UNIT-IV

MEMORY MANAGEMENT

MAIN MEMORY

The main purpose of a computer system is to execute programs.

- During the execution of these programs together with the data they access, must be stored in main memory.
- Memory consists of a large array of bytes. Each Byte has its own address.
- CPU fetches instructions from memory according to the value of the program counter.

BASIC HARDWARE

CPU can access data directly only from Main memory and processor registers.

- Main memory and the Processor registers are called **Direct Access Storage Devices**.
- Any instructions in execution and any data being used by the instructions must be in one of these direct-access storage devices.
- If the data are not in memory then the data must be moved to main memory before the CPU can operate on them.
- Registers that are built into the CPU are accessible within one CPU clock cycle.
- Completing a memory access from main memory may take many CPU clock cycles. Memory access from main memory is done through memory bus.
- In such cases, the processor needs to **stall**, since it does not have the required data to complete the instruction that it is executing.
- To avoid **memory stall**, we need to implement Cache memory in between Main memory and CPU.

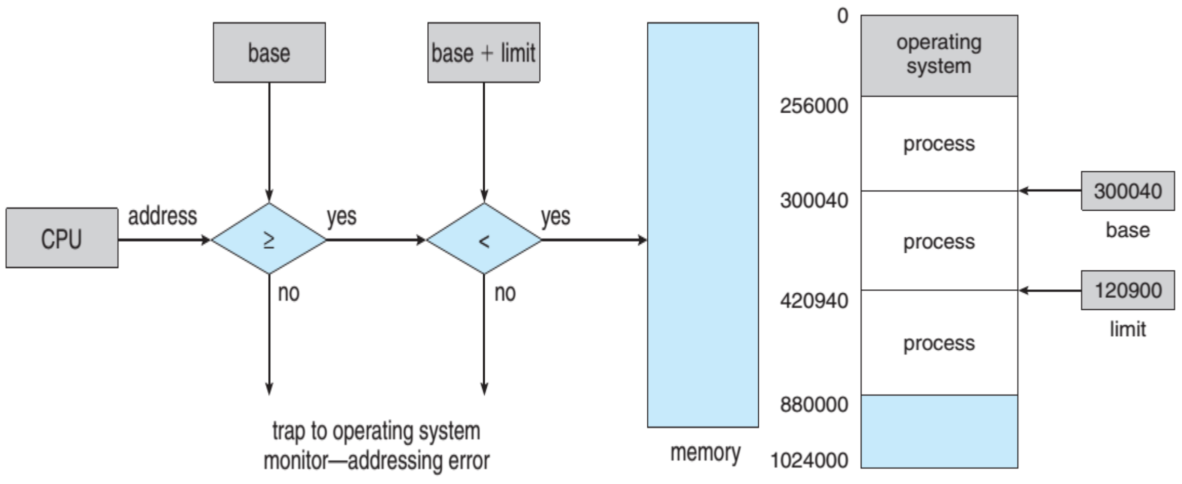
BASE REGISTER & LIMIT REGISTER

Each process has a separate memory space that protects the processes from each other. It is fundamental to having multiple processes loaded in memory for concurrent execution.

There are two register that provides protection: Base register and Limit register

- **Base Register** holds the smallest legal physical memory address.
- **Limit register** specifies the size of the range (i.e. process size).

Example: if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).



The base and limit registers can be loaded only by the operating system by using a special privileged instruction that can be executed only in kernel mode.

- 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 operating system, which treats the attempt as a **Fatal Error**.
- This scheme prevents a user program from either accidentally or deliberately modifying the code or data structures of other users and the operating system.

Operating system executing in kernel mode is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to do certain tasks such as:

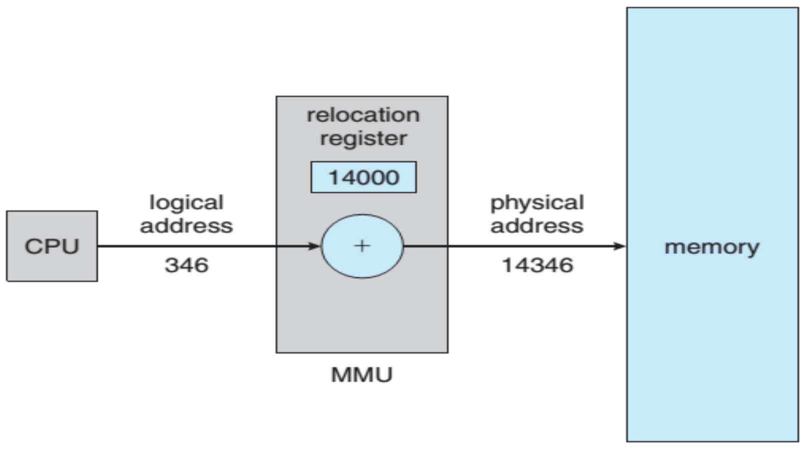
- Load users' programs into users' memory
- To dump out those programs in case of errors
- To access and modify parameters of system calls
- To perform I/O to and from user memory etc.

Example: A Multiprocessing Operating system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

Logical Versus Physical Address Space

- Logical address is the address generated by the CPU.
- Physical address is the address that is loaded into the **Memory-Address Register** of the memory.
- The set of all logical addresses generated by a program is a **Logical Address Space**.
- The set of all physical addresses corresponding to these logical addresses is a **Physical Address Space**.
- The Compile-time and Load-time address-binding methods generate identical logical and physical addresses.
- The execution-time address binding scheme results in different logical and physical addresses. At this time we call logical address as **Virtual address**.
- The run-time mapping from virtual address to physical addresses is done by a hardware device called the **Memory-Management Unit (MMU)**.
- Base register is now called a **Relocation Register**. Value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.

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.
- Only when it is used as a memory address, it is relocated relative to the base register.
- The user program deals with logical addresses. The Memory-mapping hardware converts logical addresses into physical addresses.
- Final location of a referenced memory address is not determined until the reference is made.

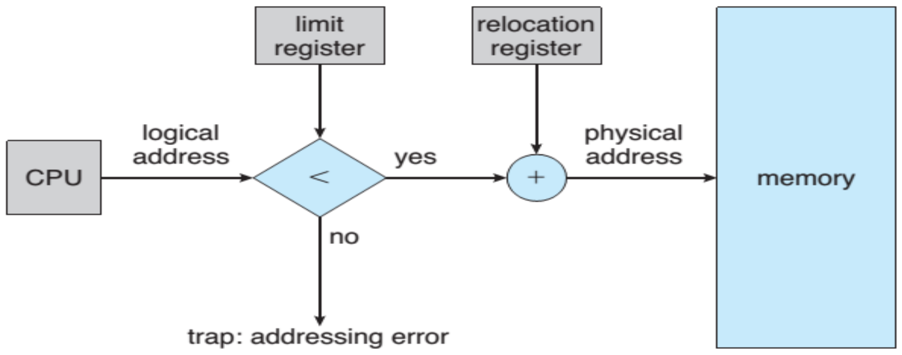
Example: Logical addresses in the range **0 to max** and Physical addresses in the range **(R+0)** to (**R + max**) for a base value **R**.

- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- These logical addresses must be mapped to physical addresses before they are used.

Memory Protection

OS can prevent a process from accessing other process memory. We use two registers for this purpose: Relocation register and Limit register.

- Relocation register contains the value of the smallest physical address such as 100040.
- Limit register contains the range of logical addresses such as 74600.
- Each logical address must be within the range specified by the limit register.
- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.
- Memory Management Unit maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation register and limit registers with the correct values as part of the context switch.
- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.



CONTIGUOUS MEMORY ALLOCATION

Memory allocation can be done in two ways:

1. Fixed Partition Scheme (Multi-programming with Fixed Number of Tasks)
2. Variable partition scheme (Multi-programming with Variable Number of Tasks)

Fixed Partition Scheme (MFT)

The memory can be divided into several **Fixed-Sized** partitions.

- Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.
- In this **Multiple-Partition** method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Note: This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use.

FRAGMENTATION

There are 2-problems with Memory allocation

1. Internal Fragmentation
2. External Fragmentation

INTERNAL FRAGMENTATION

Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.

- Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.
- The overhead to keep track of this hole will be substantially larger than the hole itself.
- 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.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two numbers is **Internal Fragmentation**. It is unused memory that is internal to a partition.

VARIABLE PARTITION SCHEME (MVT)

In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

- Initially, all memory is available for user processes and it is considered one large block of available memory called as **Hole**.
- Eventually the memory contains a set of holes of various sizes.
- As processes enter the system, they are put into an **Input Queue**.
- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory and it can then compete for CPU time.
- When a process terminates, it releases its memory. The operating system may use this free fill with another process from the input queue.

Memory is allocated to processes until the memory requirements of the next process cannot be satisfied (i.e.) there is no available block of memory is large enough to hold that process.

Then operating system can wait until a large block is available for the process or it can skip the process and moves down to the input queue to see whether the smaller memory requirements of some other process can be met.

- The memory blocks available comprise a **set** of holes of various sizes scattered throughout main memory.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts. One part is allocated to the arriving process and the other part is returned to the set of holes.
- 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.
- At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

EXTERNAL FRAGMENTATION

- Both the first-fit and best-fit strategies for memory allocation suffer from **External Fragmentation**.
- As processes are loaded and removed from main memory, the free memory space is broken into small pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous, the storage is fragmented into a large number of small holes.

