

## UNIT - III    STORAGE MANAGEMENT

### Main Memory

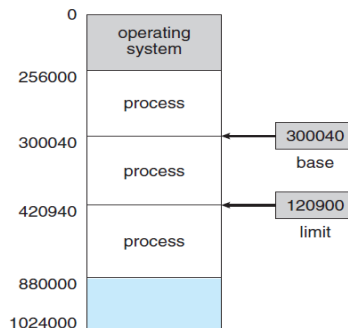
#### Background

- 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.
- 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.

#### Basic Hardware

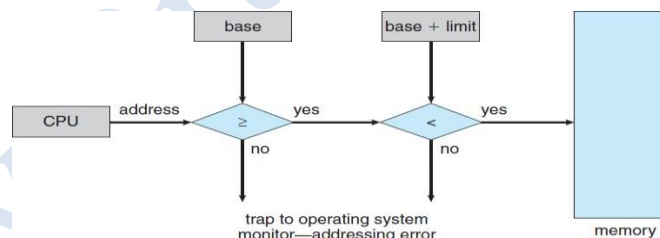
- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- Therefore, any instructions in execution, and any data being used by the 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.

**A base and limit register defines a logical address space**



- 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.
  - 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.
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.

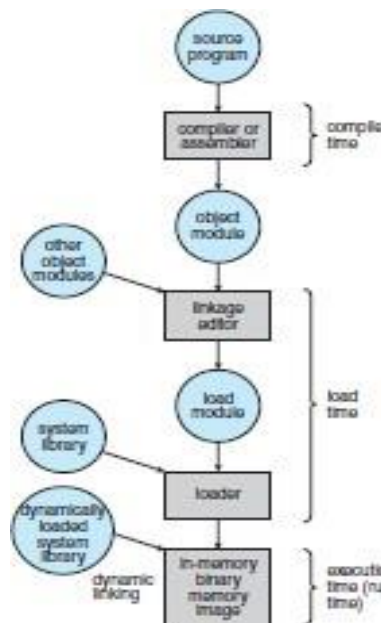
### Hardware address protection with base and limit registers



### 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

## Multistep processing of a user program



Addresses may be represented in different ways during these steps.

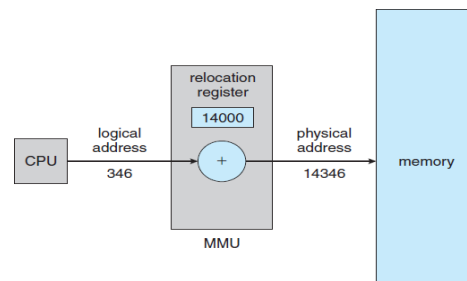
- Addresses in the source program are generally symbolic
- A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module").
- The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014).
- 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.

## 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 a relocation register.



- The Base register is 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.

#### Dynamic Loading

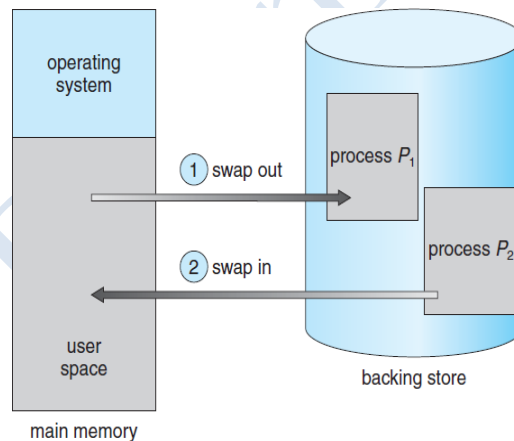
- To obtain better memory-space utilization, we can use dynamic loading.
  - With dynamic loading, a routine is not loaded until it is called.
  - All routines are kept on disk in a relocatable load format.
  - The main program is loaded into memory and is executed. When a routine needs to call another 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 Linking and Shared Libraries

- The concept of dynamic linking is similar to that of dynamic loading.
- Here, though, linking, rather than loading, is postponed until execution time. With dynamic linking, a stub is included in the image for each library-routine reference.
- The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
  - When the stub is executed, it checks to see whether the needed routine is already in memory.
  - If not, the program loads the routine into memory.
- This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.

## 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.



## Swapping

- A round-robin CPU-scheduling algorithm; when a quantum expires
  - 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. Backing store is usually a hard disk drive or any other secondary storage which is fast in access and large enough to accommodate copies of all memory images for all users.
- It must be capable of providing direct access to these memory images.

- Major time consuming part of swapping is transfer time.
- Total transfer time is directly proportional to the amount of memory swapped.

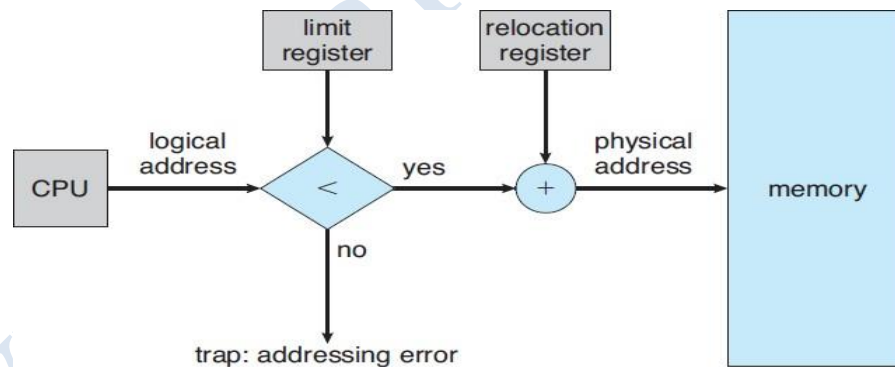
### Contiguous Memory Allocation

- The memory is usually divided into two partitions:
  - one for the resident OS
  - one for the user processes.
- We can place the OS in either low memory or high memory (depends on the location of the interrupt vector).
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

### Memory Mapping and Protection

- With relocation and limit registers, each logical address must be less than the limit register;
- The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

#### Hardware support for relocation and limit registers.



- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

### 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.

### 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 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.

### 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 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.

### Fragmentation

- 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.
- **Internal Fragmentation** Memory block assigned to process is bigger than requested. 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.



- 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.
- 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.

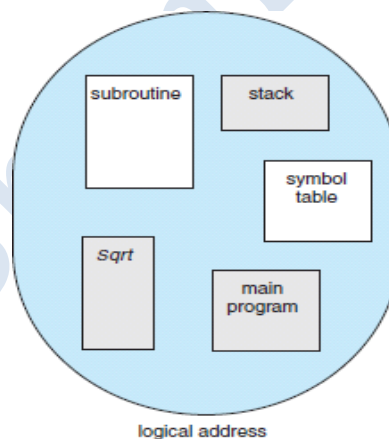
### Segmentation

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory.
- The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory.

### Basic Method

- Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments

### Programmer's view of memory



- Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions.
- Segmentation is a memory-management scheme that supports this user 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 user therefore specifies each address by two quantities:
  - a segment name
  - an offset

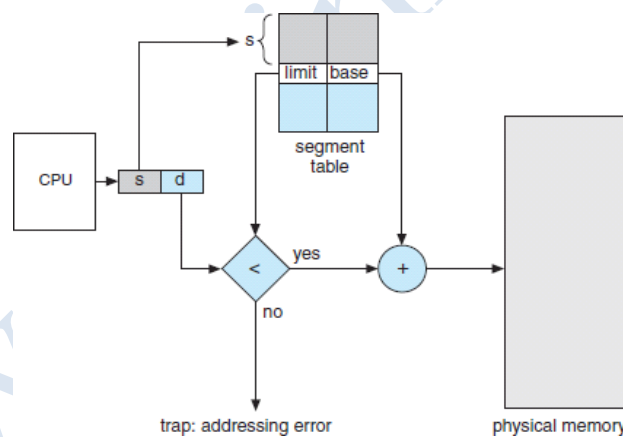


- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a two tuple:
- $\langle \text{segment-number, offset} \rangle$

## Hardware

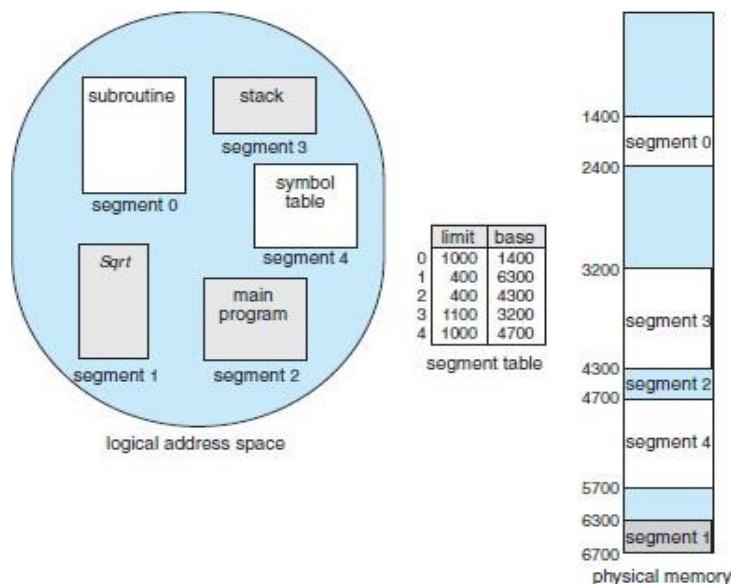
- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is 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, whereas the segment limit specifies the length of the segment

### Segmentation Hardware



- 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.

