Memory Management

- Memory Management consists of many tasks, including
  - Being aware of what parts of the memory are in use and which parts are not
  - Allocating memory to processes when they request it and de-allocating memory when a process releases it
  - Moving data from memory to disc, when the physical capacity becomes full, and vice versa.

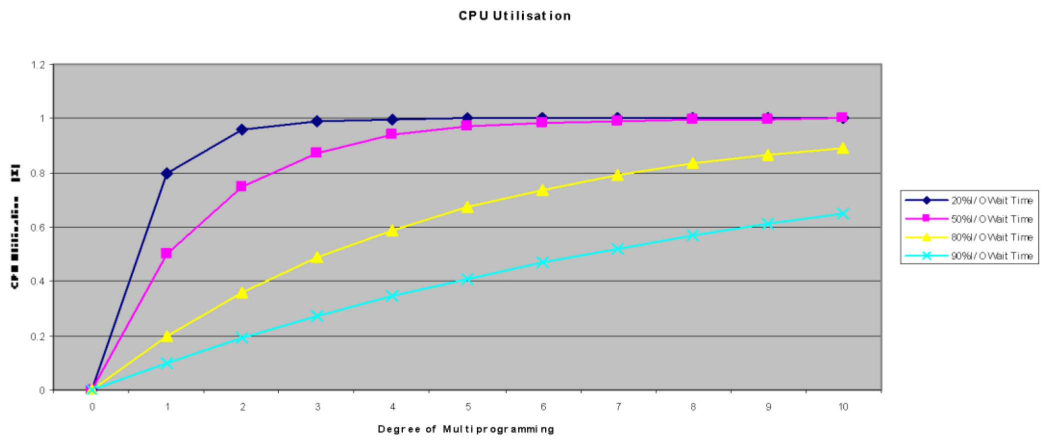
Monoprogramming without Swapping & Paging

- Only allow a single process in memory and only allow one process to run at any one time
  - Very Simple
  - No swapping of processes to disc when we run out of memory
  - No problems in having separate processes in memory
- Even this simple scheme has its problems.
  - We have not yet considered the data that the program will operate upon
  - Process must be self contained
    - e.g. drivers in every process
  - Operating system can be seen as a process – so we have two anyway
- Additional Problems
  - Monoprogramming is unacceptable as multi-programming is expected
  - Multiprogramming makes more effective use of the CPU
  - Could allow only a single process in memory at any one time but allow multi-programming
  - i.e. swap out to disc
  - Context switch would take time

Modeling Multiprogramming

- Assumption that multiprogramming can improve the utilization of the CPU – Is this true?
  - Intuitively it is
  - But, can we model it?
- Probabilistic model
  - A process spends *p* percent of its time waiting for I/O

- There are n processes in memory
- The probability that all n processes are waiting for I/O (CPU is idle) is p^n
- The CPU utilization is then given by
    - CPU Utilization = $1 - p^n$



- With an I/O wait time of 20%, almost 100% CPU utilization can be achieved with four processes
- I/O wait time of 90% then with ten processes, we only achieve just above 60% utilization
- The more processes we run, the better the CPU utilization

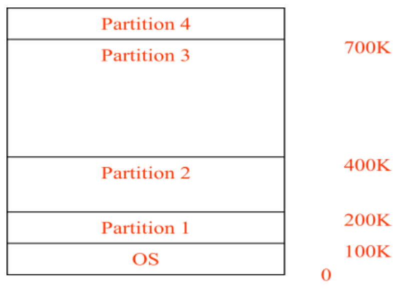
– Model assumes that all the processes are independent. This is not true

More complex models could be built using queuing theory but we can still use this simplistic model to make approximate predictions

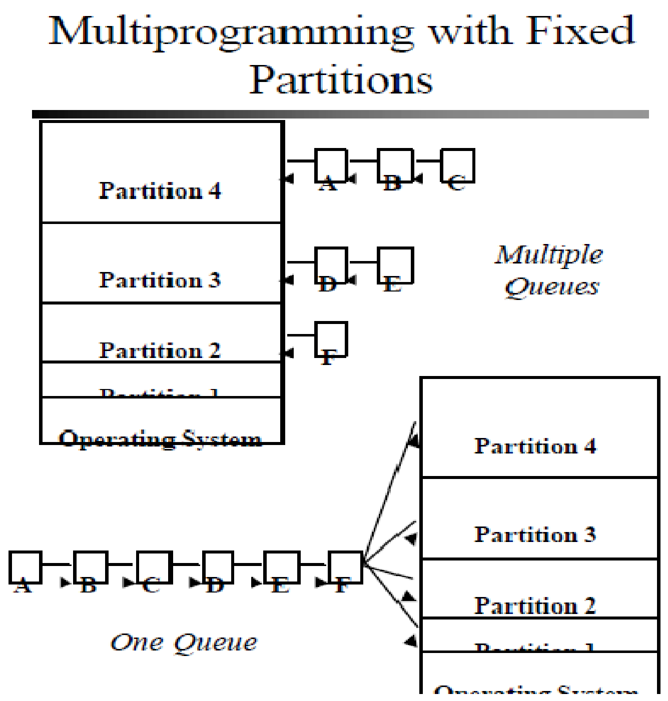
- Example of model use
    - Assume a computer with one megabyte of memory
    - The operating system takes up 200K, leaving room for four 200K processes
    - If we have an I/O wait time of 80% then we will achieve just under 60% CPU utilization
    - If we add another megabyte of memory, it allows us to run another five processes
    - We can now achieve about 86% CPU utilization
    - If we add another megabyte of memory (fourteen processes) we will find that the CPU utilization will increase to about 96%.

Multiprogramming with Fixed Partitions

- Accept that multiprogramming is a good idea
- One method is to divide the memory into fixed sized partitions
  - Partitions can be of different sizes but their size remain fixed



- Memory divided into four partitions
- When job arrives it is placed in the input queue for the smallest partition that will accommodate it



- Drawbacks
  - As the partition sizes are fixed, any space not used by a particular job is lost.
  - It may not be easy to state how big a partition a particular job needs.
  - If a job is placed in (say) queue three it may be prevented from running by other jobs waiting (and using) that partition.

- Just have a single input queue where all jobs are held
- When a partition becomes free we search the queue looking for the first job that fits into the partition
- Alternative search strategy
    - Search the entire input queue looking for the largest job that fits into the partition
- Do not waste a large partition on a small job but smaller jobs are discriminated against
- Have at least one small partition or ensure that small jobs only get skipped a certain number of times.