- External fragmentation problem can be severe. In the worst case, we could have a block of free memory between every two processes that is wasted.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

COMPACTION : Solution to External fragmentation:

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.
- Compaction is possible only if relocation is dynamic and is done at execution time.
- If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
- If relocation is static and is done at assembly or load time, compaction cannot be done.

Note: Compaction can be expensive, because it moves all processes toward one end of memory. All holes move in the other direction and produces one large hole of available memory.

Other solutions to External fragmentation are **Segmentation** and **Paging.** They allow a process to be allocated physical memory wherever such memory is available. These are Non-contiguous memory allocation techniques.

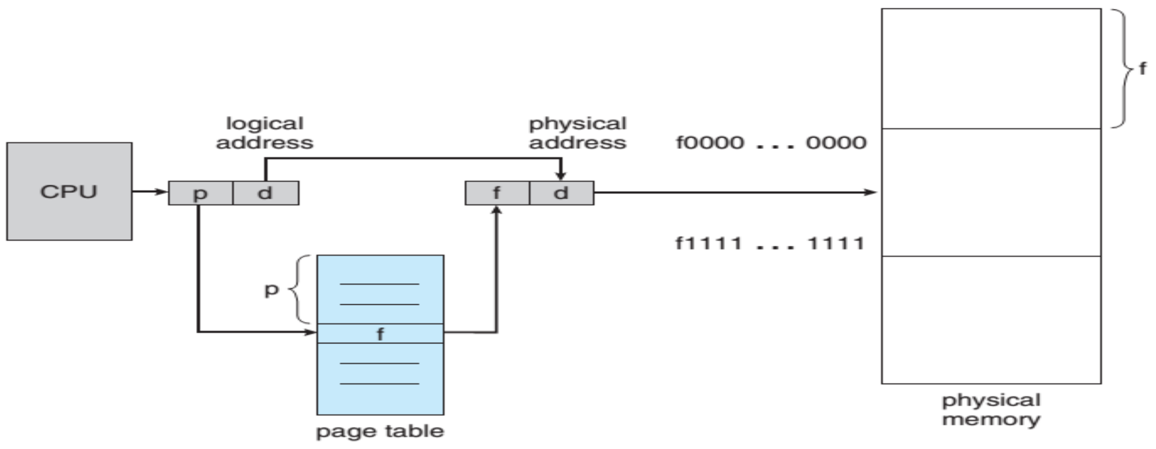
NON-CONTIGUOUS MEMORY ALLOCATION

PAGING

Paging also permits the physical address space of a process to be noncontiguous.
Paging avoids External fragmentation and need for compaction.
Paging is implemented through cooperation between the operating system and the computer hardware.

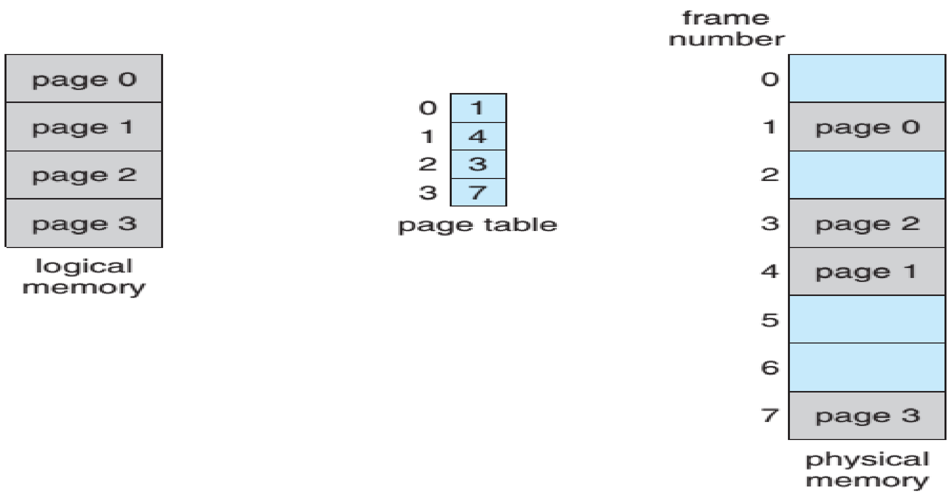
- Physical memory is divided into fixed-sized blocks called **Frames.**
- Logical memory is divided into blocks of the same size called **Pages**.
- Frame size is equal to the Page size.
- When a process is to be executed, its pages are loaded into any available memory frames from their source such as a file system or backing store.
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
- Frame table maintains list of frame and the allocation details of the frames (i.e.) A frame is free or allocated to some page.
- Each process has its own page table. When a page is loaded into main memory the corresponding page table is active in the system and all other page tables are inactive.
- Page tables and Frame tables are kept in main memory. A **Page-Table Base Register (PTBR)** points to the page table.

PAGING HARDWARE SUPPORT DIAGRAM



- 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.

The below shows the paging of Logical and Physical memory:



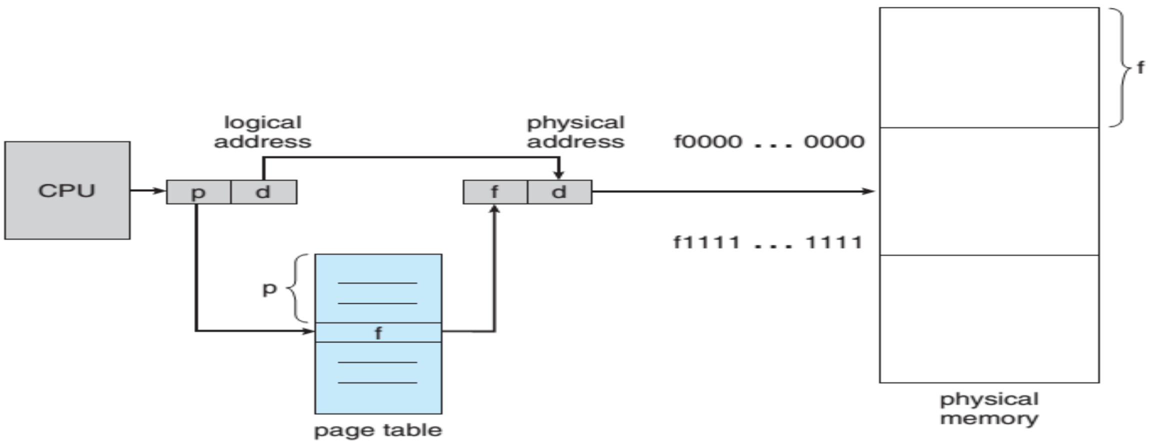
Paging scheme avoids external fragmentation but it creates internal fragmentation because of fixed size pages.

- If page size is **2048** bytes, a process of **20489** bytes will need **10** pages plus **9** bytes.
- It will be allocated **11** frames, resulting in internal fragmentation of **2048−9 = 2037** bytes.
- In the worst case, a process would need ***n*** pages plus **1** byte. It would be allocated ***n*** **+ 1** frames resulting in internal fragmentation of almost an entire frame.
- If the page size is small then the number of entries in page table is more this will leads to huge number of context switches.
- If the page size is large then the number of entries in page table is less and the number of context switches is less.

Problem: Slow access of a user memory location

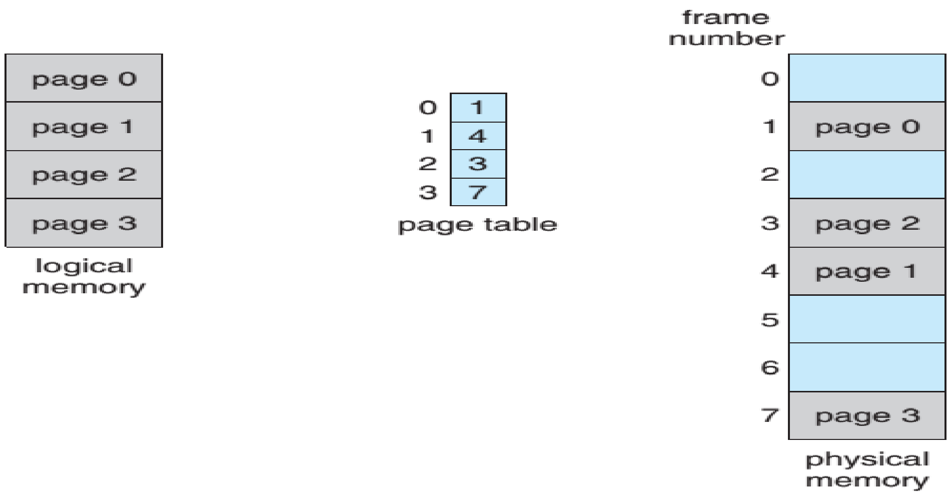
- If we want to access location ***i***, we must first index into the page table using the value in the PTBR offset by the page number for ***i***. This task requires a memory access.
- It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory.
- With this scheme, ***two*** memory accesses are needed to access a byte (i.e.) one for the page-table entry, one for the byte.
- Thus, memory access is slowed by a factor of 2. This delay is intolerable.

PAGING HARDWARE SUPPORT DIAGRAM



- 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.