### 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.

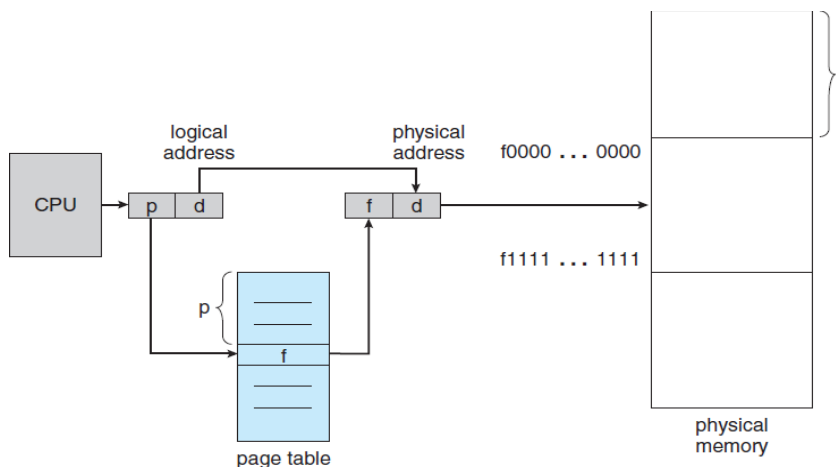
### Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.

#### **Basic Method**

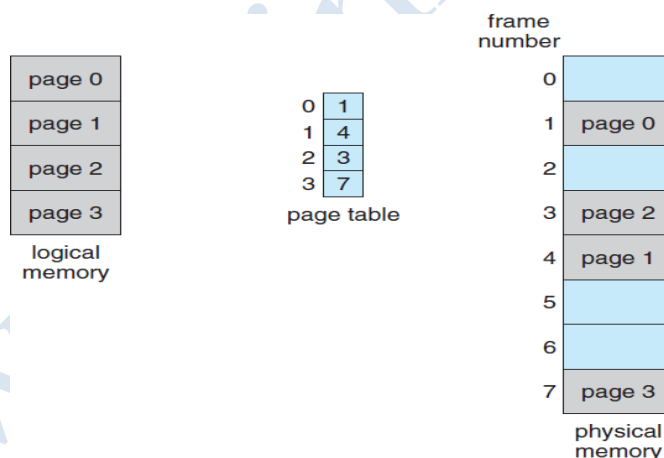
- The basic method for implementing paging involves
  - breaking physical memory into fixed-sized blocks called frames
  - breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

### **Paging Hardware**



Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).

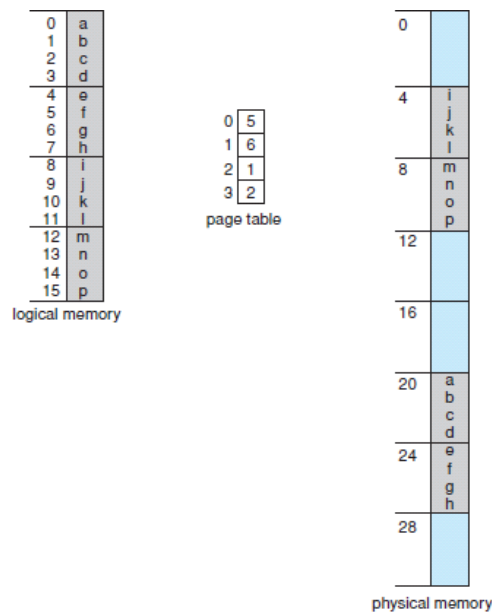
- The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



**Paging model of logical and physical memory.**

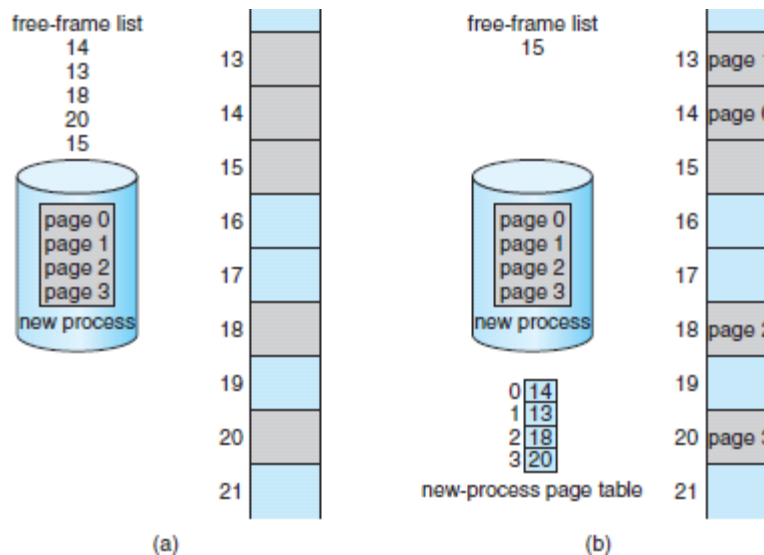
- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

**Paging example for a 32-byte memory with 4-byte pages**



- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages). 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 ( $= (5 \times 4) + 0$ ).
  - Logical address 3 (page 0, offset 3) maps to physical address 23 ( $= (5 \times 4) + 3$ ).
  - 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 ( $= (6 \times 4) + 0$ ).
  - Logical address 13 maps to physical address 9.
- 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

Free frames (a) before allocation and (b) after allocation.



- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.
- The user program views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.
- The logical addresses are translated into physical addresses by the address-translation hardware. This mapping is hidden from the user and is controlled by the OS.
- The user process has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
- Since the OS is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

## Hardware Support

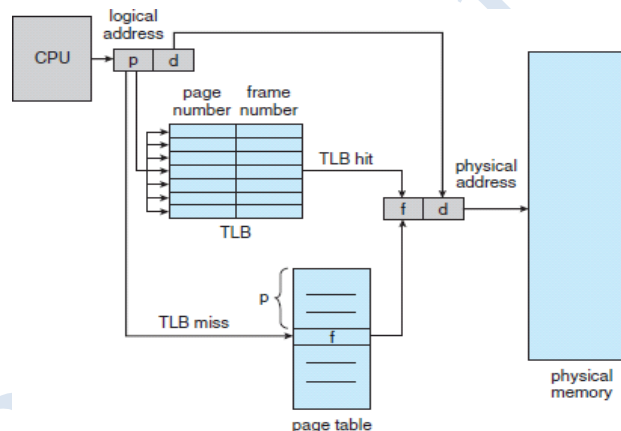
### Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

## TLB

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts:
  - a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made.
- Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.
- If the page number is found, its frame number is immediately available and is used to access memory.

### Paging Hardware with TLB



**Hit Ratio** - The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

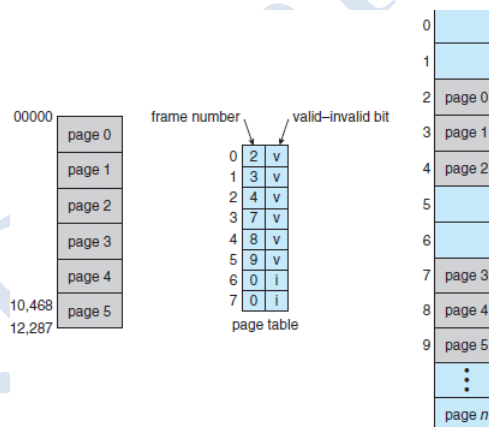
### Effective Memory Access Time

- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds.
$$\begin{aligned}\text{effective access time} &= 0.80 \times 100 + 0.20 \times 200 \\ &= 120 \text{ nanoseconds}\end{aligned}$$
- For a 99-percent hit ratio, which is much more realistic, we have
$$\text{effective access time} = 0.99 \times 100 + 0.01 \times 200 = 101 \text{ nanoseconds}$$

## Protection

- 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.