<u>Relocation and Protection</u>

- Introducing multiprogramming gives two problems
    - Relocation: When a program is run it does not know in advance what location it will be loaded at. Therefore, the program cannot simply generate static addresses (e.g. from jump instructions). Instead, they must be made relative to where the program has been loaded
    - Protection: Once you can have two programs in memory at the same time there is a danger that one program can write to the address space of another program. This is obviously dangerous and should be avoided
- Solution to solve both relocation and protection
    - Have two registers (called the base and limit registers)
    - The base register stores the start address of the partition
    - The limit register holds the length of the partition
    - Additional benefit of this scheme is moving programs in memory

<u>Swapping</u>

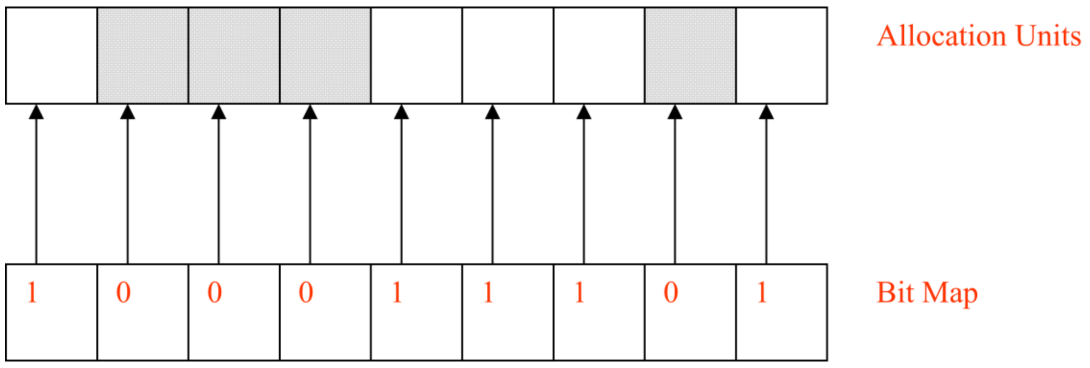
- Fixed partitions becomes ineffective when we have more processes than we can fit into memory at one time (e.g. when timesharing)
- Solution: Hold some of the processes on disc and swap processes between disc and main memory as necessary.
- Because we are swapping processes between memory and disc does not stop us using fixed partition sizes
- But, the reason we are having swap processes out to disc is because memory is a scare resource and as fixed partitions can be wasteful of memory it would be better to implement a more efficient scheme
- Obvious step is to use variable partition sizes
    - That is a partition size can change as the need arises

- Variable partitions
    - The number of partitions vary
    - The sizes of the partitions vary
    - The starting addresses of the partitions vary.
- Makes for a more effective memory management system but it makes the process of maintaining the memory much more difficult
    - As memory is allocated and deallocated holes will appear in the memory (fragmentation)
        - Eventually holes will be too small to have a process allocated to it
        - We could shuffle the memory downwards (memory compaction) but this is inefficient
    - If processes are allowed to dynamically request more memory what happens if a process requests extra memory such that increasing its partition size is impossible without it having to overwrite another partitions memory
        - Wait until memory is available that the process is able to grow into it?
        - Terminate the process?
        - Move the process to a hole in memory that is large enough to accommodate the growing process?
        - Only realistic option is the last one but very inefficient
        - None of the above proposed solutions are ideal so it would seem a good idea to allocate more memory than is initially required
        - Most processes will have two growing data segments
        - Stack
        - Heap

Memory Management with Bit Maps

Memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map

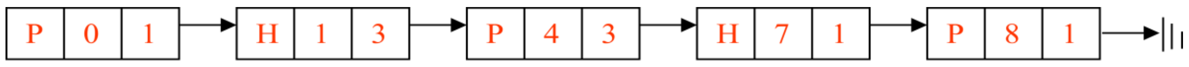
If the bit is zero, the memory is free. If the bit is one, then the memory is being used



- Main decision is the size of the allocation unit
  - The smaller the allocation unit, the larger the bit map has to be
  - But a larger allocation unit could waste memory as we may not use all the space allocated in each allocation unit.
- Another problem comes when we need to allocate memory to a process
  - Assume the allocation size is 4 bytes
  - If a process requests 256 bytes of memory, we must search the bit map for 64 consecutive zeroes (256/4 = 64)
  - Slow operation, therefore bit maps are not often used

Memory Management with Linked Lists

- Free and allocated memory can be represented as a linked list
- The memory shown on the bit map slide can be represented as a linked list as follows



- Each entry in the list holds the following data
  - P or H : for Process or Hole
  - Starting segment address
  - The length of the memory segment
  - The next pointer is not shown but assumed to be present
- In the list above, processes follow holes and vice versa
- But, processes can be next to each other and we need to keep them as separate elements in the list
- Consecutive holes can always be merged into a single list entry

Memory Management with Buddy System

- If we keep a list of holes sorted by their size, we can make allocation to processes very fast as we only need to search down the list until we find a hole that is big enough
- The problem is that when a process ends the maintenance of the lists is complicated
    - In particular, merging adjacent holes is difficult as the entire list has to be searched in order to find its neighbors
    - The Buddy System is a memory allocation that works on the basis of using binary numbers as these are fast for computers to manipulate
- Lists are maintained which stores lists of free memory blocks of sizes 1, 2, 4, 8,…, n, where *n* is the size of the memory (in bytes). This means that for a one megabyte memory we require 21 lists.
- If we assume we have one megabyte of memory and it is all unused then there will be one entry in the 1M list; and all other lists will be empty.
- The buddy system is fast as when a block size of 2k bytes is returned only the 2k list has to be searched to see if a merge is possible
- The problem with the buddy system is that it is inefficient in terms of memory usage. All memory requests have to be rounded up to a power of two
- This type of wastage is known as internal fragmentation. As the wasted memory is internal to the allocated segments
- Opposite is external fragmentation where the wasted memory appears between allocated segments.

Fragmentation

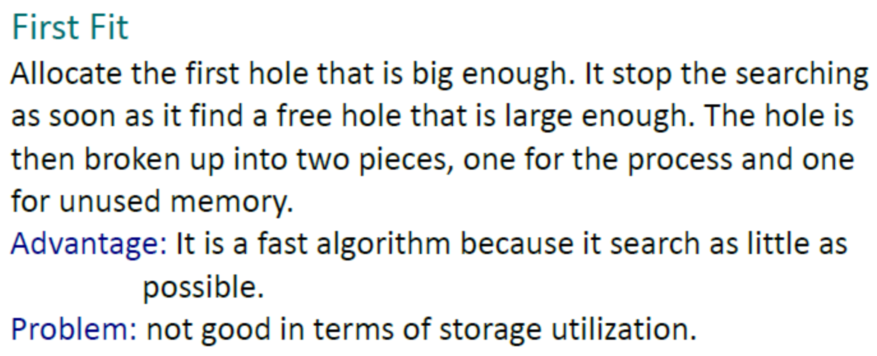
Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.