The below shows the paging of Logical and Physical memory:



Paging scheme avoids external fragmentation but it creates internal fragmentation because of fixed size pages.

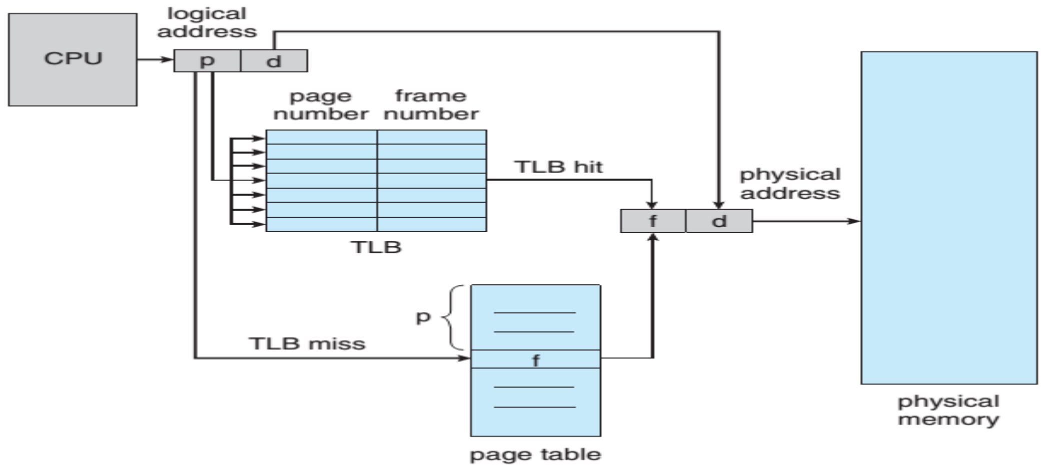
- If page size is **2048** bytes, a process of **20489** bytes will need **10** pages plus **9** bytes.
- It will be allocated **11** frames, resulting in internal fragmentation of **2048−9 = 2037** bytes.
- In the worst case, a process would need ***n*** pages plus **1** byte. It would be allocated ***n* + 1** frames resulting in internal fragmentation of almost an entire frame.
- If the page size is small then the number of entries in page table is more this will leads to huge number of context switches.
- If the page size is large then the number of entries in page table is less and the number of context switches is less.

Problem: Slow access of a user memory location

- If we want to access location ***i***, we must first index into the page table using the value in the PTBR offset by the page number for ***i***. This task requires a memory access.
- It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory.
- With this scheme, ***two*** memory accesses are needed to access a byte (i.e.) one for the page-table entry, one for the byte.
- Thus, memory access is slowed by a factor of 2. This delay is intolerable.

Translation Look-aside Buffer (TLB)

TLB is a special, small, fast lookup hardware cache. It is associative, high-speed memory.



- Each entry in the TLB consists of two parts: **a key (or tag)** and **a value**.
- The size of TLB is between 32 and 1024 entries.
- 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.
- Multiple levels of TLBs are maintained if the system is having multiple levels of Cache.

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 found, its frame number is immediately available and is used to access memory. This is called **TLB Hit.**
- These TLB lookup steps are executed as part of the instruction pipeline within the CPU, which does not add any performance penalty compared with a system that does not implement paging.
- If the page number is not in the TLB is known as a **TLB miss**. At the time of 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 **OS**. When the frame number is obtained, we can use it to access memory.
- Then we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, an existing entry must be selected for replacement by using any of page replacement algorithms.
- **Wired Down entries:** These are the entries that cannot be removed from the TLB. Examples for these are **Key Kernel Code entries.**

Address Space Identifiers (ASID's) in TLB

TLBs store Address-Space Identifiers in each TLB entry.

- An ASID uniquely identifies each process and is used to provide address-space protection for that process.
- When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.

- If the ASIDs do not match, the attempt is treated as a TLB miss.
- In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.
- If the TLB does not support separate ASIDs, then every time a new page table is selected, the TLB must be **flushed** or erased to ensure that the next executing process does not use the wrong translation information.
- Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

TLB Hit Ratio/ Miss Ratio

Percentage of times that the page number is found in the TLB is called the **Hit ratio**.
Percentage of times that the page number is not found in the TLB is called the **Miss ratio**.

$$\boxed{\textbf{TLB Miss ratio=1-Hit ratio}}$$

Effective Memory Access Time

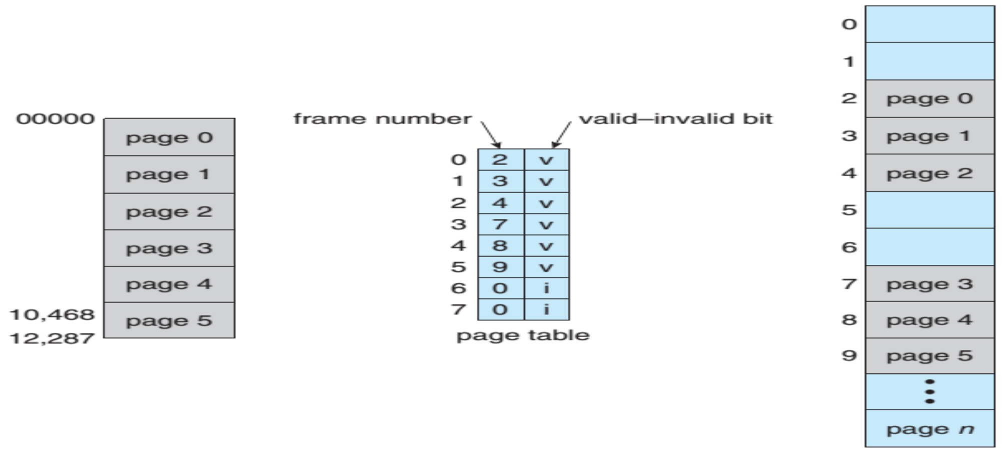
It is the sum of time taken for a page to access for TLB hit ratio and TLB miss ratio.
Example: An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the 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 for 100 nanoseconds and then access the desired byte in memory for 100 nanoseconds with a total of 200 nanoseconds.

$$\text{Effective Memory Access Time} = (0.80 \times 100\text{ ns}) + (0.20 \times 200\text{ ns})$$
$$= 120\text{ nanoseconds}$$

Memory Protection in Paging Environment

Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table.

- A one bit **valid–invalid** bit is attached to each entry in the page table.
- When this bit is set to *valid*, the associated page is in the process's logical address space and it is 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 operating system sets this bit for each page to allow or disallow access to the page.



Consider the above figure: A system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Each page size is of 2 KB.

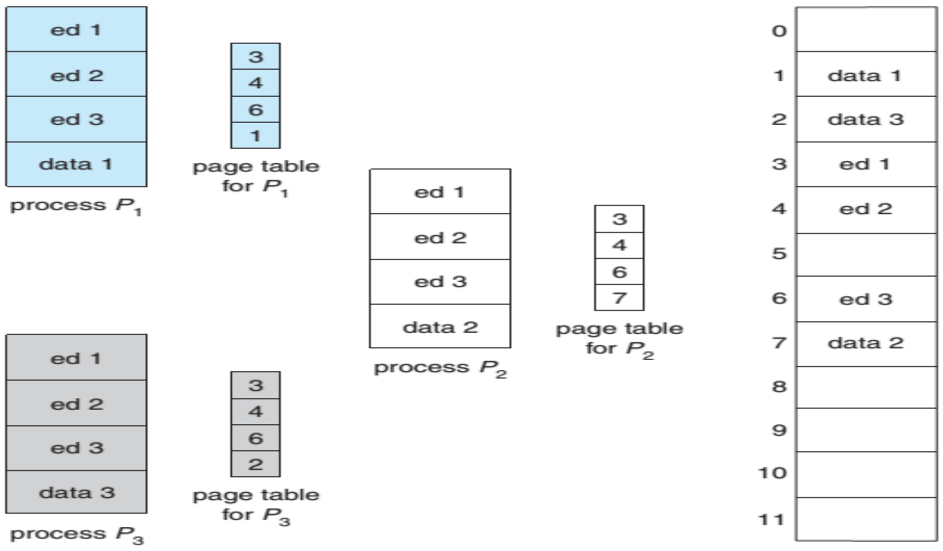
- Addresses in pages 0, 1, 2, 3, 4 and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7 will find that the valid–invalid bit is set to invalid and the computer will trap to the operating system indicating that **Invalid** page reference.

Problem: The program extends only to address 10468, any reference beyond that address is illegal. But references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid.

Solution: To avoid this problem we use a register **Page-Table Length Register (PTLR)** to indicate the size of the page table. This PLTR value is checked against every logical address to verify that the address is in the valid range for the process.

Shared Pages

Paging has an advantage of **Sharing Common Code** This is important in Time sharing Environment.



Consider the above figure that shows a system that supports 40 users, each of whom executes a text editor.

- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- If the code is **Reentrant Code** or **Pure Code** or **Reusable code** it can be shared.
- Each process has its own data page. All three processes sharing a three-page editor each page 50 KB in size.
- Reentrant code is non-self-modifying code (i.e.) it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- 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 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.

- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2,150 KB instead of 8,000 KB by saving 5850 KB.