### Valid (v) or invalid (i) bit in a page table.

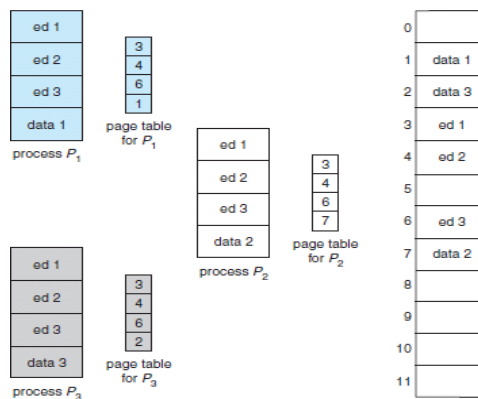


- Item Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS

## Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment
  - **Reentrant code** is non-self-modifying code; it never changes during execution. Thus, two or more processes can execute the same code at the same time.





### Sharing of code in a paging environment

- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.
- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

### Structure of the Page Table

The most common techniques for structuring the page table are

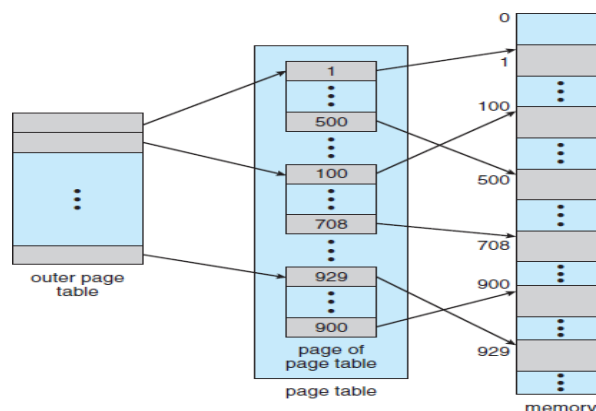
Hierarchical paging  
Hashed page tables  
Inverted page tables.

### Hierarchical Paging

The page table itself becomes large for computers with large logical address space (232 to 264).

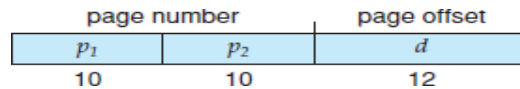
Example:

- Consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- The page table should be allocated contiguously in main memory.
- The solution to this problem is to divide the page table into smaller pieces.
- One way of dividing the page table is to use a two-level paging algorithm, in which the page table itself is also paged as in the figure:



For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

Thus, a logical address is as follows:

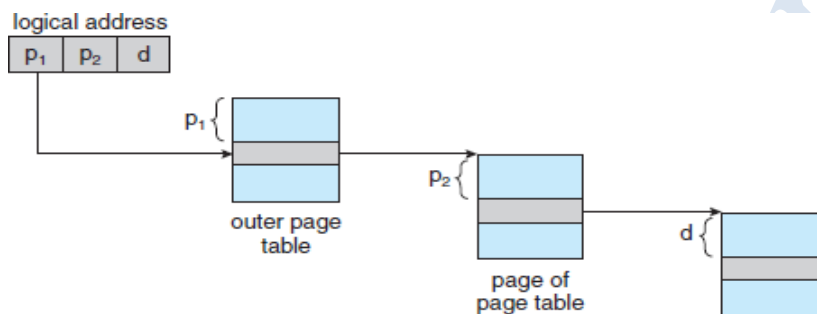


where

$p_1$  - an index into the outer page table

$p_2$  - the displacement within the page of the inner page table.

The address-translation method for this architecture is shown in the figure. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.



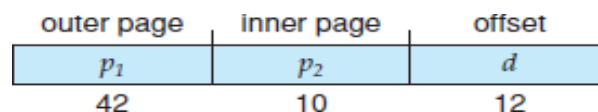
### Address translation for a two-level 32-bit paging architecture

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.

- Suppose that the page size in such a system is 4 KB (212).

- In this case, the page table consists of up to 252 entries.
- If a two-level paging scheme is used, then the inner page tables can conveniently be one page long, or contain 210 4-byte entries.

The addresses look like this:



The outer page table consists of 242 entries, or 244 bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.

The outer page table can be divided in various ways.

- we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (210 entries, or 212 bytes).

In this case, a 64-bit address space is as follows :

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

The outer page table is still 234 bytes (16 GB) in size.

- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—to translate each logical address.

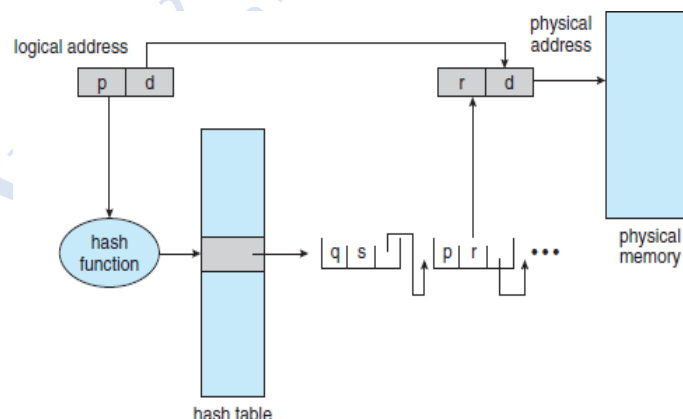
### Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).
- Each element consists of three fields:
  - the virtual page number
  - the value of the mapped page frame
  - a pointer to the next element in the linked list.

### Algorithm:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

This scheme is shown below:



- A variation of this scheme that is useful for 64-bit address spaces has been proposed.
- This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page

frames.

- Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

### Inverted Page Tables

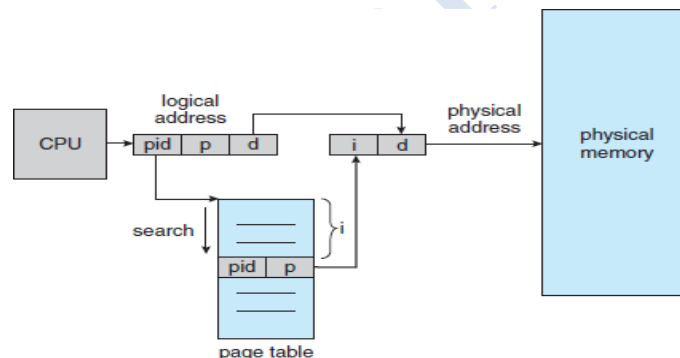
- Each process has an associated page table.
- The page table has one entry for each page that the process is using. This table representation is a natural one, since processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address.
- Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly.

### Drawbacks of this method

- Each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used. To solve this problem, we can use an **inverted page table**.

An inverted page table has one entry for each real page (or frame) of memory.

- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page.
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory.



- The operation of an inverted page table is shown above:
- Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.
- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, a simplified version of the inverted page table used in the **IBM RT** is described. For the IBM RT, each virtual address in the system consists of a triple:

**<process-id, page-number, offset>.**

where

process-id - the role of the address-space identifier.

- When a memory reference occurs, part of the virtual address, consisting of <process-id, pagenumber>, is presented to the memory subsystem.

- The inverted page table is then searched for a match.
- If a match is found—say, at entry  $i$ —then the physical address  $\langle i, \text{offset} \rangle$  is generated.
- If no match is found, then an illegal address access has been attempted.
- Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.
- Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long.
- To alleviate this problem, a hash table is used, to limit the search to one—or at most a few—page-table entries.
- Each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table.
- Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses that are mapped to one physical address.
- This standard method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses.
- A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address. This means that references to virtual addresses that are not mapped result in page faults.

### Example: Intel 32 and 64-bit Architectures

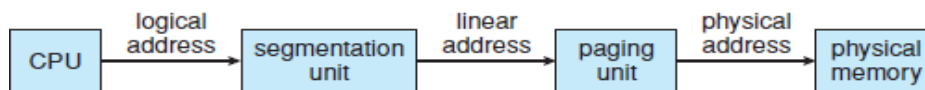
- Intel later produced a series of 32-bit chips —the IA-32—which included the family of 32-bit Pentium processors. The IA-32 architecture supported both paging and segmentation.
- More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture.
- Currently, all the most popular PC operating systems run on Intel chips, including Windows, Mac OS X, and Linux .