External Fragmentation: External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.

Internal Fragmentation: Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Allocation of Swap Space

When we need to allocate memory, storing the list in segment address order allows us to implement various strategies.

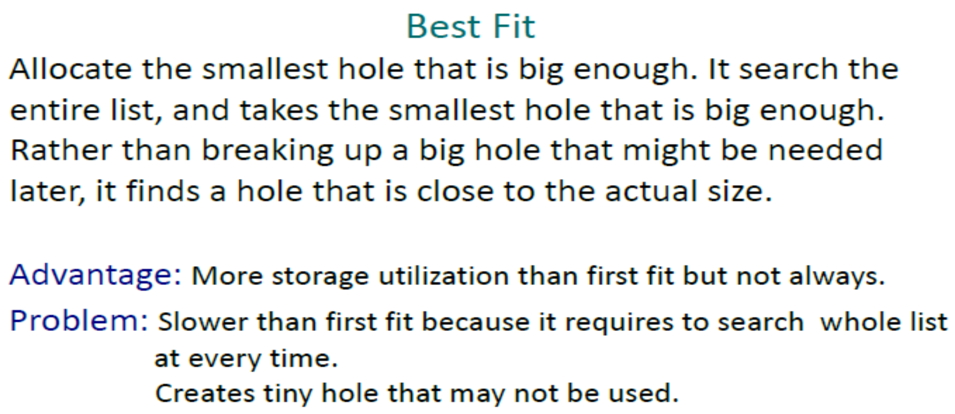


First Fit

Allocate the first hole that is big enough. It stop the searching as soon as it find a free hole that is large enough. The hole is then broken up into two pieces, one for the process and one for unused memory.

Advantage: It is a fast algorithm because it search as little as possible.

Problem: not good in terms of storage utilization.

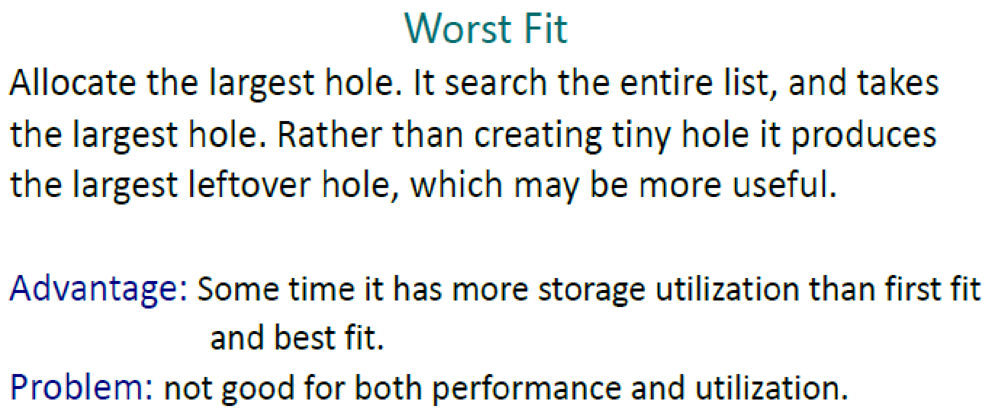


Best Fit

Allocate the smallest hole that is big enough. It search the entire list, and takes the smallest hole that is big enough. Rather than breaking up a big hole that might be needed later, it finds a hole that is close to the actual size.

Advantage: More storage utilization than first fit but not always.

Problem: Slower than first fit because it requires to search whole list at every time.
Creates tiny hole that may not be used.



Worst Fit

Allocate the largest hole. It search the entire list, and takes the largest hole. Rather than creating tiny hole it produces the largest leftover hole, which may be more useful.

Advantage: Some time it has more storage utilization than first fit and best fit.

Problem: not good for both performance and utilization.

Analysis of Swapping Systems

- Swapping allows us to allocate memory to processes when they need it. But what happens when we do not have enough memory?
- In the past, overlays were used
    - Responsibility of the programmer
    - Program split into logical sections (called overlays)

- Only one overlay would be loaded into memory at a time
- Meant that more programs could be running than would be the case if the complete program had to be in memory

- Downsides
    - Programmer had to take responsibility for splitting the program into logical sections
    - Time consuming, boring and open to error

Virtual Memory

- Idea is that that the computer is able to run programs even if the amount of physical memory is not sufficient to allow the program and all its data to reside in memory at the same time
- At the most basic level we can run a 500K program on a 256K machine
- We can also use virtual memory in a multiprogramming environment. We can run twelve programs in a machine that could, without virtual memory, only run four

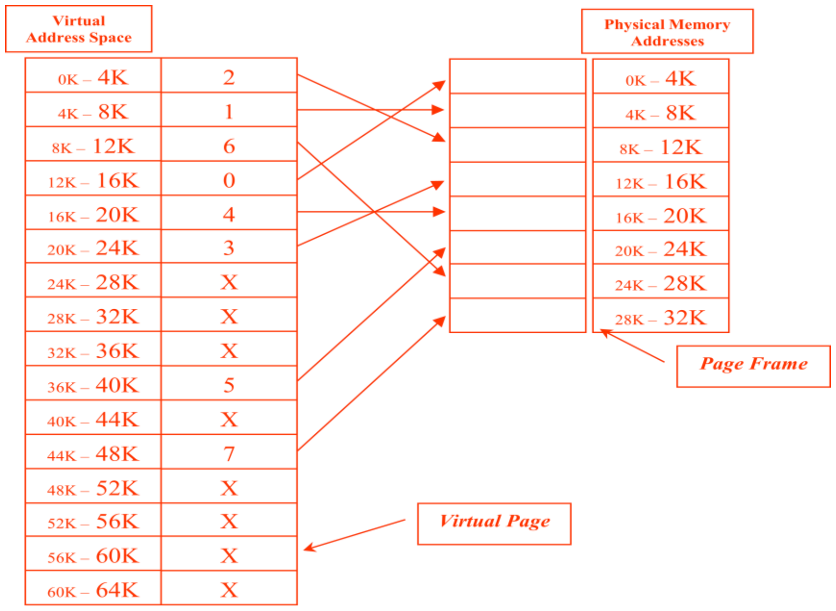
Main Issues in VM Design

- Address mapping
    - How to translate virtual addresses to physical addresses
- Placement
    - Where to place a portion of VM needed by process
- Replacement
    - Which portion of VM to remove when space is needed
- Load control
    - How much of VM to load at any one time
- Sharing
    - How can processes share portions of their VMs
    -