Other heavily used programs can also be shared such as compilers, window systems, run-time libraries, database systems and so on.

STRUCTURE OF THE PAGE TABLE

Page tables can be structured in 3 ways:

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

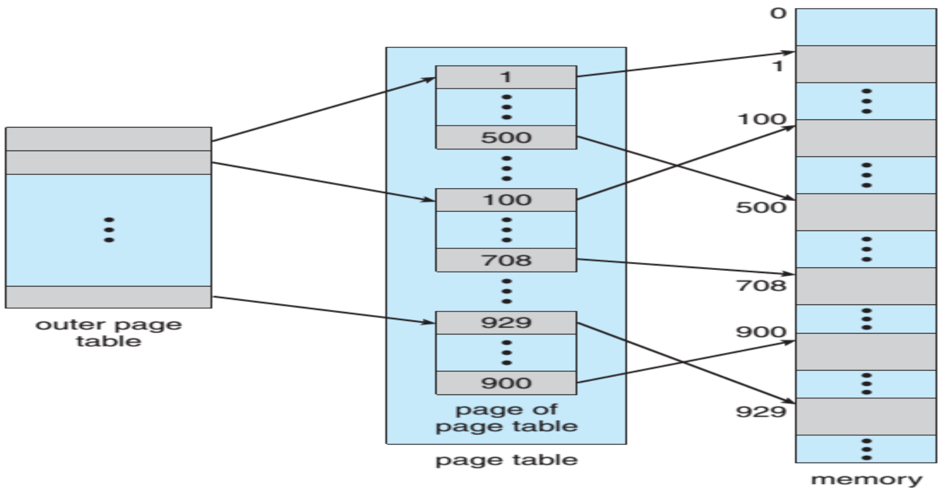
Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64} Bytes) that leads to excessively larger page tables.

Consider a system with a 32-bit (4GB) logical address space.

- If the page size in such a system is 4 KB (**2^{12}**), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$).
- 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.
- These page tables are not allocated in main memory contiguously and we will divide this page tables into smaller pieces.

Hierarchical paging uses Two Level Paging Algorithm for structuring of page tables. In Two level paging algorithm Page tables are itself paged.

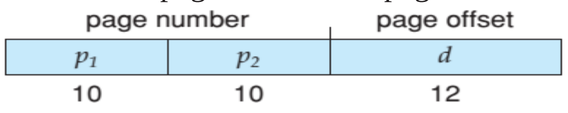


Consider the system with a 32-bit logical address space and a page size of 4 KB.

- 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.

A logical address has two indexes: p1 and p2.

- *p*1 is an index into the outer page table
- *p*2 is the displacement within the page of the inner page table.



The below figure shows the Address translation for a **Two-level** 32-bit paging architecture. Address translation works from the outer page table inward, this scheme is also known as a

STRUCTURE OF THE PAGE TABLE

Page tables can be structured in 3 ways:

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

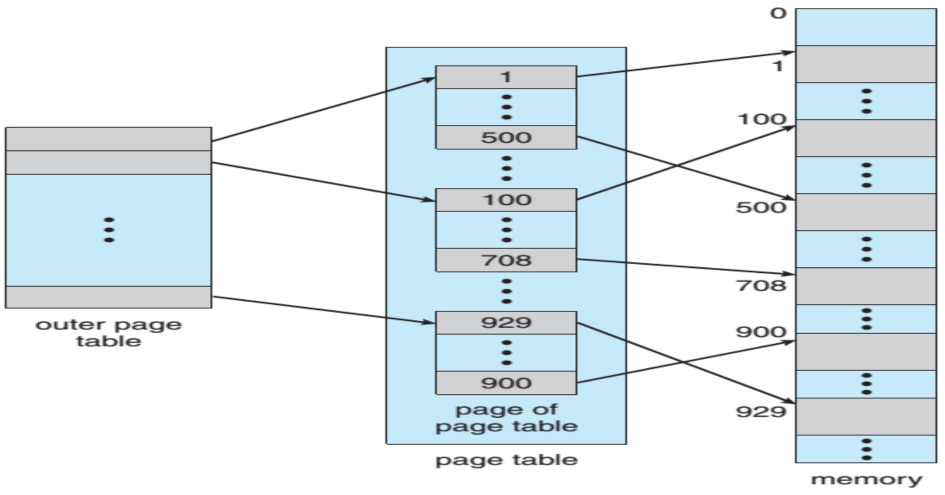
Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64} Bytes) that leads to excessively larger page tables.

Consider a system with a 32-bit (4GB) logical address space.

- If the page size in such a system is 4 KB (**2^{12}**), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$).
- 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.
- These page tables are not allocated in main memory contiguously and we will divide this page tables into smaller pieces.

Hierarchical paging uses Two Level Paging Algorithm for structuring of page tables. In Two level paging algorithm Page tables are itself paged.

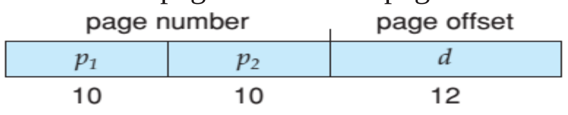


Consider the system with a 32-bit logical address space and a page size of 4 KB.

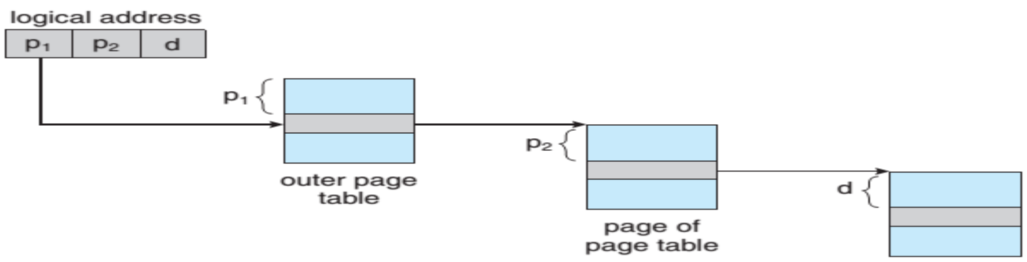
- A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.
- Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

A logical address has two indexes: p1 and p2.

- *p*1 is an index into the outer page table
- *p*2 is the displacement within the page of the inner page table.



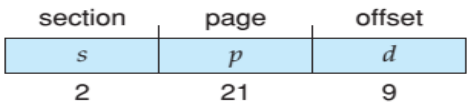
The below figure shows the Address translation for a **Two-level** 32-bit paging architecture. Address translation works from the outer page table inward, this scheme is also known as a **Forward-Mapped Page Table**.



Example of Two level paging: VAX Mini Computer

- The VAX was the most popular minicomputer from 1977 through 2000.
- The VAX architecture supported a variation of **Two-Level paging**.
- The VAX is a 32- bit machine with a page size of 512 bytes.
- The logical address space of a process is divided into **4-equal** sections, each of which consists of **2^{30}** bytes.
- Each section represents a different part of the logical address space of a process.
- The first 2 high-order bits of the logical address designate the appropriate section.
- The next 21 bits represent the logical page number of that section and the final 9 bits represent an offset in the desired page.
- By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them.
- Entire sections of virtual address space are frequently unused and multilevel page tables have no entries for these spaces, greatly decreasing the amount of memory needed to store virtual memory data structures.

An address on the VAX architecture consists of **3-parts**: Segment number (**s**), index to page table (**p**), Displacement with in the page (**d**).

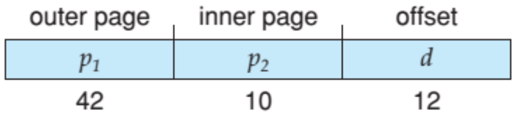


- After this scheme is used, the size of a one-level page table for a VAX process using one section is 2^{21} bits ∗ 4 bytes per entry = 8 MB.
- To further reduce main-memory use, the VAX pages the user-process page tables.

Problems with Two level paging

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

Let’s take the page size in such a system is 4 KB (**2^{12}**). In this case, the page table consists of up to **2^{52}** entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long or contain **2^{10}** 4-byte entries.

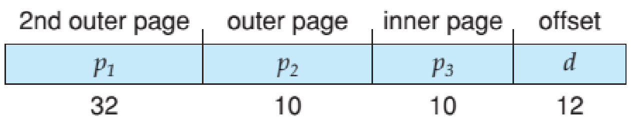


The outer page table consists of **2^{42}** entries or **2^{44}** bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.

To avoid this problem we can divide the outer page again called Three level paging.

Three level paging

Suppose that the outer page table is made up of standard-size pages (**2^{10}** entries or **2^{12}** bytes). In this case, a 64-bit address space is still daunting:



Outer page table is still 2^{34} bytes (16 GB) in size. We can still divide this into **4-Level** paging.