#### **IA-32 Architecture:**

- Memory management in IA-32 systems is divided into two components — segmentation and paging.

It works as follows:

- The CPU generates logical addresses, which are given to the segmentation unit.
- The segmentation unit produces a linear address for each logical address.
- The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
- Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU).
- This scheme is shown in Figure.



**Logical to physical address translation in IA-32.**

#### **IA-32 Segmentation**

- The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K.
- The logical address space of a process is divided into two partitions.
  - The first partition consists of up to 8 K segments that are private to that process.
  - The second partition consists of up to 8 K segments that are shared among all the processes.
- Information about the first partition is kept in the **local descriptor table (LDT)**;
- Information about the second partition is kept in the **global descriptor table (GDT)**.
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.
- The logical address is a pair (selector, offset), where the selector is a 16-bit number:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

where, *s* - the segment number,

*g* - whether the segment is in the GDT or LDT, and

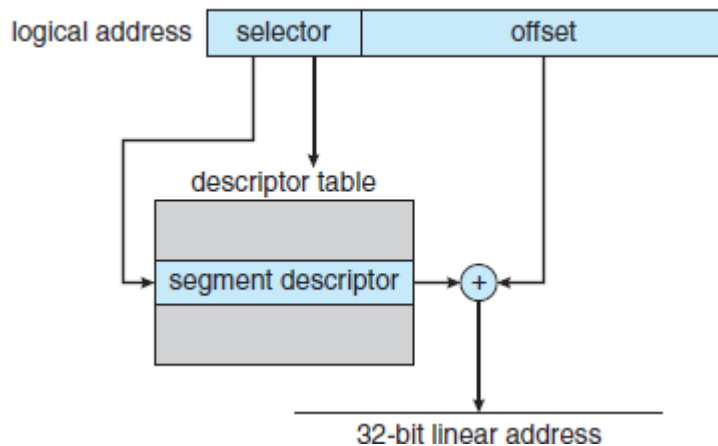
*p* - protection.

The offset is a 32-bit number specifying the location of the byte within the segment in question.

- The machine has six segment registers, allowing six segments to be addressed at any one time by a process.
- It also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT.
- This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

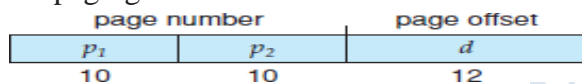
The linear address on the IA-32 is 32 bits long and is formed as follows.

- The segment register points to the appropriate entry in the LDT or GDT.
- The base and limit information about the segment in question is used to generate a **linear address**.
- First, the limit is used to check for address validity.
- If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address.
- This is shown in Figure .

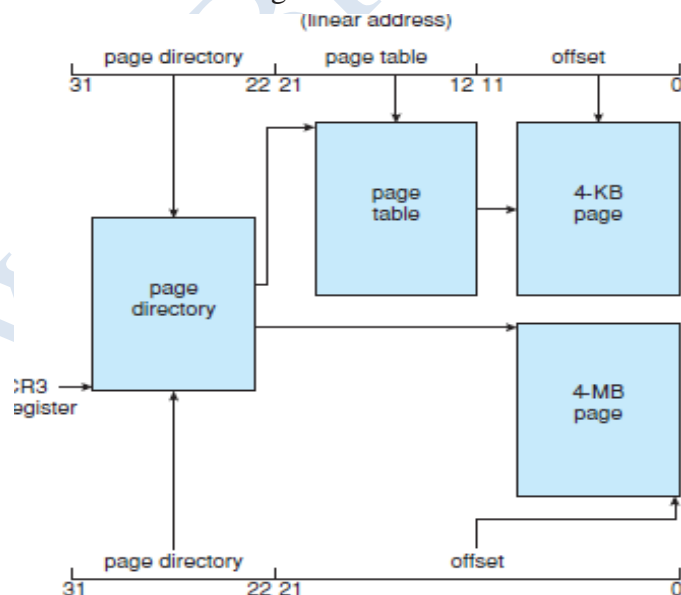


### IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



The address-translation scheme for this architecture is similar to the scheme shown in Figure. The IA-32 address translation is shown in Figure.

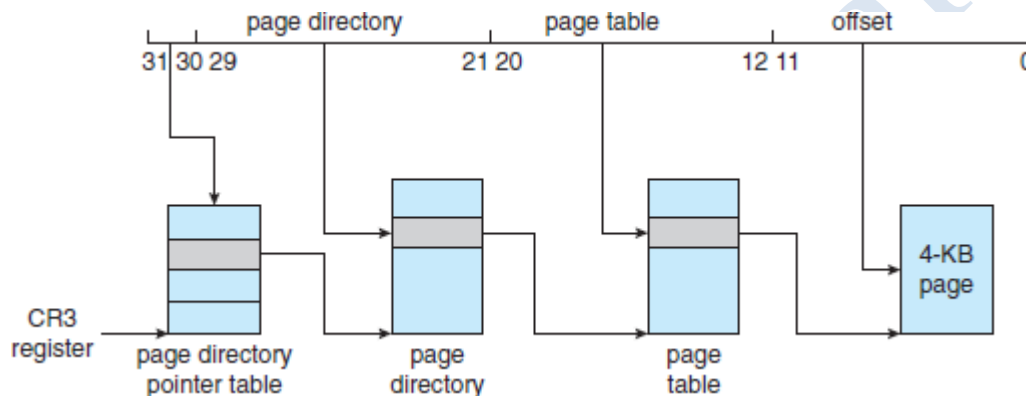


- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**. (The CR3 register points to the page directory for the current process.)



- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.
- One entry in the page directory is the Page Size flag, which—if set— indicates that the size of the page frame is 4 MB and not the standard 4 KB.
- If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.
- To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk.
- In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk..
- If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table. The table can then be brought into memory on demand.

Intel adopted a **page address extension (PAE)**, which allows 32-bit processors to access a physical address space larger than 4 GB. The fundamental difference introduced by PAE support was that paging went from a two-level scheme to a three-level scheme, where the top two bits refer to a **page directory pointer table**. Figure illustrates a PAE system with 4-KB pages. (PAE also supports 2-MB pages.)



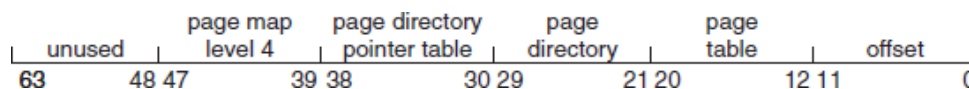
**Page address extensions**

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits. Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory.

### x86-64

- The initial entry of Intel developing 64-bit architectures was the IA-64 (later named **Itanium**) architecture, but was not widely adopted.
- Meanwhile, AMD —began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set.
- The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
- Support for a 64-bit address space yields an astonishing 264 bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes).
- However, even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs.
- The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes

of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy.



- The representation of the linear address appears in Figure

### **x86-64 linear address**

Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4096 terabytes).

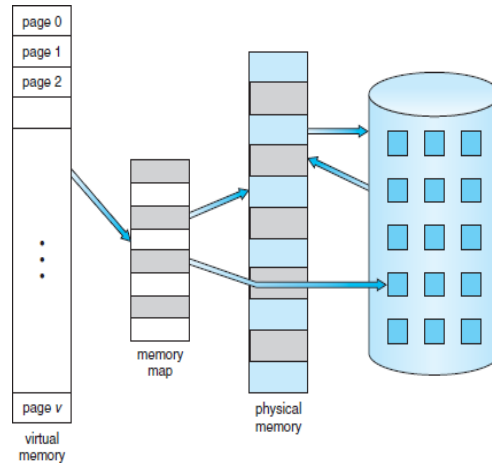
### **Virtual Memory**

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
  - One major advantage of this scheme is that programs can be larger than physical memory.
  - Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- Virtual memory also allows processes to share files easily and to implement shared memory.
- Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

### **Background**

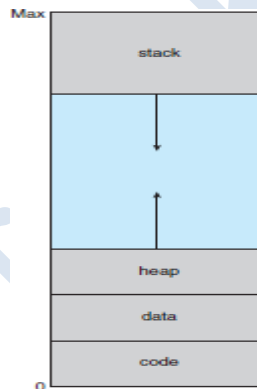
- The instructions being executed must be in physical memory.
- An examination of real programs shows us that, in many cases, the entire program (in memory) is not needed.
  - Programs often have code to handle unusual error conditions (seldom used).
  - Arrays, lists, and tables are often allocated more memory than they actually need.
  - Certain options and features of a program may be used rarely.
- The ability to execute a program that is only partially in memory would offer many benefits:
  - A program would no longer be constrained by the amount of physical memory that is available (simplifying the programming task).
  - Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
  - Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.
- Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available

### Diagram showing virtual memory that is larger than physical memory



- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address 0 -and exists in contiguous memory.