<u>Paging</u>

- In a computer system that does not support virtual memory, when a program generates a memory address it is placed directly on the memory bus which causes the requested memory location to be accessedd

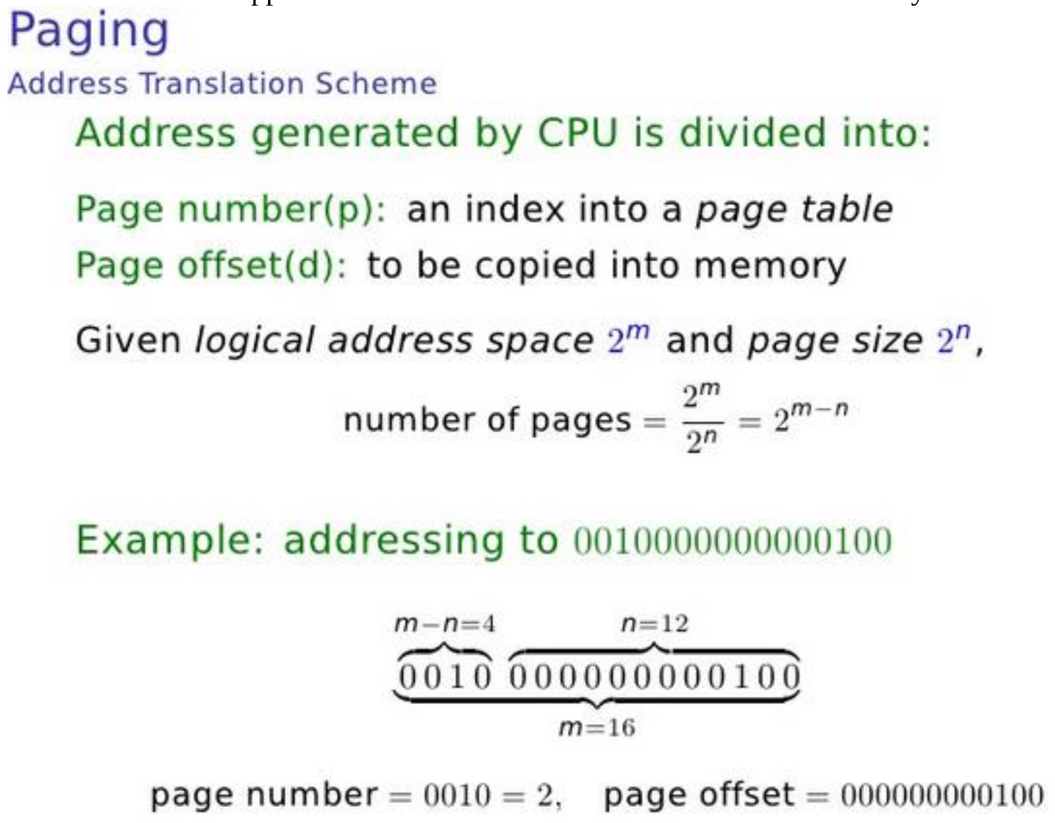
- On a computer that supports virtual memory, the address generated by a program goes via a memory management unit (MMU). This unit maps virtual addresses to physical addresses



- Example
  - Assume a program tries to access address 8192
  - This address is sent to the MMU
  - The MMU recognizes that this address falls in virtual page 2 (assume pages start at zero)
  - The MMU looks at its page mapping and sees that page 2 maps to physical page 6
  - The MMU translates 8192 to the relevant address in physical page 6 (this being 24576)
  - This address is output by the MMU and the memory board simply sees a request for address 24576. It does not know that the MMU has intervened. The memory board simply sees a request for a particular location, which it honors.
- If a virtual memory address is not on a page boundary (as in the above example) then the MMU also has to calculate an offset (in fact, there is always an offset – in the above example it was zero)
- We have not really achieved anything yet as, in effect, we have eight virtual pages which do not map to a physical page
- Each virtual page will have a present/absent bit which indicates if the virtual page is mapped to a physical pag
- What happens if we try to use an unmapped page? For example, the program tries to access address 24576 (i.e. 24K)
  - The MMU will notice that the page is unmapped and will cause a trap to the operating system
  - This trap is called a page fault

- The operating system will decide to evict one of the currently mapped pages and use that for the page that has just been referenced
- The page that has just been referenced is copied (from disc) to the virtual page that has just been freed.
- The virtual page frames are updated.
- The trapped instruction is restarted.

• Example (trying to access address 24576)
- The MMU would cause a trap to the operating system as the virtual page is not mapped to a physical location
- A virtual page that is mapped is elected for eviction (we'll assume that page 11 is nominated)
- Virtual page 11 is mark as unmapped (i.e. the present/absent bit is changed)
- Physical page 7 is written to disc (we'll assume for now that this needs to be done). That is the physical page that virtual page 11 maps onto
- Virtual page 6 is loaded to physical address 28672 (28K)
- The entry for virtual page 6 is changed so that the present/absent bit is changed. Also the 'X' is replaced by a '7' so that it points to the correct physical page
- When the trapped instruction is re-executed it will now work correctly



Paging

Address Translation Scheme

Address generated by CPU is divided into:

Page number(p): an index into a *page table*

Page offset(d): to be copied into memory

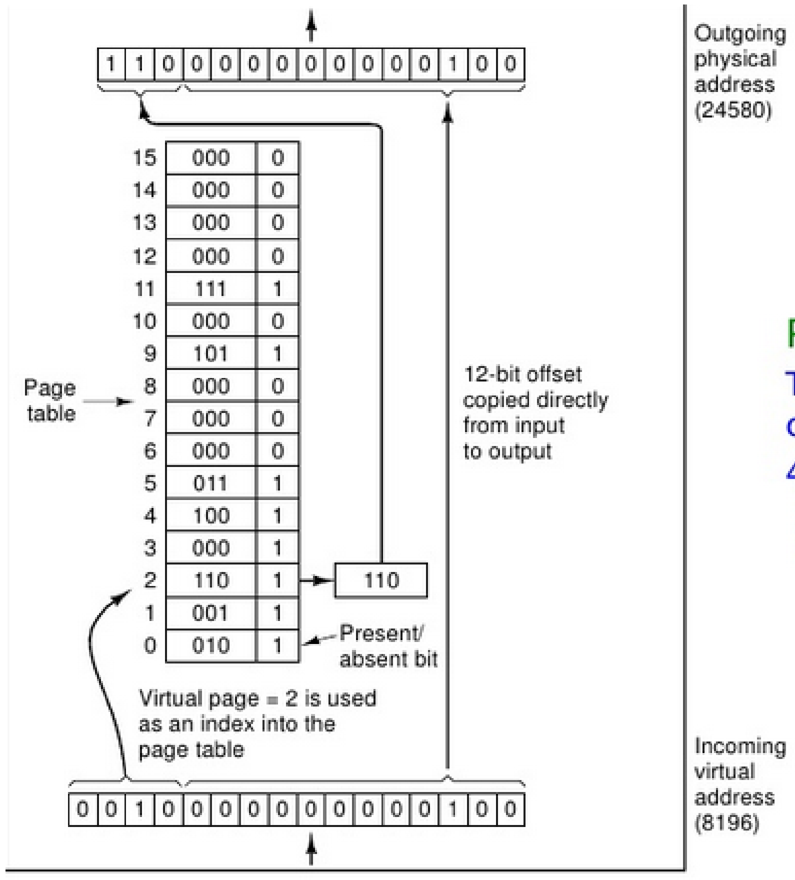
Given *logical address space* 2^m and *page size* 2^n,

