating system running the Intel 80x 86 families of processors uses four relocation registers when loading and running processes.

SWAPPING

A process needs to be in memory to be executed. A process can be swapped temporarily out of memory to a backing store and then brought back to memory for continued execution. For example: assume a multi programmed environment with round robin CPU

scheduling. When a quantum expires the memory manager will start to swap out the process that just finished and swap in another process to the memory space that has just been freed. In the mean time the CPU will allocate time slice to some other process in the memory. A variant of this policy is used in priority based scheduling algorithm. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute high priority process. When a higher priority process finishes, the lower priority process can be swapped in and continued. This variant of swapping is called roll-out roll-in.

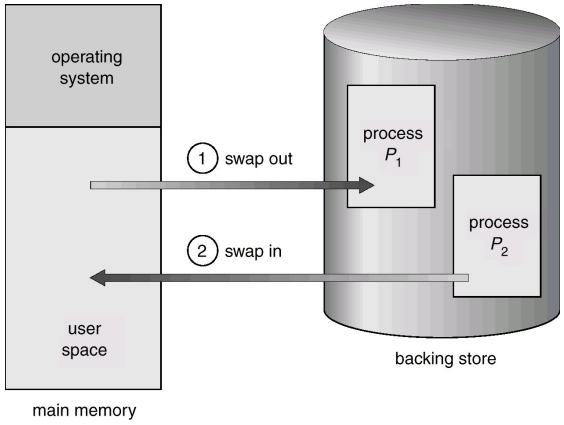

Figure: swapping of two processes using a disk as backing store

Normally, a process that is swapped out will be swapped back into the same memory space that it occupied previously. If the binding is done at assembly/ load time, the process cannot be moved to different locations. If execution time binding is being used, then a process can be swapped into different memory space because physical addresses are computed during execution time. Swapping requires a backing store. Backing store is usually a fast disk. It must be large enough to accommodate copies of all memory images for all users. It must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If there is no free memory region the dispatcher swaps out a process currently in memory and swaps in a desired process. It then reloads the registers as normal and transfers control to the selected process.

Limitations:

- Context switch time is high.
- Major part of swap time is transfer time.

CONTIGUOUS MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user processes. Hence, we need to allocate different parts of main memory in an efficient way. The memory is divided into two parts: one for resident operating system and one for the user processes. We may place the operating system either at low memory or high memory. Since the interrupt vector is in the low memory the operating system is also placed in the low memory. Contiguous memory allocation deals with each process contained in the single contiguous section of memory.

MEMORY ALLOCATION

One simple approach is to divide the memory into fixed sized partitions. Each partition may contain exactly one process. The degree of multiprogramming is bounded by the number of partitions. In 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 free for another process. The operating system keeps a table indicating which parts of the memory are available and which are occupied. Initially all memory is available for user processes and is considers as one large block of available memory - a hole. When a process arrives and needs memory, we search for a hole that is large enough for the process. If we find one, we allocate only as much memory as needed, keeping the rest to satisfy future requirements. As the processes enter the system, they are put in an input queue.

The operating system takes into account the memory requirement of each process and the amount of memory space available. When a process is allocated space, it is loaded into memory and it can then, compete for CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue. At any given point of time, we have a list of block sizes and input queue.

The operating system can order the input queue according to any scheduling algorithm. Memory is allocated to processes until the memory requirements of the next process cannot be satisfied or no available block of memory is large enough to hold that process. The operating system can then wait until a large block that is big enough is available or it can skip down the input queue to see whether smaller memory requirement processes can be met. A set of holes of various sizes is scattered in the memory at any given point of time.

When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: one part is allocated to the arriving process and the other is returned to the set of holes. If a new hole is adjacent to other holes these adjacent holes are merged to form a larger hole. At this point of time, the system needs to check whether there are processes waiting for memory and whether their requirements can be satisfied. This is an instance of dynamic storage allocation problem - how to satisfy a request of size n from a list of free holes. The set of holes is searched to determine which of the hole is best to allocate. The most common strategies are

- **First fit -** Allocate the hole that is large enough. Searching can start at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching once 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 kept ordered by size. This strategy produces smallest leftover holes.
- **Worst fit -** Allocate the largest hole. We must search the entire list. This strategy produces the largest leftover holes which may be more useful than the smaller leftover holes from the best fit approach.