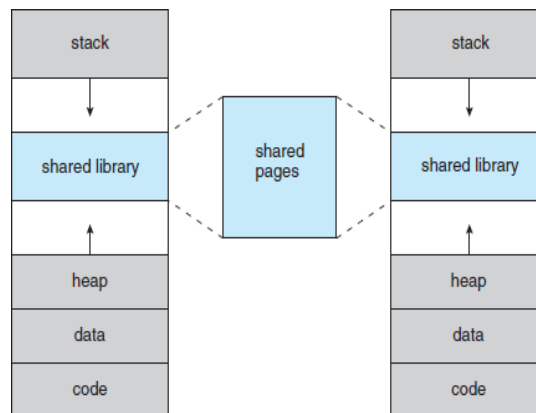
### Virtual Address Space



The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

- heap to grow upward in memory as it is used for dynamic memory allocation
- stack to grow downward in memory through successive function calls
- Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.
- In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by two or more processes through page sharing.

### Shared Library using Virtual Memory



This leads to the following benefits:

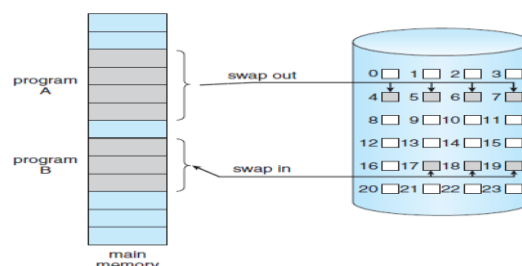
- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Actual pages where the libraries reside in physical memory are shared by all the processes.
- Similarly, virtual memory enables processes to share memory. Two or more processes can communicate through the use of shared memory.
- Virtual memory can allow pages to be shared during process creation with the `fork()` system call, thus speeding up process creation.

### Demand Paging

- Consider how an executable program might be loaded from disk into memory.
  - One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory.
  - **Demand paging** load pages only as they are needed.
- A **demand-paging** system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).

### Transfer of a paged memory to contiguous disk space

- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper.



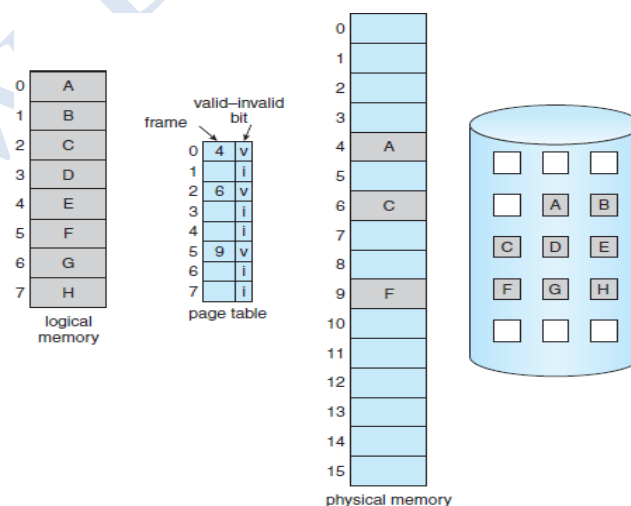
A **lazy swapper** never swaps a page into memory unless that page will be needed.

- A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

### Basic Concepts

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- It avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- Some form of hardware support is needed to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid -invalid bit scheme can be used for this purpose.
  - This time however, when this bit is set to ``valid'', the associated page is both legal and in memory.
  - If the bit is set to ``invalid'', the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.
  - The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk
- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- Access to a page marked invalid causes a **page-fault** trap.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the OS.
- The procedure for handling this page fault is straightforward.

**Page Table when some pages are not in main memory**

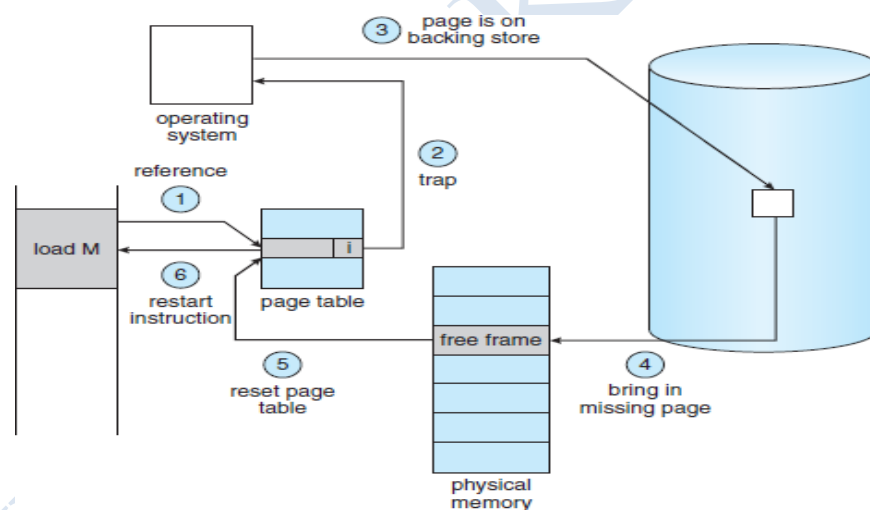


### Steps in handling a page fault

1. We check an internal table (in PCB) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table.
6. We restart the instruction that was interrupted by the trap.

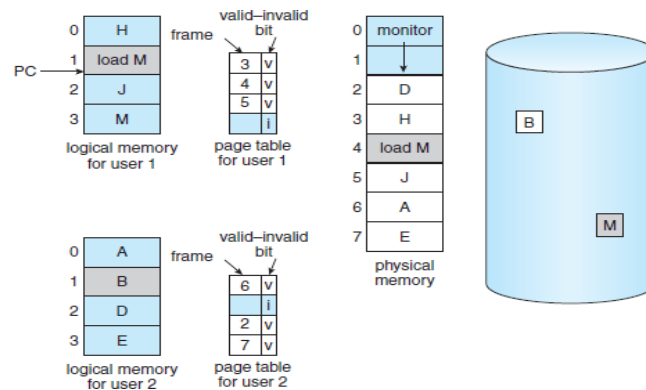
In the extreme case, we can start executing a process with no pages in memory.

- When the OS sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.
- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.
- At that point, it can execute with no more faults. This scheme is **pure demand paging**: Never bring a page into memory until it is required.



### Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. Let's compute the effective access time for a demand-paged memory.
  - For most computer systems, the memory-access time, denoted  $m_a$ , ranges from 10 to 200 nanoseconds.
  - As long as we have no page faults, the effective access time is equal to the memory access time.
  - If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.
  - Let  $p$  be the probability of a page fault ( $0 \leq p \leq 1$ ). We would expect  $p$  to be close to zero -that is, we would expect to have only a few page faults.



- The effective access time is then  

$$\text{effective access time} = (1 - p) \cdot t_a + p \cdot t_{pf}$$

We see, then, that the effective access time is directly proportional to the page-fault rate.

It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

- An additional aspect of demand paging is the handling and overall use of swap space.
  - Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used.
  - The system can therefore gain better paging throughput by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space.
  - Another option is to demand pages from the file system initially but to write the pages to swap space as they are replaced.

### Page Replacement

- If we increase our degree of multiprogramming, we are over-allocating memory.
  - If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare.
  - It is possible that each of these processes may suddenly try to use all ten of its pages resulting in a need for sixty frames.
- Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a significant amount of memory. This use can increase the strain on memory-placement algorithms.
- Over-allocation of memory manifests itself as follows
  - While a user process is executing, a page fault occurs.
  - The OS determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use.
- The OS could swap out a process, freeing all its frames and reducing the level of multiprogramming.