$$\text{number of pages} = \frac{2^m}{2^n} = 2^{m-n}$$

Example: addressing to 0010000000000100

$$\underbrace{\overbrace{0010}^{m-n=4}\ \overbrace{000000000100}^{n=12}}_{m=16}$$

page number $= 0010 = 2$, page offset $= 000000000100$



Paging Example

The internal operation of the MMU with 16 4-KB pages

Physical memory:	32K
Virtual memory:	64K

28 / 81

- **Multilevel Paging (Refer class note ,book and go** www.csitnepal.com)
- **Hashed Page Table(go** www.csitnepal.com)
- **Inverted Page Table(go** www.csitnepal.com)

Page Replacement Algorithms

- Choose a mapped page at random
  - Likely to lead to degraded system performance
  - Page chosen has a reasonable chance of being a page that will need to be used again in the near future
- **The Optimal Page Replacement Algorithm**
  - Evict the page that we will not use for the longest period
  - Problem is we cannot look into the future and decide which page to evict
  - But, if we could, we could implement an optimal algorithm

- But, if we cannot implement the algorithm then why bother discussing it?

- In fact, we can implement it, but only after running the program to see which pages we should evict at what point
  We can then use this as a measure to see how other algorithms perform against this ideal

- **The Not-Recently-Used Page Replacement Algorithm**
  - Make use of the referenced and modified bits
  - When a process starts all its page entries are marked as not in memory
  - When a page is referenced a page fault will occur
  - The R (reference) bit is set and the page table entry modified to point to the correct page
  - The page is set to read only. If the page is later written to the M (modified) bit is set and the page is changed so that it is read/write
- Updating the flags in this way allows a simple paging algorithm to be built
- When a process is started up all R and M bits are cleared set to zero
- Periodically (e.g. on each clock interrupt) the R bit is cleared (allows us to recognize which pages have been recently referenced)
- When a page fault occurs (so that a page needs to be evicted), the pages are inspected and divided into four categories based on their R and M bits

Class 0:	Not Referenced, Not Modified
Class 1:	Not Referenced, Modified
Class 2:	Referenced, Not Modified
Class 3:	Referenced, Modified

- The NRU algorithm removes a page at random from the lowest numbered class that has entries in it
- Not optimal algorithm, NRU often provides adequate performance and is easy to understand and implement

- **The First-In, First-Out (FIFO) Page Replacement Algorithm**
  - Maintains a linked list, with new pages being added to the end of the list
  - When a page fault occurs, the page at the head of the list (the oldest page) is evicted
  - Simple to understand and implement but does not lead to good performance as a heavily used page is just as likely to be evicted as a lightly used page.
- **The Second Chance Page Replacement Algorithm**
  - Modification of the FIFO algorithm
  - When a page fault occurs if the page at the front of the linked list has not been referenced it is evicted.
  - If its reference bit is set, then it is placed at the end of the linked list and its reference bit reset
  - In the worst case, SC, operates the same as FIFO
- **The Clock Page Replacement Algorithm**
  - The clock page (CP) algorithm differs from SC only in its implementation
  - SC suffers in the amount of time it has to devote to the maintenance of the linked list
  - More efficient to hold the pages in a circular list and move the pointer rather than move the pages from the head of the list to the end of the list.
- **The Least Recently Used (LRU) Page Replacement Algorithm**
  - Approximate an optimal algorithm by keeping track of when a page was last used
  - If a page has recently been used then it is likely that it will be used again in the near future

    - Therefore, if we evict the page that has not been used for the longest amount of time we can implement a least recently used (LRU) algorithm
    - Whilst this algorithm can be implemented it is not cheap as we need to maintain a linked list of pages which are sorted in the order in which they have been used
    - We can implement the algorithm in hardware
        - The hardware is equipped with a counter (typically 64 bits). After each instruction the counter is incremented
        - Each page table entry has a field large enough to accommodate the counter
        - Every time the page is referenced the value from the counter is copied to the page table field
        - When a page fault occurs the operating system inspects all the page table entries and selects the page with the lowest counter
        - This is the page that is evicted as it has not been referenced for the longest time
- Another hardware implementation of the LRU algorithm is given below.
    - If we have n page table entries a matrix of n x n bits , initially all zero, is maintained
    - When a page frame, k, is referenced then all the bits of the k row are set to one and all the bits of the k column are set to zero
    - At any time the row with the lowest binary value is the row that is the least recently used (where row number = page frame number)
    - The next lowest entry is the next recently used; and so on.
- LRU in Software
    - We cannot, as OS writers, implement LRU in hardware if the hardware does not provide the facilities
    - We can implement a similar algorithm in software
    - Not Frequently Used – NFU associates a counter with each page
    - This counter is initially zero but at each clock interrupt the operating system scans all the pages and adds the R bit to the counter

When a page fault occurs the page with the lowest counter is selected for replacement