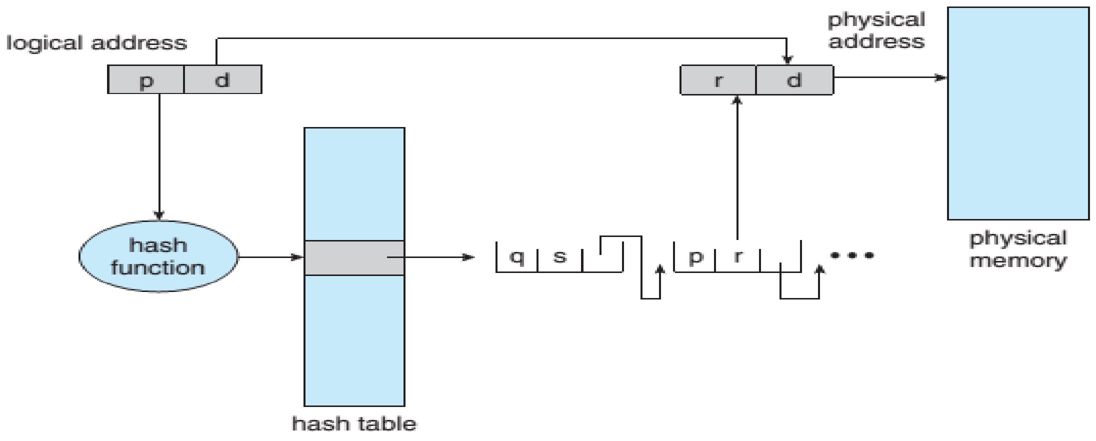
HASHED PAGE TABLES

Hashed page table is used to handling address spaces larger than 32 bits.

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:

1. Virtual page number
2. Value of the mapped page frame
3. A pointer to the next element in the linked list.



- 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.

For 64 bit address space Clustered page table has been proposed.

- Clustered page tables 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

Problem with page tables:

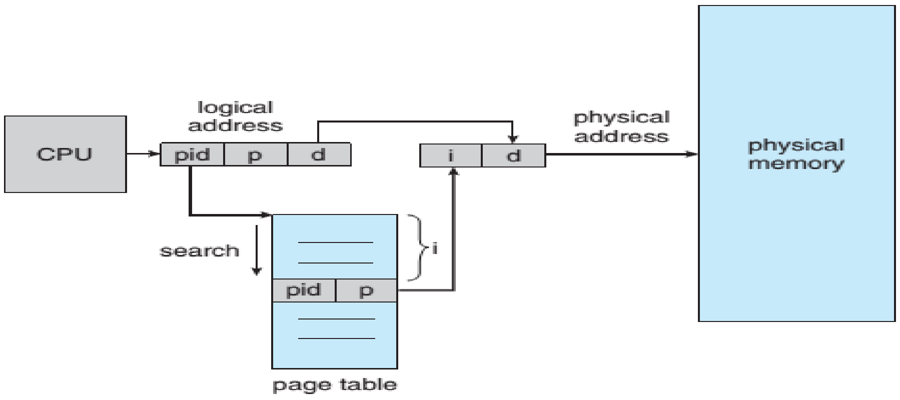
- Each process has an associated page table. The page table has one entry for each page that the process is using.
- Processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address.
- Since the table is sorted by 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.
- The drawback of this method is, 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.

Solution: Inverted page tables will solve the above problem

An inverted page table has one entry for each real page (i.e. 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.
- 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: Inverted page tables are used in the 64-bit UltraSPARC and PowerPC systems.



The above figure shows the inverted page tables used in IBM RT:

- Each virtual address in the system consists of: **<process-id, page-number, offset>**
- Each inverted page-table entry is a pair <process-id, page-number> where the process-id is the address-space identifier.
- When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number> is presented to the memory subsystem. The inverted page table is then searched for a match.
- If a match is found at entry ***i*** then the physical address **<*i*, offset>** is generated.
- If no match is found, then an illegal address access has been attempted.

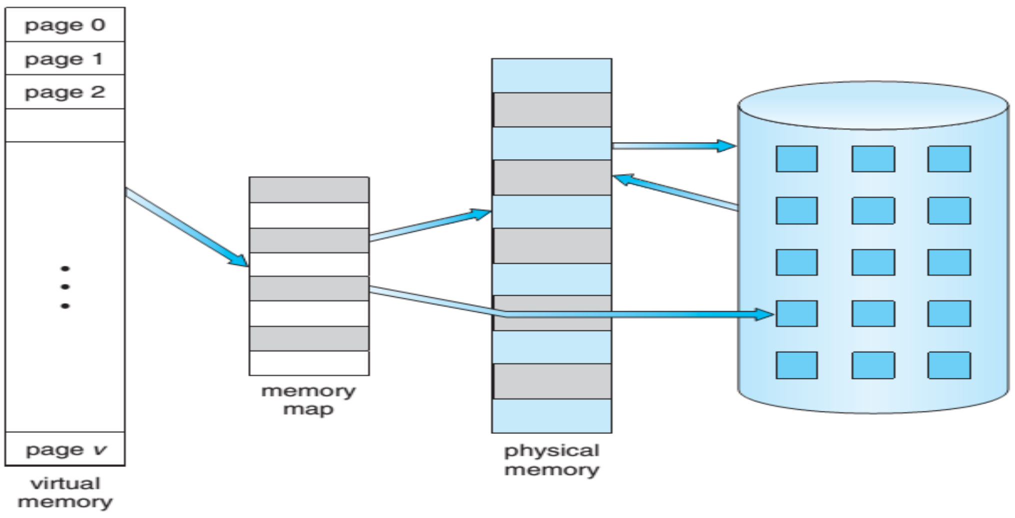
Problem with inverted page table

- 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.
- Systems that use inverted page tables have difficulty implementing shared memory.

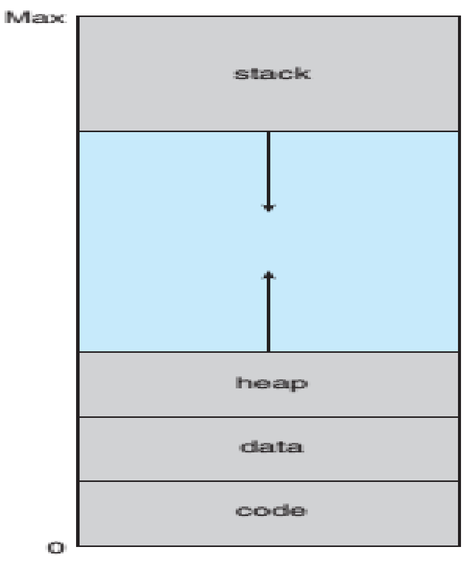
VIRTUAL MEMORY

INTRODUCTION

- Virtual Memory is a technique that allows the execution of processes that are not completely in Main-memory.
- 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.
- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for a large ***virtual*** address space.
- 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 it will not increase the Response time or Turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster. This process will benefit both system and user.



Virtual Address Space of a process refers to the logical (or virtual) view of how a process is stored in memory. The below figure shows a virtual address space:

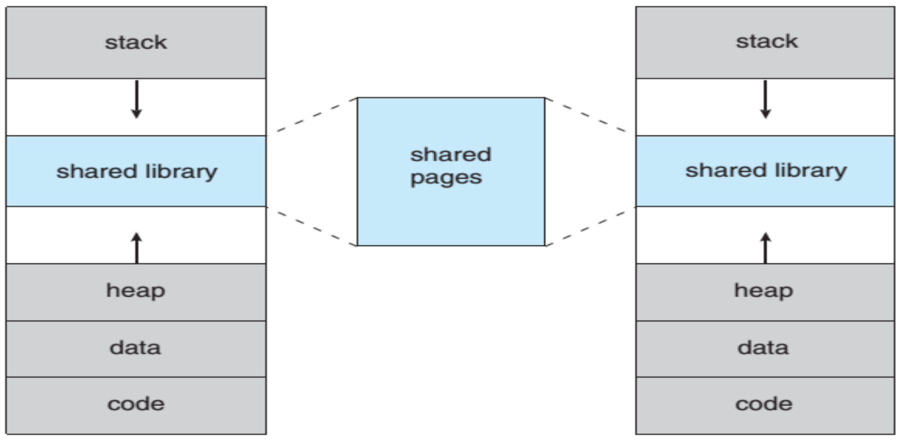


- In the above figure, a process begins at a certain logical address (say address 0) and exists in contiguous memory.
- Physical memory organized in page frames and the physical page frames assigned to a process may not be contiguous.
- **Memory Management Unit** (**MMU**) maps logical pages to physical page frames in Main memory.
- In the above figure, we allow the heap to grow upward in memory as it is used for dynamic memory allocation.
- We allow for the stack to grow downward in memory through successive function calls.
- The large blank space (i.e. 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. 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.

<u>Shared Library using Virtual Memory</u>

System libraries can be shared by several processes through mapping of the shared object into a virtual address space.

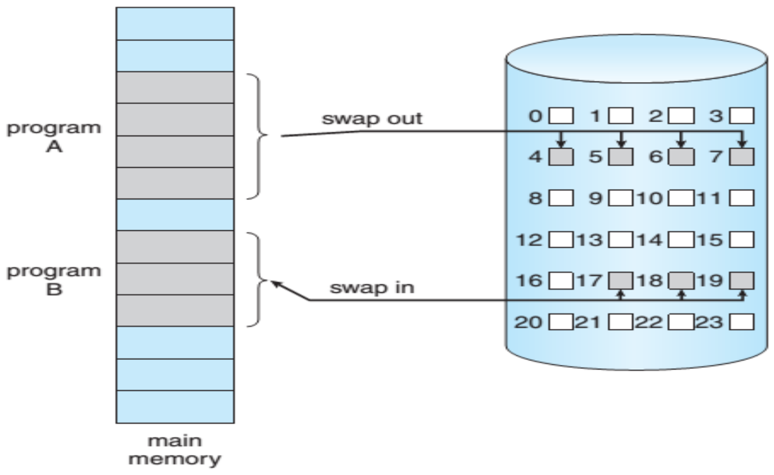
- Each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.
- A library is mapped read-only into the space of each process that is linked with it.



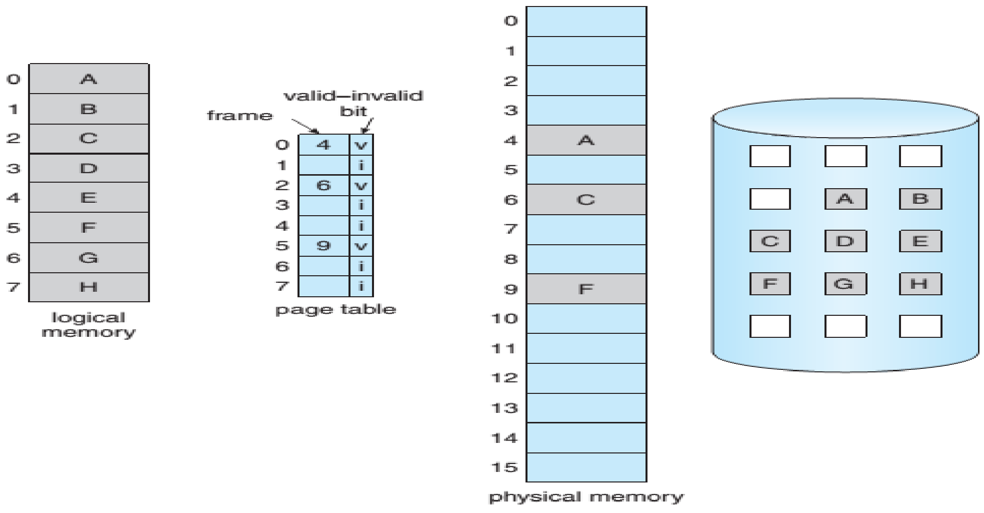
- Two or more processes can communicate through the use of shared memory.
- Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes sharing this region consider it is part of their virtual address space, yet the actual physical pages of memory are shared.
- Pages can be shared during process creation with the fork( ) system call, thus speeding up process creation.

DEMAND PAGING

With Demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are never loaded into physical memory.



- A demand-paging system is similar to a paging system with swapping, where processes reside in secondary memory (i.e. disk).
- Demand paging uses the concept of **Lazy Swapper** or **Lazy Pager**. A lazy swapper or pager never swaps a page into memory unless that page will be needed.
- **Valid–Invalid** bit is used to distinguish between the pages that are in memory and the pages that are on the disk.
- When this bit is set to "valid," the associated page is both legal and is in main memory.
- If the bit is set to "invalid," the page either is not valid (i.e. 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 main memory is set to valid, but the page-table entry for a page that is not currently in main memory is either marked as invalid or contains the address of the page on disk.

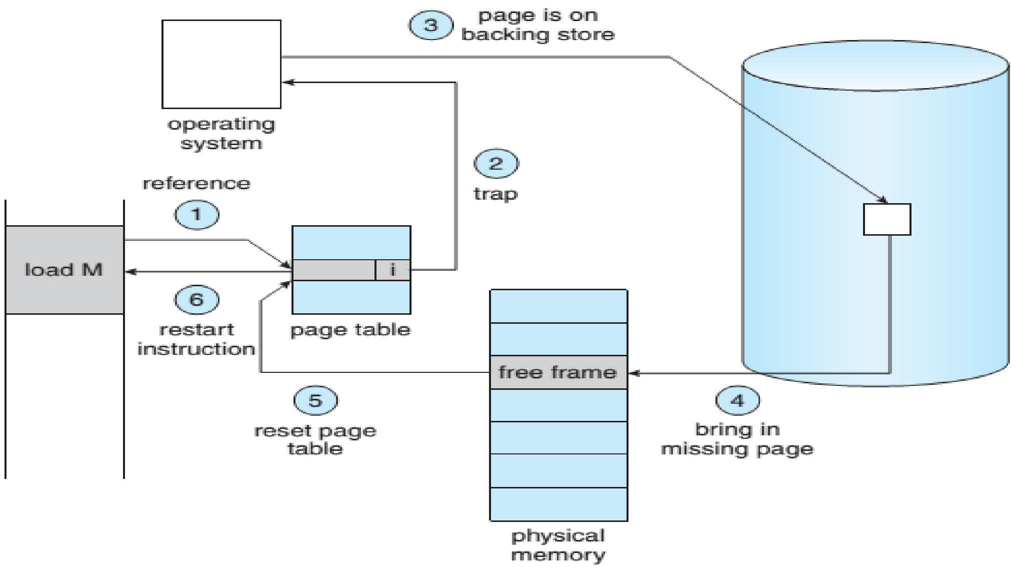


Note: The process executes and accesses pages that are **Main Memory Resident** then the execution proceeds normally.

PAGE FAULT

Access to a page marked invalid causes a **Page Fault**. The paging hardware, in translating the address through the page table will notice that the invalid bit is set that causes a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

Procedure for Handling Page Fault



1. We check an internal page table kept with the **Process Control Block** 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 (Example: by taking one from the free-frame list).
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 also the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page in main memory.

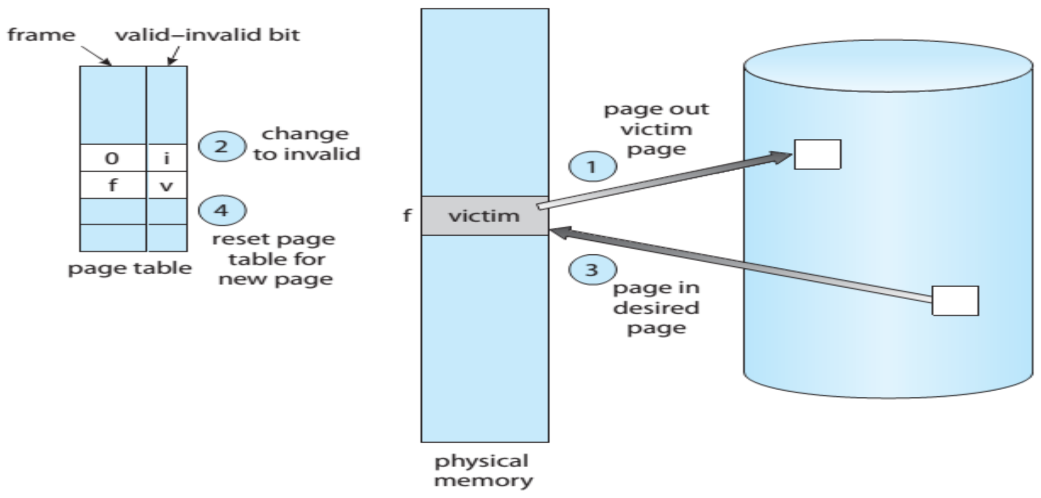
Modify bit or Dirty bit

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 to indicate that the page is no longer in main memory.
- We can now use the freed frame to hold the page for which the process faulted.

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 and change the page table and frame table.
3. Read the desired page into newly freed frame and change the page table and frame table.
4. Continue the user process from where the page fault occurred.



Each page or frame has a modify bit associated with it in the hardware.

- The modify bit for a page is set by the hardware whenever any byte in the page has been modified.
- When we select a page for replacement, we examine its modify bit.
- If the bit is set, the page has been modified since it was **read in** from the disk. Hence we must write the page to the disk.
- If the modify bit is not set, the page has ***not*** been modified since it was read into memory. Hence there is no need for write the memory page to the disk, because it is already there.

Note: To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. As the number of frames available increases, the number of page faults decreases.

Pure Demand Paging

- System can start executing a process with ***no*** pages in memory.
- When the operating system sets the instruction pointer to the first instruction of the process and that process is not resides in main memory then the process immediately faults for the page.
- After this page is brought into memory, the process continues to execute and 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 called **Pure Demand Paging**.
- Pure Demand Paging never brings a page into memory until it is required.

Hardware support for Demand Paging

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table** has the ability to mark an entry as invalid through a valid–invalid bit or a special value of protection bits.

- **Secondary Memory** holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device and the section of disk used for this purpose is known as **Swap Space**.

PAGE REPLACEMENT ALGORITHMS

There is several Page Replacement Algorithms are in use:

1. FIFO Page Replacement Algorithm
2. Optimal Page Replacement Algorithm
3. LRU Page Replacement Algorithm

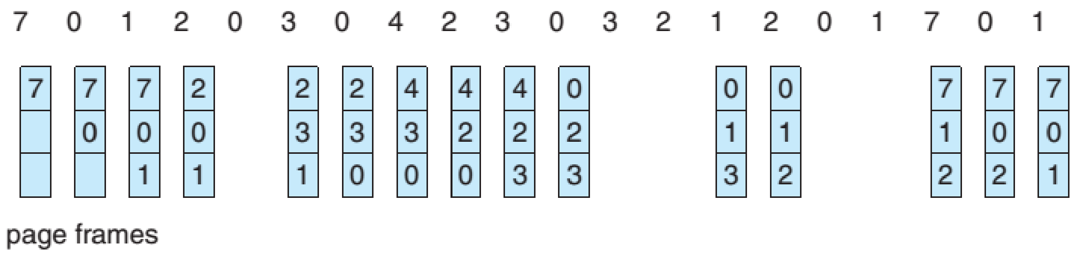
First-In-First-Out Page Replacement Algorithm

FIFO algorithm associates with time of each page when it was brought into main memory.