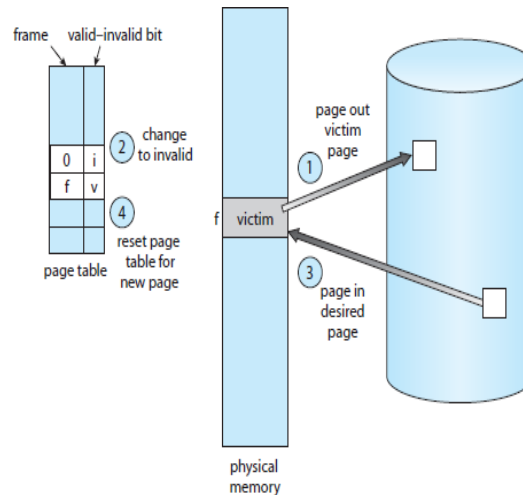


## Need for page replacement

### Basic Page Replacement

- Page replacement takes the following approach.
  - If no frame is free, we find one that is not currently being used and free it.
  - We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory.
  - We can now use the freed frame to hold the page for which the process faulted.

### Page Replacement



We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
  2. Find a free frame:
    - a. If there is a free frame, use it.
    - b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    - c. Write the victim frame to the disk; change the page and frame tables accordingly.
  3. Read the desired page into the newly freed frame; change the page and frame tables.
  4. Restart the user process.
- Notice that, if no frames are free, two page transfers (one out and one in) are required.
  - We can reduce this overhead by using a modify bit (or dirty bit).
  - The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
  - When we select a page for replacement, we examine its modify bit.
    1. If the bit is set, we know that the page has been modified since it was read in from the disk (write that page to the disk).
    2. If the modify bit is not set, the page has not been modified since it was read into memory (not write the memory page to the disk: It is already there).
  - This technique also applies to read-only pages (for example, pages of binary code).
  - This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.
  - Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.

- We must solve two major problems to implement demand paging:
  1. develop a frame-allocation algorithm. If we have multiple processes in memory, we must decide how many frames to allocate to each process.
  2. develop a page-replacement algorithm. When page replacement is required, we must select the frames that are to be replaced.
- Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.
- Second major problem will be discussed firstly.
- For a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address.

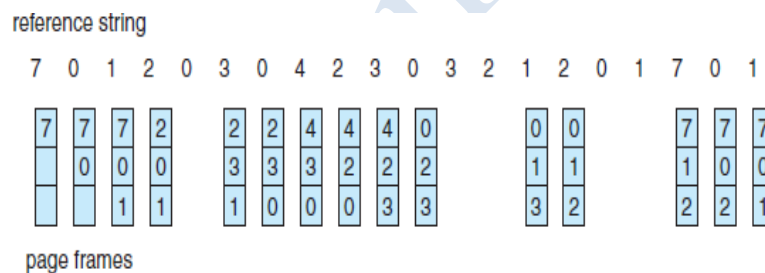
The following reference string will be used to exemplify for a memory with three frames.

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

### FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- For our example reference string, our three frames are initially empty.

The first three references (7, 0, 1) cause page faults and are brought into these empty frames.



- The next reference (2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- Because of this replacement, the next reference, to 0, will fault.
- Page 1 is then replaced by page 0. This process continues and there are 15 faults altogether.
- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
- On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed.
- On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.
- Notice that, even if we select for replacement a page that is in active use, everything still works correctly.

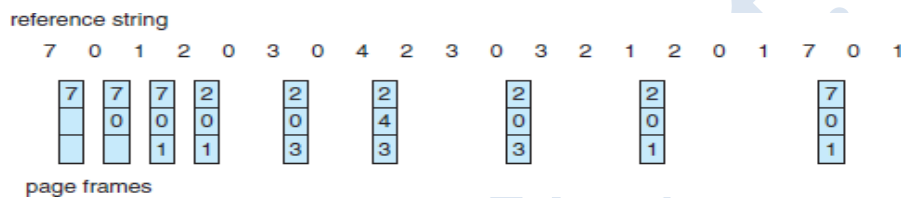
- After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page.
- Some other page will need to be replaced to bring the active page back into memory.
- Thus, a bad replacement choice increases the page-fault rate and slows process execution.
- It does not cause incorrect execution.

## Optimal Page Replacement

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms (called OPT or MIN). It is simply this:

**Replace the page that will not be used for the longest period of time.**

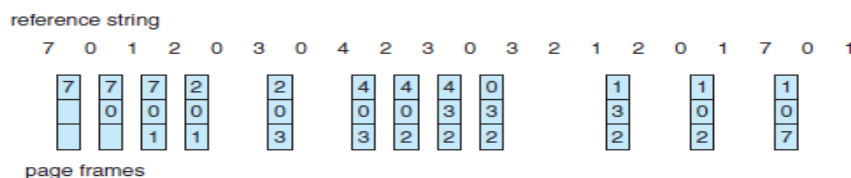
- Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.



- The first three references cause faults that fill the three empty frames.
  - The reference to page 2 replaces page 7, because 7 will not be used until reference 18,
  - whereas page 0 will be used at 5, and page 1 at 14.
  - The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
- With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.
- If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string (similar situation with the SJF CPU-scheduling algorithm).

## LRU Page Replacement

- The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that
  - the FIFO algorithm uses the time when a page was brought into memory,
  - whereas the OPT algorithm uses the time when a page is to be used.
- If we use the recent past as an approximation of the near future, then we can **replace the page that has not been used for the longest period of time**



This approach is the least-recently-used (LRU) algorithm. The LRU algorithm produces 12 faults.

- Notice that the first 5 faults are the same as those for optimal replacement.
- When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
- Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.
- Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

## Implementation of LRU

An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use.

Two implementations are feasible:

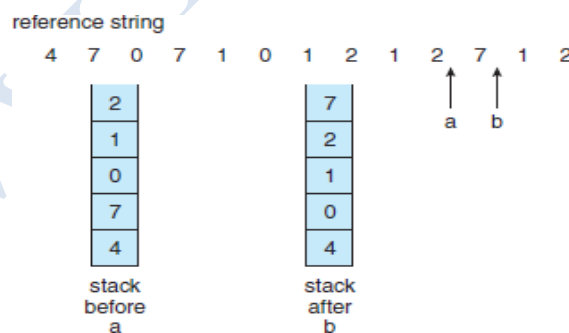
### Counters.

- we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter.
- The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- We replace the page with the smallest time value.

### Stack.

- Another approach to implementing LRU replacement is to keep a stack of page numbers.
- Whenever a page is referenced, it is removed from the stack and put on the top.
- The most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

Use of a stack to record the most recent page references.



## LRU-Approximation Page Replacement

- Few computer systems provide sufficient hardware support for true LRU page replacement.
- Some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used.
- Many systems provide some help, however, in the form of a **reference bit**.
- The reference bit for a page is set by the hardware whenever that page is referenced

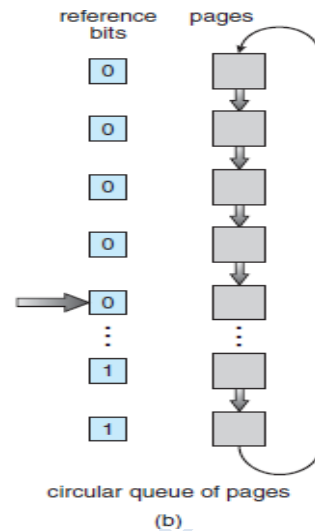
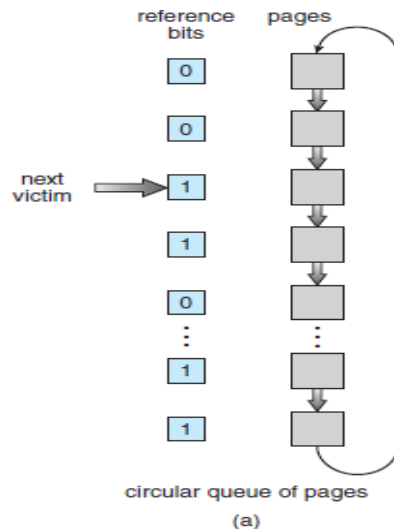
- Reference bits are associated with each entry in the page table.
- Initially, all bits are cleared (to 0) by the operating system.
- As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the *order* of use.

### Additional-Reference-Bits Algorithm

- We can gain additional ordering information by recording the reference bits at regular intervals.
- We can keep an 8-bit byte for each page in a table in memory.
- At regular intervals, a timer interrupt transfers control to the operating system.
- The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods.
- A page that is used at least once in each period has a shift register value of 11111111.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
- If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced.