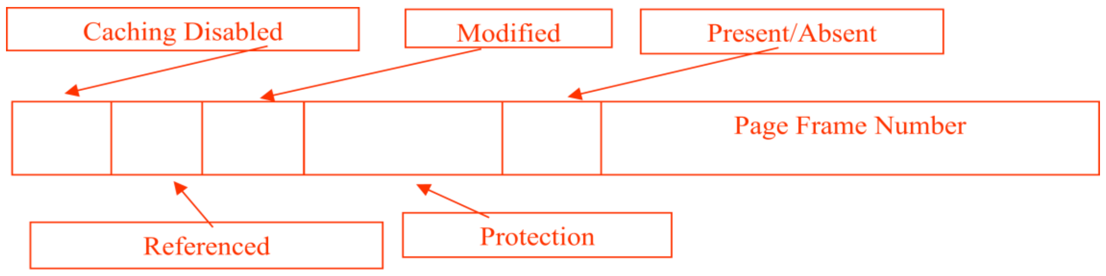
Design Issues for Paging

- Demand Paging
    - The most obvious way to implement a paging system is to start a process with none of its pages in memory
    - When the process starts to execute it will try to get its first instruction, which will cause a page fault
    - Other page faults will quickly follow
    - After a period of time the process should start to find that most of its pages are in memory
    - Known as demand paging as pages are brought into memory on demand
- Working Set
    - The reason that page faults decrease (and then stabilise) is because processes normally exhibit a locality of reference
    - At a particular execution phase of the process it only uses a small fraction of the pages available to the entire process
    - The set of pages that is currently being used is called its working set
    - If the entire working set is in memory then no page faults will occur
    - Only when the process moves onto its next phase will page faults begin to occur again

- If the memory of the computer is not large enough to hold the entire working set, then pages will constantly be copied out to disc and subsequently retrieved
- **This drastically slows a process down and the process is said to be thrashing**

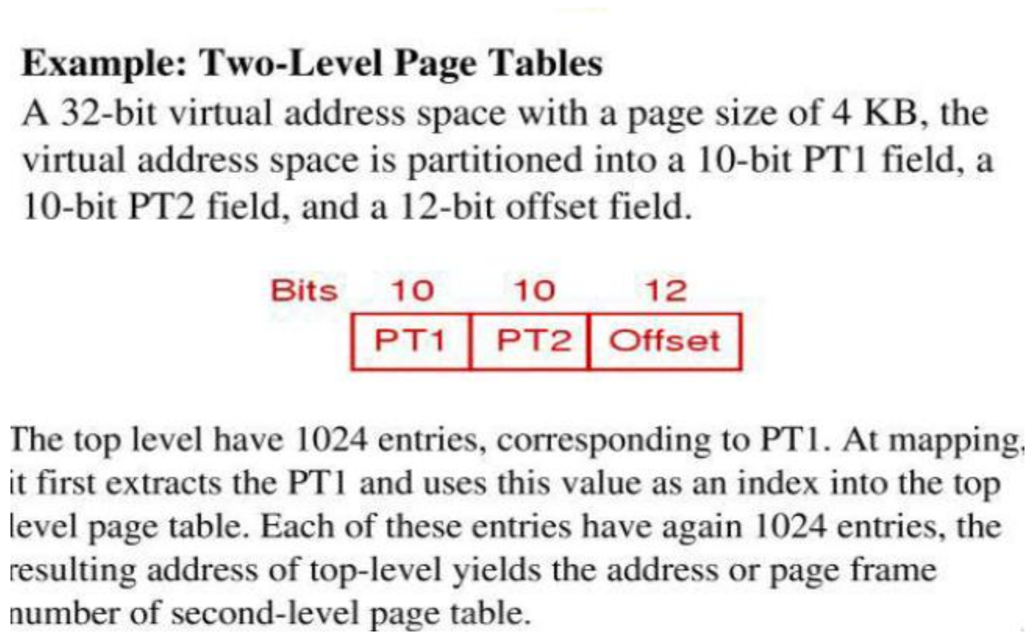
- **Prepaging/Working Set Model**
  - In a system that allows many processes to run at the same time it is common to move all the pages for a process to disc (i.e. swap it out)
  - When the process is restarted we have to decide what to do
  - Do we simply allow demand paging?
  - Or do we move all its working set into memory so that it can continue with minimal page faults?
  - The second option is to be preferred
  - We would like to avoid a process, every time it is restarted, raising page faults
  - The paging system has to keep track of a processes' working set so that it can be loaded into memory before it is restarted.
  - The approach is called the working set model (or prepaging). Its aim, as we have stated, is to avoid page faults being raised
  - A problem arises when we try to implement the working set model as we need to know which pages make up the working set
  - One solution is to use the aging algorithm described above. Any page that contains a 1 in n high order bits is deemed to be a member of the working set. The value of n has to be experimentally although it has been found that the value is not that sensitive
- Paging Daemons
  - If a page fault occurs it is better if there are plenty of free pages for the page to be copied to
  - If every page is full we have to find a page to evict and we may have to write the page to disc before evicting it
  - Many systems have a background process called a paging daemon
  - This process sleeps most of the time but runs at periodic intervals
  - Its task is to inspect the state of the page frames and, if too few pages are free, it selects pages to evict using the page replacement algorithm that is being used
  - A further performance improvement can be achieved by remembering which page frame a page has been evicted from
  - If the page frame has not been overwritten when the evicted page is needed again then the page frame is still valid and the data does not have to copied from disc again
  - In addition the paging daemon can ensure pages are clean

Page Table



- Page Frame Number : This is the number of the physical page that this page maps to. As this is the whole point of the page, this can be considered the most important part of the page frame entry
- Present/Absent Bit : This indicates if the mapping is valid. A value of 1 indicates the physical page, to which this virtual page relates is in memory. A value of zero indicates the mapping is not valid and a page fault will occur if the page is accesse
- Protection : The protection bit could simply be a single bit which is set to 0 if the page can be read and written and 1 if the page can only be read. If three bits are allowed then each bit can be used to represent read, write and execute
- Modified : This bit is updated if the data in the page is modified. This bit is used when the data in the page is evicted. If the modified bit is set, the data in the page frame needs to be written back to disc. If the modified bit is not set, then the data can simply be evicted, in the knowledge that the data on disc is already up to date
- Referenced : This bit is updated if the page is referenced. This bit can be used when deciding which page should be evicted (we will be looking at its use later)
- Caching Disabled : This bit allows caching to be disabled for the page. This is useful if a memory address maps onto a device register rather than to a memory address. In this case, the register could be changed by the device and it is important that the register is accessed, rather than using the cached value which may not be up to date.