- When a page must be replaced, the oldest page is chosen.
- We can create a FIFO queue to hold all pages in memory.
- We replace the page at the **Head** of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.

Example: Consider the below reference string and memory with three frames

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



- First **3-references** (7, 0, 1) cause **3-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.
- By the end, there are **Fifteen** page faults altogether.

Problem: Belady's Anomaly

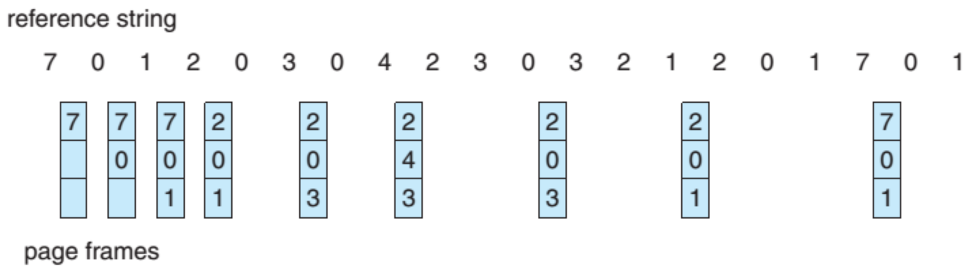
Belady's Anomaly states that: the page-fault rate may ***increase*** as the number of allocated frames increases. Researchers identifies that Belady's anomaly is solved by using Optimal Replacement algorithm.

Optimal Page Replacement Algorithm (OPT Algorithm)

- It will never suffer from Belady’s anomaly.
- OPT states that: Replace the page that will not be used for the longest period of time.
- OPT guarantees the lowest possible page fault rate for a fixed number of frames.

Example: Consider the below reference string and memory with three frames

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



- The first **3-references** cause faults that fill the **3-empty** frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference number 18, whereas page 0 will be used at 5 and page 1 at 14.
- The reference to page 3 replaces page 1 because page 1 will be the last of the three pages in memory to be referenced again.
- At the end there are only **9-page faults** by using optimal replacement algorithm which is much better than a FIFO algorithm with **15-page faults**.

Note: No replacement algorithm can process this reference string in **3-frames** with fewer than **9-faults**.

Problem with Optimal Page Replacement algorithm

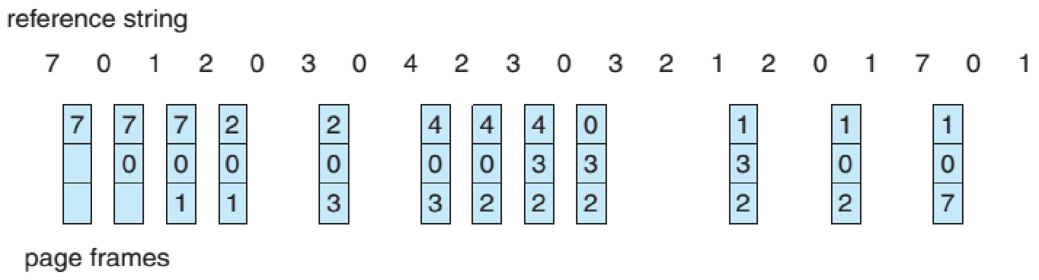
The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. The optimal algorithm is used mainly for comparison studies (i.e. performance studies).

LRU Page Replacement Algorithm

In LRU algorithm, the page that has **not** been used for the **longest** period of time will be replaced (i.e.) we are using the recent past as an approximation of the near future.
LRU replacement associates with each page the time of that page’s last use.

Example: Consider the below reference string and memory with three frames

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



- The first five faults are the same as those for optimal replacement.
- When the reference to page 4 occurs LRU replacement sees that out 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.
- The total number of page faults with LRU is 12 which is less as compared to FIFO.

LRU can be implemented in two ways: **1. Counters** and **2. Stack**

Counters

- Each page-table entry is associated with 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. This scheme requires a search of the page table to find the LRU page and a write to memory to the time-of-use field in the page table for each memory access.
- The times must also be maintained when page tables are changed due to CPU scheduling.

Stack

LRU replacement is implemented by keeping a stack of page numbers.

- Whenever a page is referenced, it is removed from the stack and put on the top.
- In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.
- Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a Head pointer and a Tail pointer.
- Removing a page and putting it on the top of the stack then requires changing six pointers at worst.
- Each update is a little more expensive, but there is no search for a replacement.
- The tail pointer points to the bottom of the stack, which is the LRU page.
- This approach is particularly appropriate for software or microcode implementations of LRU replacement.

ALLOCATION OF FRAMES

There are different approaches used for allocation of frames:

1. Minimum Number of Frames
2. Allocation Algorithms
3. Global versus Local Allocation

Minimum Number of Frames

Allocation is based on minimum number of frames per process is defined by the computer architecture. The maximum number is defined by the amount of available physical memory. We must also allocate at least a minimum number of frames.

- One reason for allocating at least a minimum number of frames involves performance.
- When a page fault occurs before an executing instruction is complete, the instruction must be restarted.
- As the number of frames allocated to each process decreases, the page-fault rate increases and slows the process execution.
- Hence the process must have enough frames to hold all the different pages that any single instruction can reference.

Allocation Algorithms

There are two algorithm are used: Equal allocation and Proportional allocation.

Equal Allocation

- It splits ***m*** frames among ***n*** processes is to give everyone an equal share, ***m/n*** frames.
- Example: If there are 93 frames and five processes, each process will get 18 frames (93/5=18). The **3-leftover** frames can be used as a free-frame buffer pool.

Problem with Equal Allocation

Consider a system with a **1-KB** frame size and two processes P1 and P2.

- Process **P1** is of **10 KB** and process **P2** is of **127 KB** are the only two processes running in a system with **62** free frames.
- Now if we apply Equal allocation then both P1 and P2 will get **31** frames.
- It does not make sense to give P1 process to **31** frames where its maximum use is **10** frames and other **21** frames are **wasted**.

Proportional Allocation

- In this algorithm we allocate available memory to each process according to its size.
- Let the $\mathbf{s_i}$ be the size of the virtual memory for process $\mathbf{P_i}$ then $\mathbf{S = \sum s_i}$
- If the total number of available frames is m, we allocate $\mathbf{a_i}$ frames to process **Pi**, where $\mathbf{a_i}$ is approximately: $\mathbf{a_i = s_i/S \times m}$.
- We must adjust each $\mathbf{a_i}$ to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding **m**.

Example: With proportional allocation, we would split **62** frames between **2-processes**, one of **10** pages and one of **127** pages, by allocating **4** frames for P1 and **57** frames for P2.

$$\mathbf{P1 \text{-}> 10/137 \times 62 \approx 4}$$

$$\mathbf{P2 \text{-}> 127/137 \times 62 \approx 57}$$

Now two processes share the available frames according to their “needs,” rather than equally.

Global versus Local Allocation

With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **Global Replacement** and **Local 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 (i.e.) one process can take a frame from another process.
- Local replacement requires that each process select from only its own set of allocated frames.

Example: Consider an allocation scheme wherein we allow High-priority processes to select frames from low-priority processes for replacement.

- A process can select a replacement from its own frames or the frames of any lower-priority process.
- This approach allows a High-priority process to increase its frame allocation at the expense of a low-priority process.
- With a local replacement, 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.

THRASHING

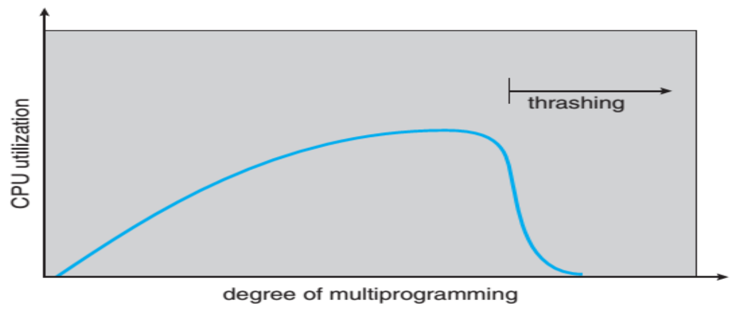
A process is thrashing if it is spending more time for paging than executing.

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution.
- We should then page out its remaining pages, freeing all its allocated frames.
- This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.
- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault and the process must replace some page.
- Since all of 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 by replacing pages that it must bring back in immediately.
- This high paging activity is called **Thrashing**.

Cause of Thrashing

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.

- If a global page-replacement algorithm is used then it replaces pages without regard to the process to which the pages are belongs to.
- 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.
- These processes need those pages which have been faulted earlier so they also fault taking frames from other processes.
- These faulting processes must use the paging device to swap pages in and out.
- As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.
- The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming by introducing new process in to the system again.
- 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 and system throughput decreases. 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.



Consider the above figure that show how thrashing will occur:

- As the degree of multiprogramming increases, CPU utilization also increases until a maximum is reached.
- If the degree of multiprogramming is increased even further then thrashing occurs and CPU utilization drops sharply.
- At this point, we must stop thrashing and increase the CPU utilization by decreasing the degree of multiprogramming.

Solutions to Thrashing

1. Local Replacement Algorithm (or) Priority Replacement Algorithm
2. Locality Model

Local Replacement Algorithm

- With local replacement, if one process starts thrashing, it cannot steal frames from another process.
- Local replacement Algorithm limits thrashing but it cannot avoid thrashing entirely.
- If processes are thrashing, they will be paging device queue most of the time.
- The average service time for a page fault will increase because of the longer average queue for the paging device.
- Thus, the effective access time will increase even for a process that is not thrashing.

Locality Model

- 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.

Example: When a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.
Note: Localities are defined by the program structure and its data structures.

Suppose we allocate enough frames to a process to accommodate its current locality.

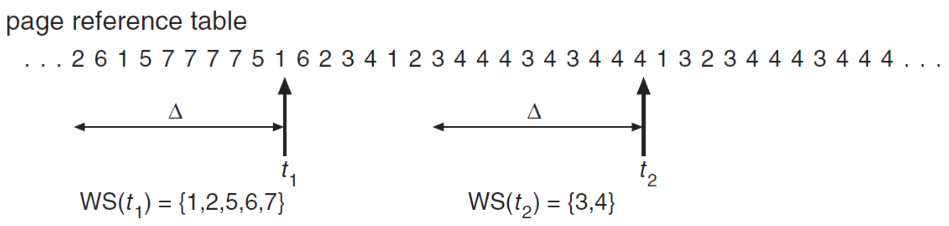
- It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

Working-Set Model

The **working-set model** is based on the assumption of locality.

- This model uses a parameter Δ to define the **working-set window**.
- The idea is to examine the most recent Δ page references.
- The set of pages in the most recent Δ 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 Δ time units after its last reference.
- Thus, the working set is an approximation of the program's locality.

Consider the below figure given the sequence of memory references if Δ = 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 Δ.
- If Δ is too small, it will not encompass the entire locality;
- If Δ is too large, it may overlap several localities.

$$D = \sum WSS_i,$$

D is the total demand for frames
WSS_i = Working set size of process P_i.
The process i needs WSS_i frames.

D>m:

- 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.
- The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size.
- If there are enough extra frames, another process can be initiated.
- If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes.
- The suspended process can be restarted later.
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

Data structures used to implement the Banker's algorithm

Consider the system with **n** number of processes and m number of resource types:

- **Available_m:** A vector of length m indicates the number of available resources of each type.
- **$\text{Max}_{n \times m}$:** An n × m matrix defines the maximum demand of each process.
- **$\text{Allocation}_{n \times m}$:** An n × m matrix defines the number of resources of each type currently allocated to each process.
- **$\text{Need}_{n \times m}$:** An n × m matrix indicates the remaining resource need of each process.

$$\textbf{Need[i][j] = Max[i][j]−Allocation[i][j].}$$

- **Available[j] = k** means then k instances of resource type Rj are available.
- **Max[i][j] = k** means process Pi may request at most k instances of resource type Rj.
- **Allocation[i][j]=k** means process Pi is currently allocated k instances of resource type Rj.

Need[i][j]=k means process Pi may need k more instances of resource type Rj to complete its task.

Safety algorithm

Safety algorithm finds out whether the system is in safe state or not. The algorithm can be described as follows:

1. Let **Work** and **Finish** be vectors of length m and n, respectively. We initialize

   Work = **Available**

   Finish[i] = **false** for i = 0, 1, ..., n − 1.
2. Find an index i such that both

   Finish[i] == **false**

   Need_i ≤ **Work**

   If no such i exists, go to step 4.
3. **Work** = **Work** + **Allocation_i**

   Finish[i] = **true**

   Go to step 2.
4. If **Finish**[i] == **true** for all i, then the system is in a safe state.

Resource-Request Algorithm

When a request for resources is made by process Pi, the following actions are taken:

1. If **Request_i** ≤ **Need_i**, go to step 2.

   Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If **Request_i** ≤ **Available,** go to step 3.

   Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

   Available = **Available**– **Request_i** ;

   Allocation_i = **Allocation_i** + **Request_i**;

   Need_i = **Need_i** – **Request_i** ;
4. If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources.

   If the new state is unsafe, then Pi must wait for **Request_i** and the old resource-allocation state is restored.

Example: Apply Banker Algorithm on the following snapshot of the system.

	Allocation	Max	Available
Process	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Finding Need Matrix

	Max			Allocation			**Need**		
	A	B	C	A	B	C	**A**	**B**	**C**
P0	7	5	3	0	1	0	**7**	**4**	**3**
P1	3	2	2	2	0	0	**1**	**2**	**2**
P2	9	0	2	3	0	2	**6**	**0**	**0**
P3	2	2	2	2	1	1	**0**	**1**	**1**
P4	4	3	3	0	0	2	**4**	**3**	**1**

Safety algorithm

Step 1: Work = available ➔ Work = [3 3 2]

Finish[5]== false

Finish	F	F	F	F	F
	0	1	2	3	4

For P_0: Finish[0]=false;

Need$_0$ $\le$ **Work** ➔ [7 4 3] $\le$ [3 3 2] **(Condition fails, go to next Process and check again)**

For P_1: Finish[1]=false;

Need$_1$ $\le$ **Work** ➔ [1 2 2] $\le$ [3 3 2] **(Condition satisfied, then goto step 3)**

Work = **Work** + **Allocation$_1$** ➔ [3 3 2] + [2 0 0]

Work = [5 3 2]

Finish[1] = **true**

For P_2: Finish[2]=false;

Need$_2$ $\le$ **Work** ➔ [6 0 0] $\le$ [5 3 2] **(Condition fails, go to next Process and check again)**

For P_3: Finish[3]=false;

Need$_3$ $\le$ **Work** ➔ [0 1 1] $\le$ [5 3 2] **(Condition satisfied, then goto step 3)**

Work = **Work** + **Allocation$_3$** ➔ [5 3 2] + [2 1 1]

Work = [7 4 3]

Finish[3] = **true**

For P_4: Finish[4]=false;

Need$_4$ $\le$ **Work** ➔ [4 3 1] $\le$ [7 4 3] **(Condition satisfied, then goto step 3)**

Work = **Work** + **Allocation$_4$** ➔ [7 4 3] + [0 0 2]

Work = [7 4 5]

Finish[4] = **true**

Now we will go back to P_0

For P_0: Finish[0]=false;

Need$_0$ ≤ Work ➔ **[7 4 3] ≤ [7 4 5]** **(Condition satisfied, then goto step 3)**

Work = **Work** + **Allocation$_0$➔ [7 4 5] +** **[0 1 0]**

Work = [7 5 5]

Finish[0] = **true**

For P_2: Finish[2]=false;

Need$_2$≤ Work ➔ **[6 0 0] ≤ [7 5 5]** **(Condition satisfied, then goto step 3)**

Work = **Work** + **Allocation$_2$➔ [7 5 5] +** **[3 0 2]**

Work = [10 5 7]

Finish[2] = **true**

At this point all the process has acquired their maximum resources and all the values in Finish array also true.

Finish	T	T	T	T	T
	0	1	2	3	4

The system is in safe state. The safe sequence can be given as < P_1 , P_3 , P_4 , P_0 , P_2 ,>

Now the system is in safe state we can grant the request asked by the processes in the sequence.

To allocate the resources Bankers Algorithm uses Resource Request Algorithm.

Resource-Request Algorithm

Example: If *P*$_1$ Requests the resources as the vector [1,0,2].

Step 1: **Request$_1$ ≤ Need$_1$** ➔ [1 0 2] ≤ [1 2 2] (Condition satisfied goto step2)

Step 2: **Request$_i$ ≤ Available** ➔ [1 0 2] ≤ [3 3 2] (Condition satisfied goto step3)

Step 3: **Available** = **Available– Request$_1$** ; ➔ [3 3 2] - [1 0 2] =[2 3 0]
Available = [2 3 0]

Allocation$_1$ = **Allocation$_1$** + **Request$_1$**; ➔ [2 0 0] + [1 0 2] = [3 0 2]
Allocation$_1$ = [3 0 2]

Need$_1$ = **Need$_1$** – **Request$_1$** ; ➔ [1 2 2] – [1 0 2] = [0 2 0]
Need$_1$ = [0 2 0]

Now replace these values in the corresponding matrices, we get the following snapshot of the system.

	Max			Allocation			**Need**			**Available**		
	A	B	C	A	B	C	**A**	**B**	**C**	**A**	**B**	**C**
P0	7	5	3	0	1	0	**7**	**4**	**3**	**2**	**3**	**0**
P1	**3**	**2**	**2**	**3**	**0**	**2**	**0**	**2**	**0**			
P2	9	0	2	3	0	2	**6**	**0**	**0**			
P3	2	2	2	2	1	1	**0**	**1**	**1**			
P4	4	3	3	0	0	2	**4**	**3**	**1**			