### Second-Chance Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm.
- When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced. One way to implement the second-chance algorithm is as a circular queue.
- A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits.



### Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair.
- With these two bits, the four possible classes are:
  1. (0, 0) neither recently used nor modified—best page to replace
  2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
  3. (1, 0) recently used but clean—probably will be used again soon
  4. (1, 1) recently used and modified—probably will be used again soon, and the page will need to be written out to disk before it can be replaced
- Each page is in one of these four classes.
- When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs.
- We replace the first page encountered in the lowest nonempty class.

### Counting-Based Page Replacement

We can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced.
- The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

### Page-Buffering Algorithms

- Other procedures are often used in addition to a specific page-replacement algorithm.
- For example, systems commonly keep a pool of free frames.
- When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.

- This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.
- An expansion of this idea is to maintain a list of modified pages.
- Whenever the paging device is idle, a modified page is selected and is written to the disk.
- Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out..
- Another modification is to keep a pool of free frames but to remember which page was in each frame.
- Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.
- This technique is used in the VAX/VMS system along with a FIFO replacement algorithm

### **Applications and Page Replacement**

- Applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all.
- A typical example is a database, which provides its own memory management and I/O buffering.
- In another example, data warehouses frequently perform massive sequential disk reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones (as it starts its sequential
- reads again). Here, MFU would actually be more efficient than LRU.
- Because of such problems, some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed raw I/O.
- Raw I/O bypasses all the file system services, such as file I/O demand paging, file locking, prefetching, space allocation, file names, and directories.
- Note that although certain applications are more efficient when implementing their own special-purpose storage services on a raw partition, most applications perform better when they use the regular file-system services.

### **Allocation of Frames**

- We can require that the OS allocate all its buffer and table space from the free-frame list.
- When this space is not in use by the OS, it can be used to support user paging. The user process is allocated any free frame.

### **Minimum Number of Frames**

- Our strategies for the allocation of frames are constrained in various ways.
- We cannot, for example, allocate more than the total number of available frames
- We must also allocate at least a minimum number of frames.
- One reason for allocating at least a minimum number of frames involves performance.
- Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

### **Allocation Algorithms**



The easiest way to split  $m$  frames among  $n$  processes is to give everyone an equal share,  $m/n$  frames

- **Equal allocation**

For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool. This scheme is called equal allocation.

- **Proportional Allocation**

- An alternative is to recognize that various processes will need differing amounts of memory.
- Consider a system with a 1-KB frame size.
- If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.
- The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.
- To solve this problem, we can use proportional allocation, in which we allocate available memory to each process according to its size.
- Let the size of the virtual memory for process  $p_i$  be  $s_i$ , and define

$$S = \sum s_i.$$

- - Then, if the total number of available frames is  $m$ , we allocate  $a_i$  frames to process  $p_i$ , where  $a_i$  is approximately
  - $$a_i = s_i/S \times m.$$
  - we must adjust each  $a_i$  to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding  $m$ .
  - With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames
- $$10/137 \times 62 \approx 4, \text{ and}$$
- $$127/137 \times 62 \approx 57.$$

In this way, both processes share the available frames according to their "needs", rather than equally.

- In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level.
  - If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
  - If the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

### Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement.
- With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories:
  1. **Global replacement.** Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.

**2. Local replacement.** Local replacement requires that each process select from only its own set of allocated frames.

- With a local replacement strategy, the number of frames allocated to a process does not change.
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it
- Global replacement generally results in greater system throughput and is therefore the more common method.

### **Non-Uniform Memory Access**

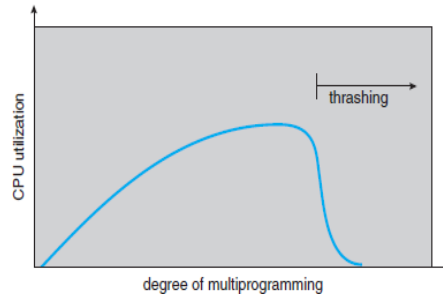
- Thus far in our coverage of virtual memory, we have assumed that all main memory is created equal—or at least that it is accessed equally.
- Systems with multiple CPUs a given CPU can access some sections of main memory faster than it can access others.
- These performance differences are caused by how CPUs and memory are interconnected in the system.
- Systems in which memory access times vary significantly are known collectively as non-uniform memory access (NUMA) systems

### **Thrashing**

- If the number of frames allocated to a low-priority process falls below the minimum number required, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames.
- In fact, look at any process that does not have "enough" frames.
  - If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
  - At this point, it must replace some page.
  - However, since all its pages are in active use, it must replace a page that will be needed again right away.
  - Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- **Definition:** This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing

### **Cause of Thrashing**

- Thrashing results in severe performance problems.
- The OS monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.



- Now suppose that a process enters a new phase in its execution and needs more frames.
  - It starts faulting and taking frames away from other processes (global page-replacement algorithm).
  - These processes need those pages, however, and so they also fault, taking frames from other processes.
  - These faulting processes must use the paging device to swap pages in and out.
  - As processes wait for the paging device, CPU utilization decreases.
- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
  - The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
  - As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.
- Thrashing has occurred. The page-fault rate increases tremendously. As a result, the effective memory-access time increases.
- No work is getting done, because the processes are spending all their time paging.
- As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

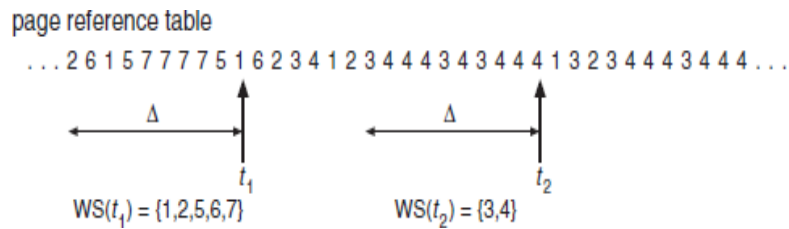
### Preventing Thrashing

- We can limit the effects of thrashing by using a local replacement algorithm.
- With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved.
- To prevent thrashing, we must provide a process with as many frames as it needs.
- The **working-set strategy** starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.
- The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

### Working-Set Model

- The **working-set model** is based on the assumption of locality.
- This model uses a parameter,  $w$ , to define the **working-set window**. The idea is to examine the most recent  $w$  page references.

- The set of pages in the most recent  $\Delta$  page references is the **working set**.



If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set time units after its last reference.

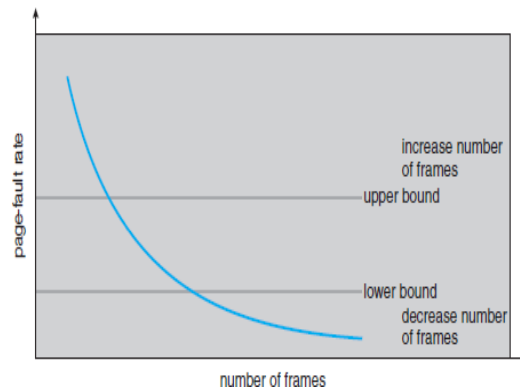
- Thus, the working set is an approximation of the program's locality.
- if  $\Delta = 10$  memory references, then the working set at time  $t_1$  is  $\{1, 2, 5, 6, 7\}$ .
- By time  $t_2$ , the working set has changed to  $\{3, 4\}$ .
- The accuracy of the working set depends on the selection of  $\Delta$ .
- If  $\Delta$  is too small, it will not encompass the entire locality; if it is too large, it may overlap several localities.
- In the extreme, if  $\Delta$  is infinite, the working set is the set of pages touched during the process execution.
- The most important property of the working set, then, is its size.
- If we compute the working-set size,  $WSS_i$ , for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

- where  $D$  is the total demand for frames. Each process is actively using the pages in its working set.
- Thus, process  $i$  needs  $WSS_i$  frames. If the total demand is greater than the total number of available frames ( $D > m$ ), thrashing will occur, because some processes will not have enough frames.
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.
- The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window.

### Page-Fault Frequency

- Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate.
- When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
- We can establish upper and lower bounds on the desired page-fault rate.



- If the actual page-fault rate exceeds the upper limit, we allocate the process another from the process.
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.

### Allocating Kernel Memory

- When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel.
- This list is typically populated using a page-replacement algorithm and most likely contains free pages scattered throughout physical memory.
- If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.
- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes.