Multilevel Page Table



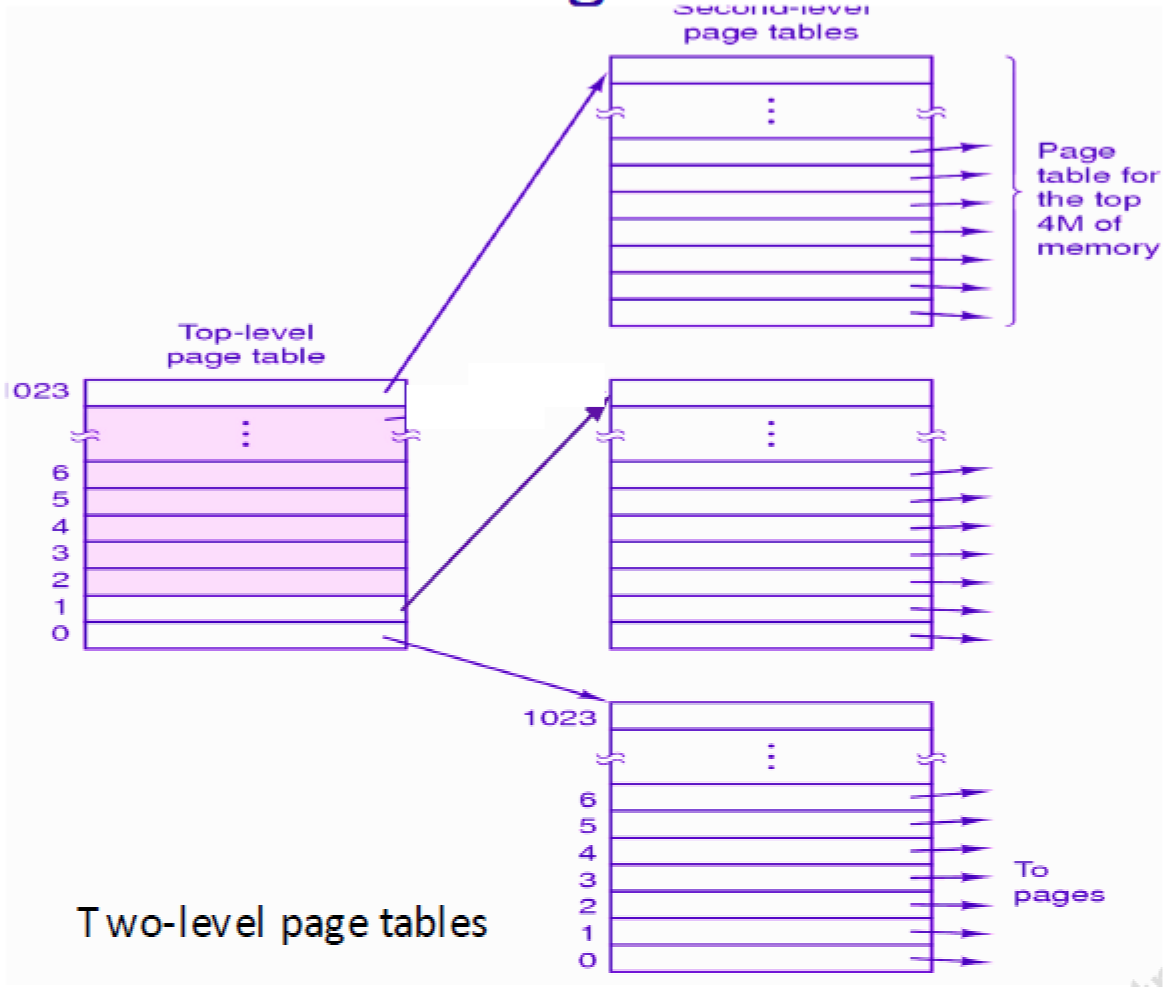
Example: Two-Level Page Tables

A 32-bit virtual address space with a page size of 4 KB, the virtual address space is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit offset field.

Bits
10
10
12
PT1
PT2
Offset

The top level have 1024 entries, corresponding to PT1. At mapping, it first extracts the PT1 and uses this value as an index into the top level page table. Each of these entries have again 1024 entries, the resulting address of top-level yields the address or page frame number of second-level page table.

Multilevel Page Tables



Hashed Page Tables

A common approach for handling address space larger than 32-bit.

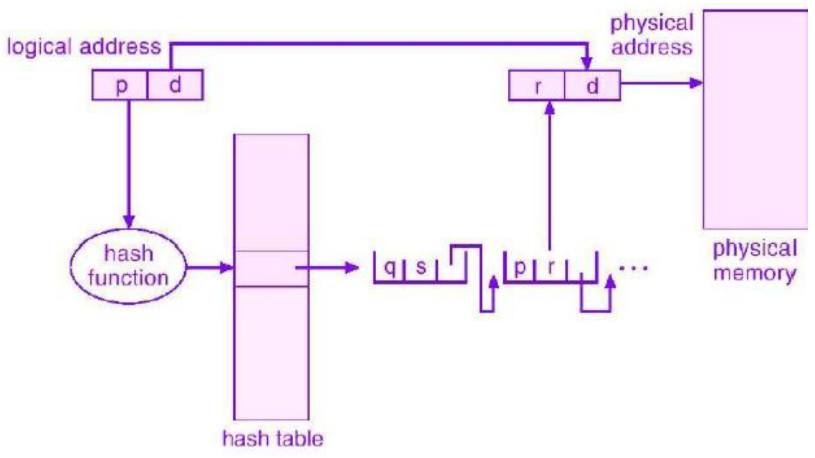
The hash value is the virtual-page number.

Each entry in the hash table contains a linked list of elements that hash to the same location.

Each element consists of three fields: virtual-page-number, value of mapped page frame, and a pointer to the next element.

The virtual address is hashed into the hash table, if there is match the corresponding page frame is used, if not, subsequent entries in the linked list are searched.

Hashed Page Tables

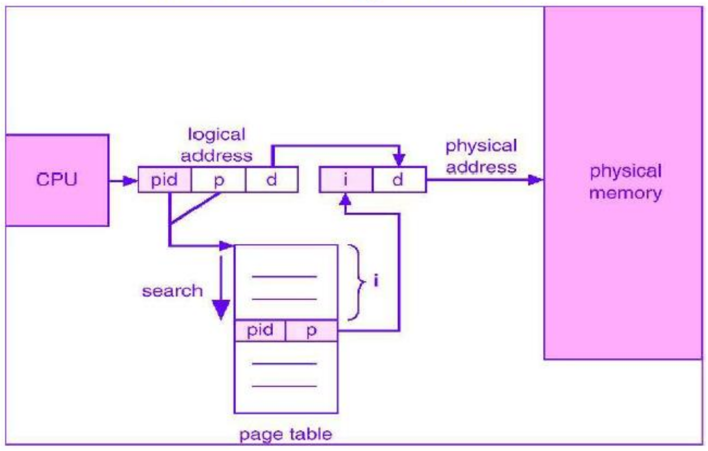


Hashed Page Table

Inverted Page Tables

A common approach for handling address space larger than 32-bit.
One entry per page frame, rather than one entry per page in earlier tables. (Ex: 256MB RAM with 4KB page requires only 65,536 entries in table)
Entry consists of the virtual address of the page stored in that physical memory location, with information about the process that owns that page.
Virtual address consists three fields: [process-id, page-number, offset]. The inverted page table entry is determined by [process-id, page-number]. The page table is search for the match, say at entry i the match is found, then the physical address [i, offset] is generated.