FRAGMENTATION

Memory fragmentation could be internal as well as external. . As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough total memory space exists to satisfy a request but it is not contiguous. Storage is fragmented into a large number of small holes. Physical memory is broken into fixed sized blocks and allocated memory in unit of block sizes. In this approach, memory allocated to a process will be slightly larger than the requested memory. The difference in memory space between the allocated and requested memory is called internal fragmentation - memory that is internal to a partition but not being used. One solution to the problem of external fragmentation is compaction. The goal here is to shuffle the memory contents to place all the free memory together in one large block. Compaction is possible only if the relocation is dynamic. The simplest compaction algorithm is simply to move all processes to one end of memory and all holes in the other direction, producing one large hole of available memory.

Another solution is to permit logical address space of a process to be non-contiguous thus allowing a process to be allocated physical memory where it is available. The two approaches to implement this strategy are paging and segmentation.

PAGING

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids problems of fitting varying-sized memory chunks on to the backing store.

Basic Method

Physical memory is divided into fixed-sized blocks called frames. Logical memory is broken into blocks of same size called pages. 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 memory frames. Every address that is 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 to the page table. The page table contains the base address of each page in the physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

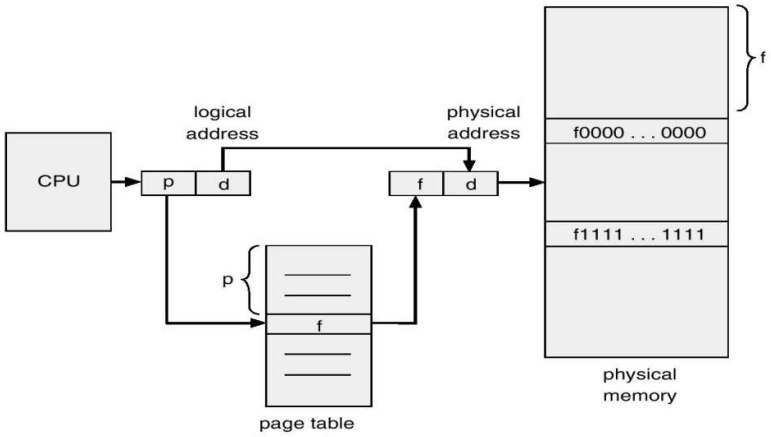

Figure: paging Hardware

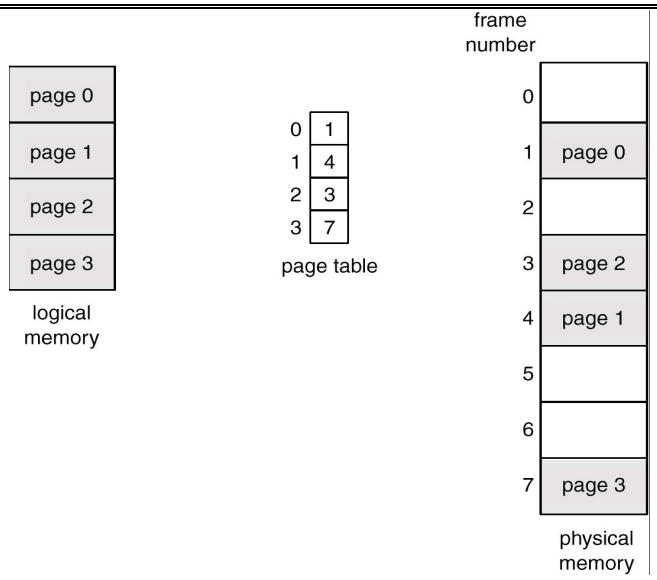

Figure: Paging model of logical and physical memory

The page size is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 64 MB/page depending on the architecture. The selection of power of 2 as page size makes it easy to translate the logical address space to page number and offset. If the size of the logical address space is 2mand page size is 2n then the higher order mn bits of logical address designate page number and the lower order bits designate the page offset.

Thus the logical address is as follows:

Page number	page offset
P	d
m-n	n

Where, p - index to page table

d - displacement within the page.

Consider an example for 32 byte memory with 4 byte pages. Using page size as 4 bytes and the physical memory of 32 bytes (8 pages), illustrate how user view of memory can be mapped into physical memory.

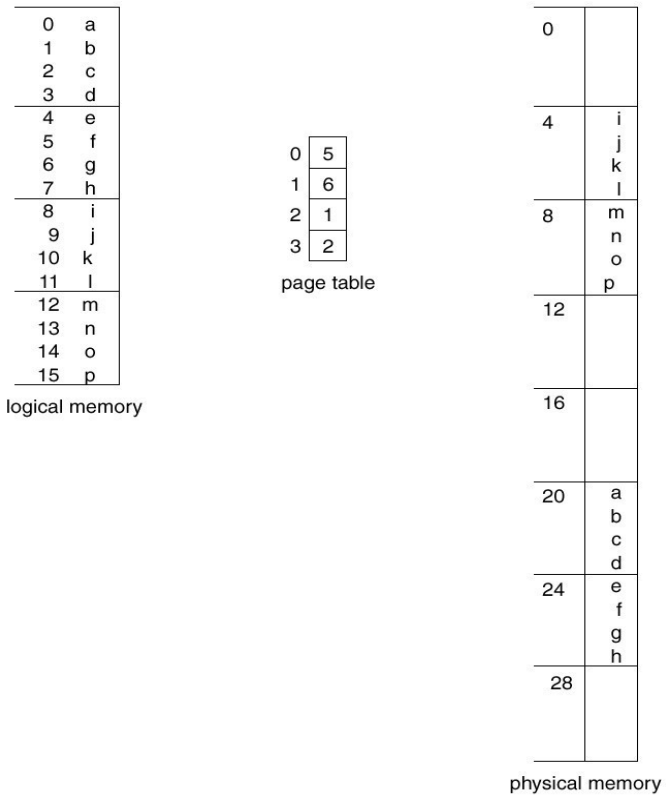

Figure: paging example for a 32-byte memory with 4 byte pages

Logical address 0 is page 0 offset O. Indexing into page table, we find that page 0 is in frame 5, Thus logical address 0 maps to physical address (5x4)+0=20. Logical address 3(page 0, offset 3) maps to physical address (5x4)+3=23. Logical address 4 is page 1 offset O. According to the page table, page 1 is mapped to frame 6. Logical address 4 maps on to physical address (6x4)+0=24. Paging is itself a form of dynamic relocation. Every logical address is bound by paging hardware to some physical address. Using paging, we have no external fragmentation. Any free frame can be allocated to a process that needs it. However, internal fragmentation may occur. If the memory requirement of a process does not happen to fall on page boundaries, the last fame allocated may need not be completely full.

For example, if each page can hold 2048 bytes and the process requires 72766 bytes, it would need 35 pages plus 1086 bytes. It would be allocating 36 frames, resulting in an internal fragmentation of 2048-1 086=962 bytes. When a process enters a system to be executed. Its size in terms of pages is examined.

Each page of a process needs one frame. Thus, if a process requires n pages, at least n frames must be available in memory. If n frames are available in memory, 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 put in the page table for this process. An important aspect of paging is the clear separation between the user memory and the actual physical memory. The user program views memory as a single contiguous space containing only this single program. But, the user's program is scattered throughout the physical memory that also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address translation hardware. The logical addresses are translated into the physical address. This aping is hidden from the user and is controlled by the operating system.

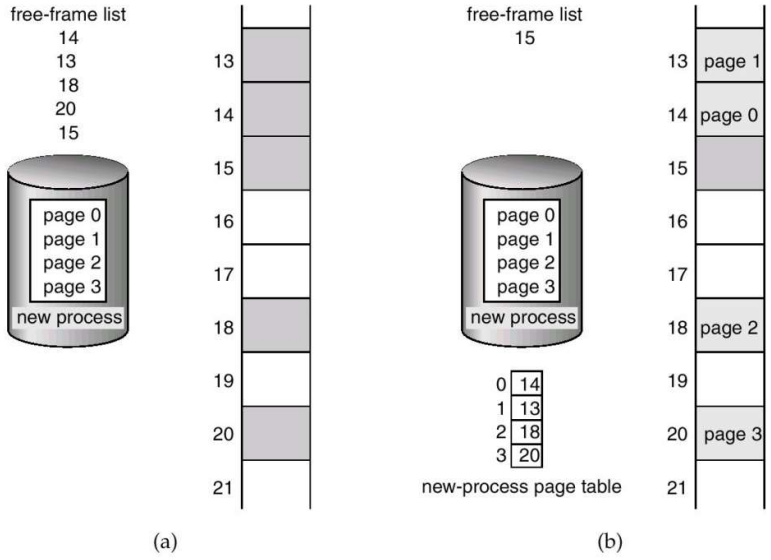

Figure: free frames (a) before allocation (b) after allocation

Since the operating system is managing the physical memory, it must be aware of the allocation details of the physical memory - which frames are available, which frames are allocated. Total frames and so on. This information is kept in a data structure called the frame table. The frame table has one entry per for each physical page frame indicating whether the latter is free or allocated. If it is allocated, to which page of which process, the operating system also maintains a copy of page table for each process in order to translate logical address to physical address. Paging increases the context switch time.

SEGMENTATION

Segmentation is one of the most common ways to achieve memory protection. In a computer system using segmentation, an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a

set of permissions, and a length, associated with it. If the currently running process is allowed by the permissions to make the type of reference to memory that it is attempting to make, and the offset within the segment is within the range specified by the length of the segment, the reference is permitted; otherwise, a hardware exception is raised.

In addition to the set of permissions and length, a segment also has associated with it information indicating where the segment is located in memory. It may also have a flag indicating whether the segment is present in main memory or not; if the segment is not present in main memory, an exception is raised, and the operating system will read the segment into memory from secondary storage. The information indicating where the segment is located in memory might be the address of the first location in the segment, or might be the address of a page table for the segment, if the segmentation is implemented with paging. In the first case, if a reference to a location within a segment is made, the offset within the segment will be added to address of the first location in the segment to give the address in memory of the referred-to item; in the second case, the offset of the segment is translated to a memory address using the page table.

In most systems in which a segment doesn't have a page table associated with it, the address of the first location in the segment is an address in main memory; in those systems, no paging is done. In the Intel 80386 and later, that address can either be an address in main memory, if paging is not enabled, or an address in a paged "linear" address space, if paging is enabled. A memory management unit (MMU) is responsible for translating a segment and offset within that segment into a memory address, and for performing checks to make sure the translation can be done and that the reference to that segment and offset is permitted.

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

Although the programmer 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 effected 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,

and the segment limit specifies the length of the segment. The use of a segment table is illustrated in below figure

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 operating system (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.

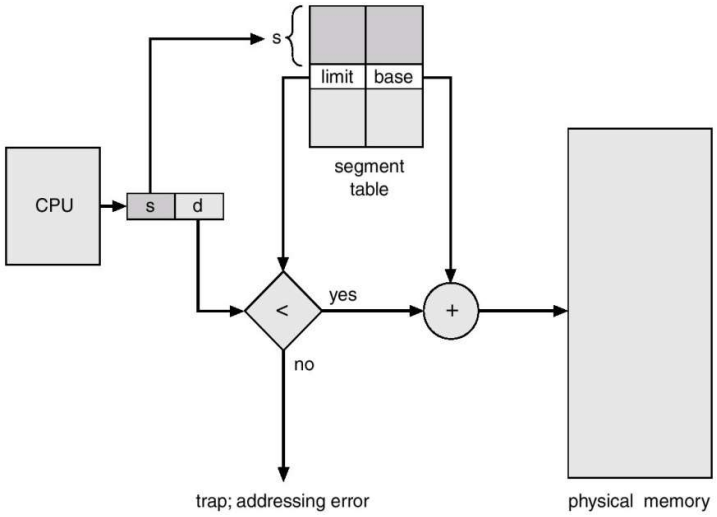

Figure: Segmentation Hardware