There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

Two strategies for managing free memory that is assigned to kernel processes:

- buddy system
- slab allocation.

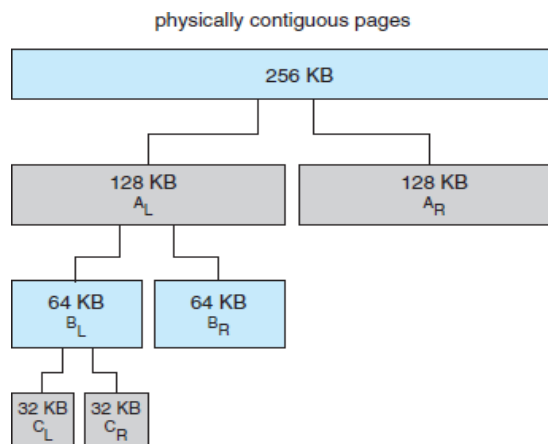
### **Buddy System**

- The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth).
- A request in units not appropriately sized is rounded up to the next highest power of 2. For

example, a request for 11 KB is satisfied with a 16-KB segment.

#### Simple example:

- Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory.
- The segment is initially divided into two **buddies**—which we will call *AL* and *AR*—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies—*BL* and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*. One of these buddies is used to satisfy the 21-KB request. *CL* is the segment allocated to the 21-KB request.
- This scheme is illustrated in the following Figure



#### Advantage :

- How quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**.
- For example, when the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment. This segment, *BL*, can in turn be coalesced with its buddy *BR* to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

#### Drawback :

- Rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments.
- For example, a 33-KB request can only be satisfied with a 64-KB segment.

#### Slab Allocation

A second strategy for allocating kernel memory is known as **slab allocation**.

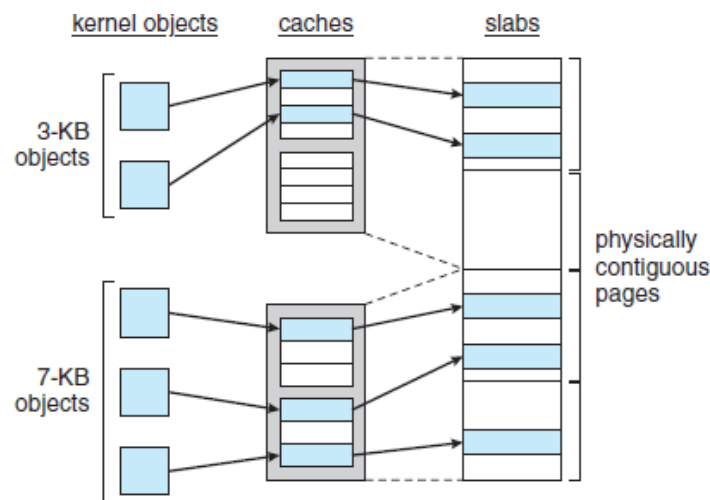
- A **slab** is made up of one or more physically contiguous pages.
- A **cache** consists of one or more slabs.
- There is a single cache for each unique kernel data structure—for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth.
- Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents.
- For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth.
- The relationship among slabs, caches, and objects is shown in Figure. The figure shows two

kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

The slab-allocation algorithm uses caches to store kernel objects.

- When a cache is created, a number of objects—which are initially marked as free—are allocated to the cache.
- The number of objects in the cache depends on the size of the associated slab.
- For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free.

When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.



Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor.

- In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory.
- When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache.
- The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.
- In Linux, a slab may be in one of three possible states:
  1. **Full.** All objects in the slab are marked as used.
  2. **Empty.** All objects in the slab are marked as free.
  3. **Partial.** The slab consists of both used and free objects.
- The slab allocator first attempts to satisfy the request with a free object in a partial slab.
- If none exists, a free object is assigned from an empty slab.
- If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

#### Benefits:

- No memory is wasted due to fragmentation. When the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
- Memory requests can be satisfied quickly. The slab allocation scheme is thus effective for



managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel.

When the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

- Recent distributions of Linux now include two other kernel memory allocators— the SLOB and SLUB allocators.
- The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for Simple List of Blocks) works by maintaining three lists of objects:  
*small* (for objects less than 256 bytes),  
*medium* (for objects less than 1,024 bytes), and  
*large* (for objects greater than 1,024 bytes).
- The SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB addresses performance issues with slab allocation by reducing much of the overhead required by the SLAB allocator.
- One change is to move the metadata that is stored with each slab under SLAB allocation to the page structure the Linux kernel uses for each page.
- Additionally, SLUB removes the per-CPU queues that the SLAB allocator maintains for objects in each cache.
- For systems with a large number of processors, the amount of memory allocated to these queues was not insignificant. Thus, SLUB provides better performance as the number of processors on a system increases.

### Operating-System Examples

#### Windows

- Windows implements virtual memory using demand paging with **clustering**.
- Clustering handles page faults by bringing in not only the faulting page but also several pages following the faulting page.
- When a process is first created, it is assigned a working-set minimum and maximum.
- The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory. If sufficient memory is available, a process may be assigned as many pages as its **working-set maximum**.
- The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that is used to indicate whether sufficient free memory is available.
- If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from this list of free pages.
- If a process that is at its working-set maximum incurs a page fault, it must select a page for replacement using a local LRU page-replacement policy.
- When the amount of free memory falls below the threshold, the virtual memory manager uses a tactic known as **automatic working-set trimming** to restore the value above the threshold.
- Automatic working-set trimming works by evaluating the number of pages allocated to processes.
- If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages until the process reaches its working-set minimum.
- A process that is at its working-set minimum may be allocated pages from the free-page-frame list once sufficient free memory is available.



- Windows performs working-set trimming on both user mode and system processes.

## Solaris

In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains.

- Therefore, the kernel keep a sufficient amount of free memory available.
- Associated with this list of free pages is a parameter—*lotsfree*—that represents a threshold to begin paging.
- The *lotsfree* parameter is typically set to 1/64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than *lotsfree*.
- If the number of free pages falls below *lotsfree*, a process known as a **pageout** starts up.
- The pageout process is similar to the second-chance algorithm except that it uses two hands while scanning pages, rather than one.

The **pageout process** works as follows:

- The front hand of the clock scans all pages in memory, setting the reference bit to 0.
- Later, the back hand of the clock examines the reference bit for the pages in memory, appending each page whose reference bit is still set to 0 to the free list and writing to disk its contents if modified.
- Solaris maintains a cache list of pages that have been “freed” but have not yet been overwritten.
- The free list contains frames that have invalid contents. Pages can be reclaimed from the cache list if they are accessed before being moved to the free list.

Parameters:

- *scanrate* is to control the rate at which pages are scanned. It is expressed in pages per second and ranges from *slowscan* to *fastscan*.
- When free memory falls below *lotsfree*, scanning occurs at *slowscan* pages per second and progresses to *fastscan*, depending on the amount of free memory available.
- The default value of *slowscan* is 100 pages per second.
- *Fastscan* is typically set to the value (total physical pages)/2 pages per second, with a maximum



of 8,192 pages per second. This is shown in Figure.

- The distance (in pages) between the hands of the clock is determined by a system parameter, *handspread*. The amount of time between the front hand’s clearing a bit and the back hand’s investigating its value depends on the *scanrate* and the *handspread*.
- If scanrate is 100 pages per second and handspread is 1,024 pages, 10 seconds can pass between the time a bit is set by the front hand and the time it is checked by the back hand.
- The amount of time between clearing and investigating a bit is often a few seconds. The pageout process checks memory four times per second. However, if free memory falls below the value of *desfree*, pageout will run a hundred times per second with the intention of keeping at least *desfree* free memory available.

- If the pageout process is unable to keep the amount of free memory at desfree for a 30-second average, the kernel begins swapping processes, thereby freeing all pages allocated to swapped processes.
- In general, the kernel looks for processes that have been idle for long periods of time. If the system is unable to maintain the amount of free memory at minfree, the pageout process is called for every request for a new page.

Recent releases of the Solaris kernel have provided **enhancements** of the paging algorithm.

- One enhancement involves recognizing pages from shared libraries. Pages belonging to libraries that are being shared by several processes—even if they are eligible to be claimed by the scanner—are skipped during the page-scanning process.
- Another enhancement concerns **distinguishing pages that have been allocated to processes from pages allocated** to regular files. This is known as **priority paging**.