Inverted Page Tables



Advantage: Decreases the memory size to store the page table.

Problem: It must search entire table in every memory reference, not just on page faults.

Searching a 64K table in every reference is not a way to make system fast! Solutions:

Use of Hashed tables.

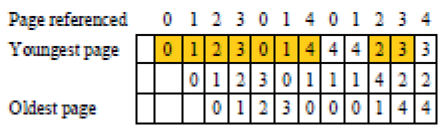
Use of TLBs.

Modeling page replacement algorithms

- Goal: provide quantitative analysis (or simulation) showing which algorithms do better
- Workload (page reference string) is important: different strings may favor different algorithms
- Show tradeoffs between algorithms
- Compare algorithms to one another
- Model parameters within an algorithm
- Number of available physical pages
- Number of bits for aging

How is modeling done?

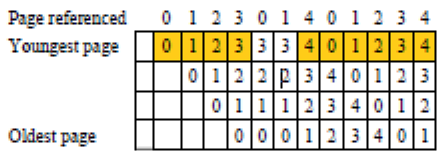
- Generate a list of references
- Artificial (made up)
- Trace a real workload (set of processes)
- Use an array (or other structure) to track the pages in physical memory at any given time
- May keep other information per page to help simulate the algorithm (modification time, time when paged in, etc.)
- Run through references, applying the replacement algorithm
- Example: FIFO replacement on reference string 0 1 2 3 0 1 4 0 1 2 3 4
- Page replacements highlighted in yellow



Page referenced	0	1	2	3	0	1	4	0	1	2	3	4
Youngest page	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest page			0	1	2	3	0	0	0	1	4	4

Belady's anomaly

- Reduce the number of page faults by supplying more memory
- Use previous reference string and FIFO algorithm
- Add another page to physical memory (total 4 pages)
- More page faults (10 vs. 9), not fewer!.
- This is called *Belady's anomaly*
- Adding more pages shouldn't result in worse performance!
- Motivated the study of paging algorithms



Page referenced		0	1	2	3	0	1	4	0	1	2	3	4
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4
			0	1	2	2	p	3	4	0	1	2	3
				0	1	1	1	2	3	4	0	1	2
Oldest page					0	0	0	1	2	3	4	0	1

Modeling more replacement algorithms

 Paging system characterized by:
 Reference string of executing process
 Page replacement algorithm
 Number of page frames available in physical memory (*m*)
 Model this by keeping track of all *n* pages referenced in array *M*
 Top part of *M* has *m* pages in memory
 Bottom part of *M* has *n-m* pages stored on disk
 Page replacement occurs when page moves from top to bottom
 Top and bottom parts may be rearranged without causing movement between memory and disk

Stack algorithms

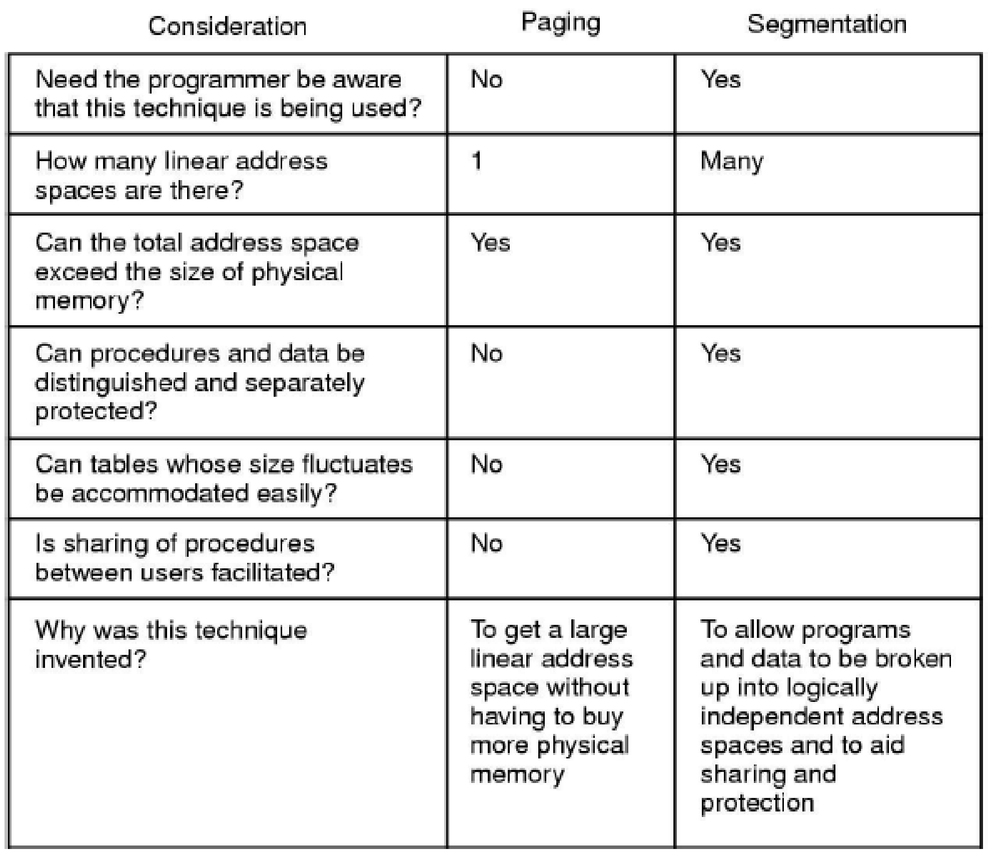
 LRU is an example of a stack algorithm
 For stack algorithms
 Any page in memory with *m* physical pages is also in memory with *m*+1 physical pages
 Increasing memory size is guaranteed to reduce (or at least not increase) the number of page faults
 Stack algorithms do not suffer from Belady's anomaly
 Distance of a reference == position of the page in the stack before the reference was made
 Distance is ∞ if no reference had been made before
 Distance depends on reference string and paging algorithm: might be different for LRU and optimal (both stack algorithms

How is a page fault handled?

 Hardware causes a page fault
 General registers saved (as on every exception)
 OS determines which virtual page needed
 Actual fault address in a special register
 Address of faulting instruction in register
 Page fault was in fetching instruction, or
 Page fault was in fetching operands for instruction
 OS must figure out which…
 OS checks validity of address
 Process killed if address was illegal
 OS finds a place to put new page frame
 If frame selected for replacement is dirty, write it out to disk
 OS requests the new page from disk
 Page tables updated
 Faulting instruction backed up so it can be restarted
 Faulting process scheduled

- Registers restored
- Program continues

Segmentation Vs Paging



Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection