CTBTCSE – SIV P4 – Operating System

“Unit-3 Memory Management”

Dr. Parag Shukla
Assistant Professor,
School of Cyber Security and Digital Forensics
National Forensic Sciences University

MEMORY MANAGEMENT

Presentation Outline

Main Memory

Contiguous Memory Allocation, Paging Structure of the Page Table, Swapping

Main Memory

Intel-32 and 64 bit, Architectures, Segmentation, Page Fault Handling

Virtual Memory

Demand Paging, Copy-On-Write, Page Replacement,

Virtual Memory

Allocation of Frames, Thrashing,

Virtual Memory

Memory Compression, Allocating Kernel Memory

Main Memory

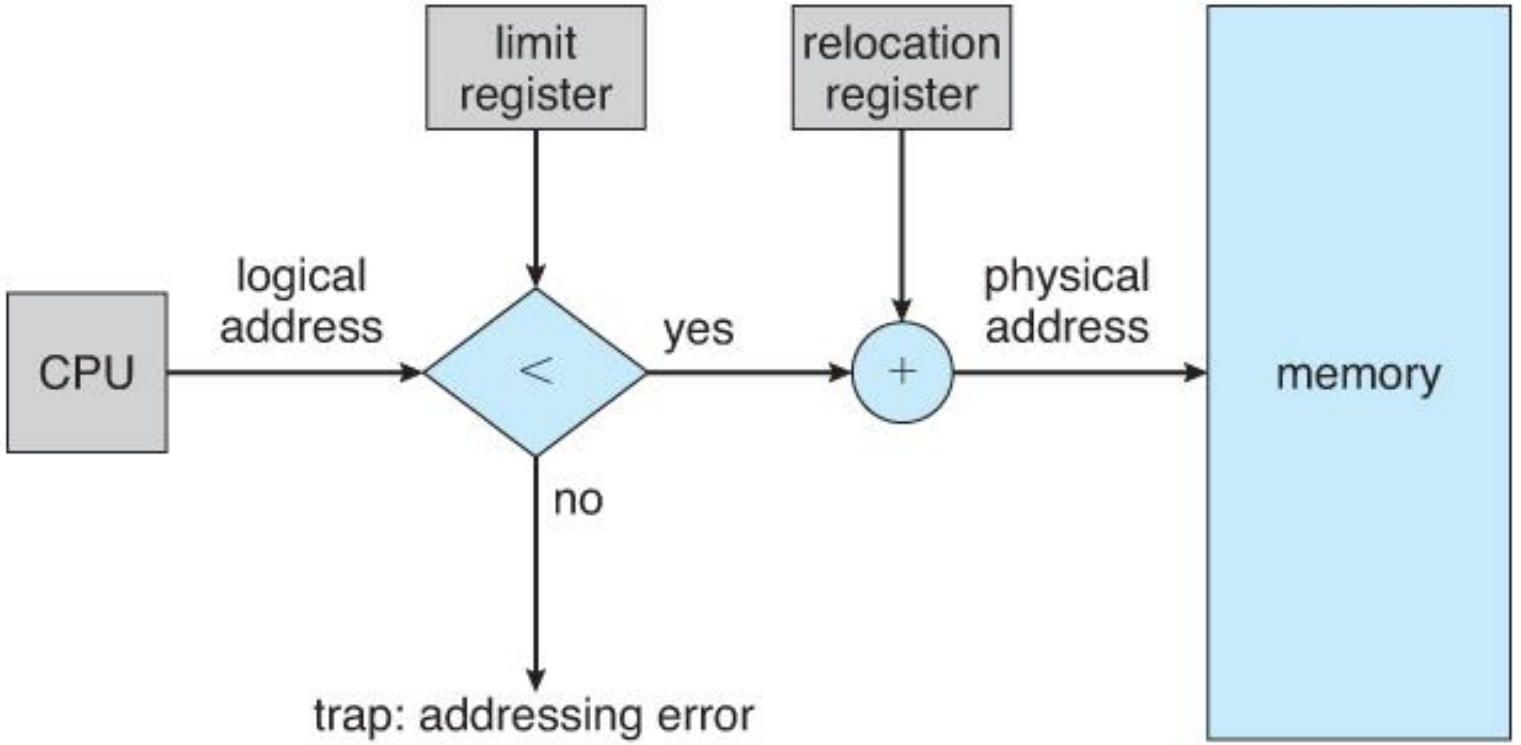
- Obviously memory accesses and memory management are a very important part of modern computer operation.
- Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.
- The advent of multi-tasking OSes compounds the complexity of memory management, because as processes are swapped in and out of the CPU, so must their code and data be swapped in and out of memory, all at high speeds and without interfering with any other processes.
- Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.
- It should be noted that from the memory chips point of view, all memory accesses are equivalent.
- The memory hardware doesn't know what a particular part of memory is being used for, nor does it care. This is almost true of the OS as well, although not entirely.
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it.
- (Device drivers communicate with their hardware via interrupts and "memory" accesses, sending short instructions for example to transfer data from the hard drive to a specified location in main memory.
- The disk controller monitors the bus for such instructions, transfers the data, and then notifies the CPU that the data is there with another interrupt, but the CPU never gets direct access to the disk.)

Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space.
- The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed.
- (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.)

Contiguous Memory Allocation

- **Memory Protection (was Memory Mapping and Protection)**
- The system shown in Figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change



Hardware support for relocation and limit registers

Contiguous Memory Allocation

Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:
  - **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
  - **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
  - **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.
- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

Contiguous Memory Allocation

Fragmentation

- All the memory allocation strategies suffer from ***external fragmentation***, though first and best fits experience the problems more so than worst fit.
- External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.
- The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.
- Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation.
- ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average 1/2 block will be wasted per memory request, because on the average the last allocated block will be only half full.
  - Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.
  - Some systems use variable size blocks to minimize losses due to internal fragmentation.
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

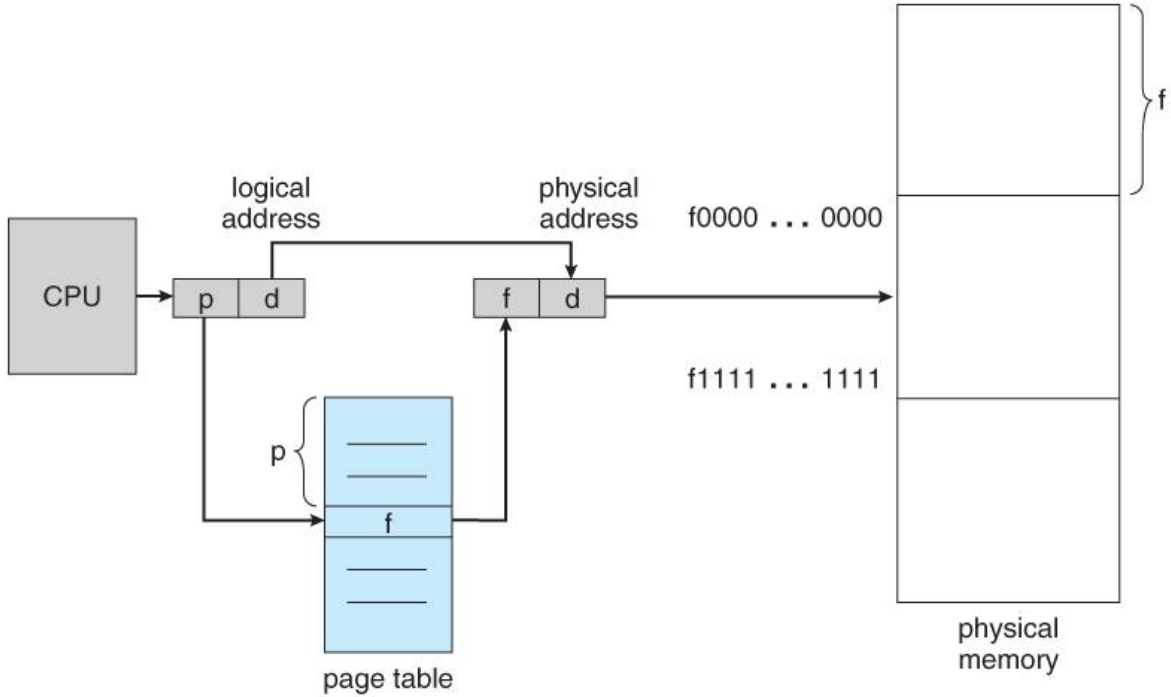
Paging

Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

Basic Method

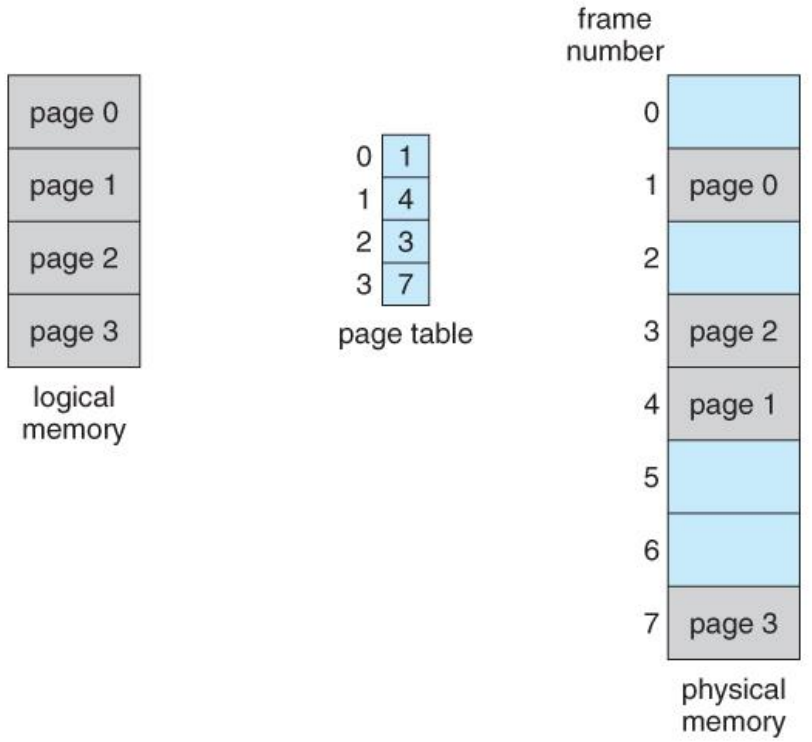
- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:



Paging Hardware

Paging

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page.
- (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.
- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical.
- The former determines the address range of the logical address space, and the latter relates to the physical address space.

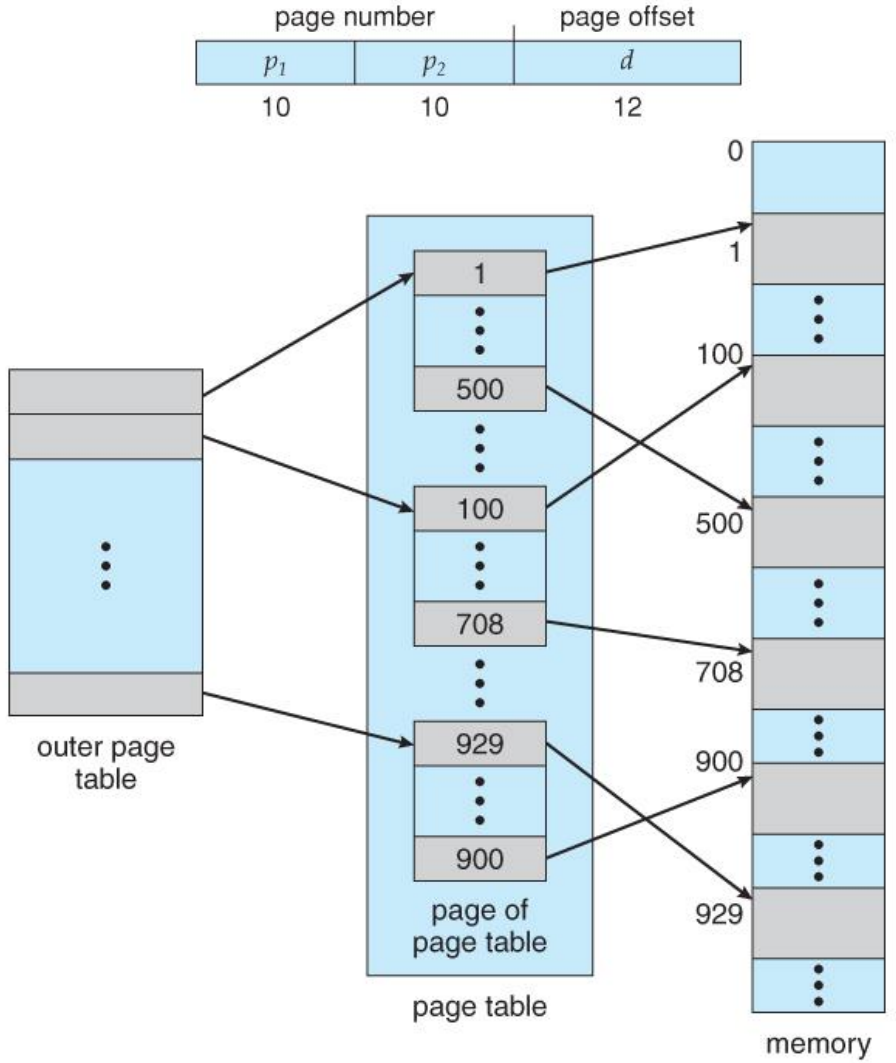


Paging Model of logical and physical memory

Structure of the Page Table

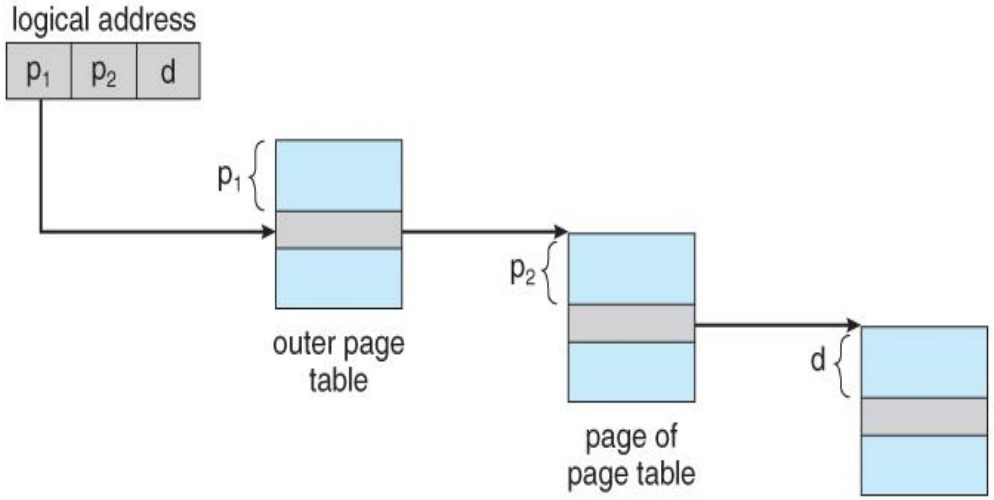
Hierarchical Paging

- Most modern computer systems support logical address spaces of 2^32 to 2^64.
- With a 2^32 address space and 4K (2^12) page sizes, this leave 2^20 entries in the page table.
- At 4 bytes per entry, this amounts to a 4 MB page table, which is too large to reasonably keep in contiguous memory. (And to swap in and out of memory with each process switch.) Note that with 4K pages, this would take 1024 pages just to hold the page table!
- One option is to use a two-tier paging system, i.e. to page the page table.
- For example, the 20 bits described above could be broken down into two 10-bit page numbers.
- The first identifies an entry in the outer page table, which identifies where in memory to find one page of an inner page table.
- The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)



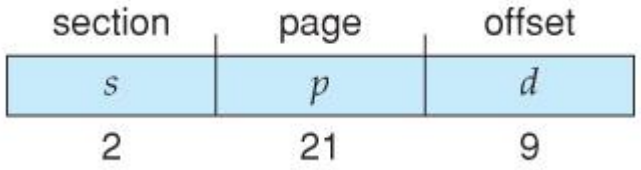
2 Level page table scheme

Structure of the Page Table

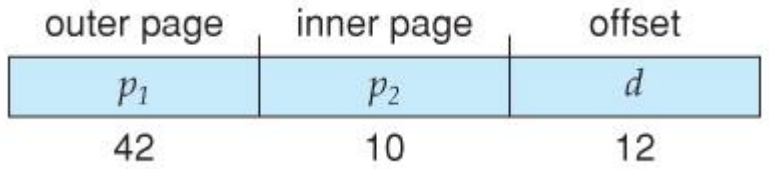


Address Translation for two-level 32-bit paging architecture

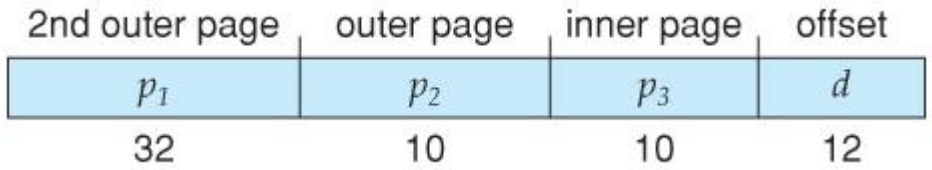
- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form of:



- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging.
- One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively slow memory access. So some other approach must be used.



64-bits Two-tiered leaves 42 bits in outer table

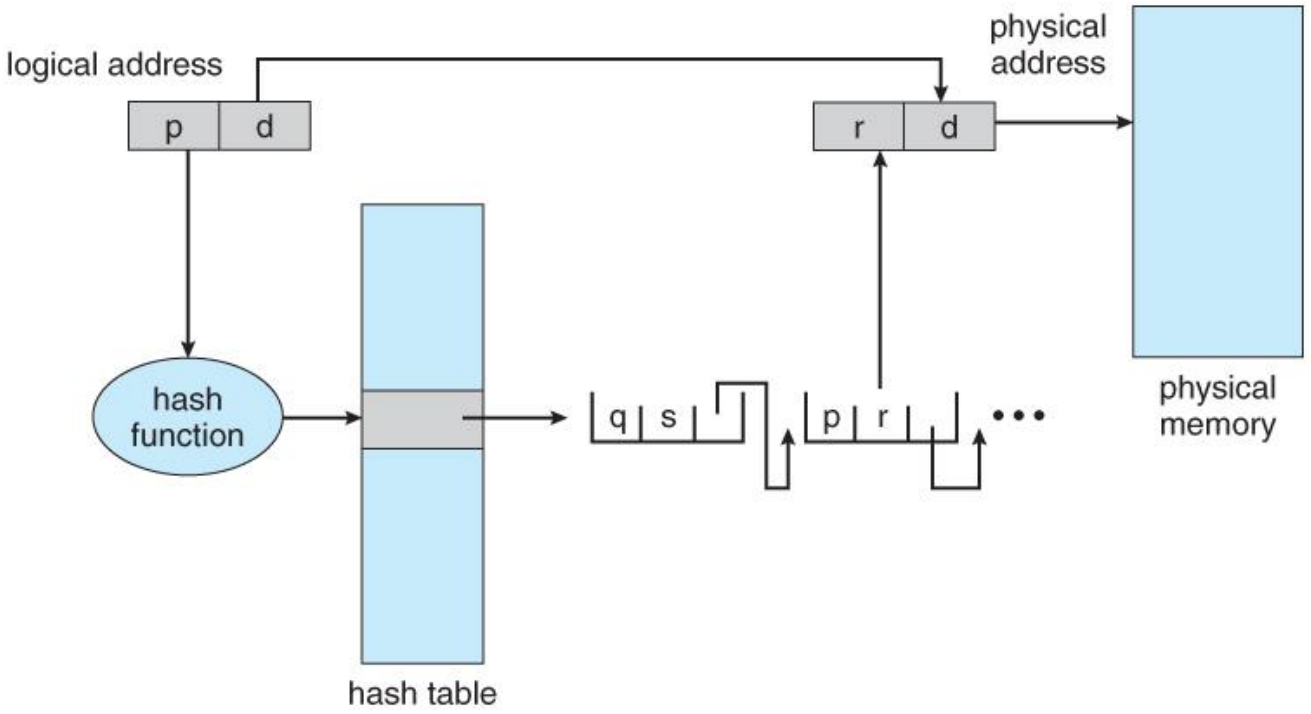


Going to a fourth level still leaves 32 bits in the outer table

Structure of the Page Table

Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***.
- Figure below illustrates a ***hashed page table*** using chain-and-bucket hashing:

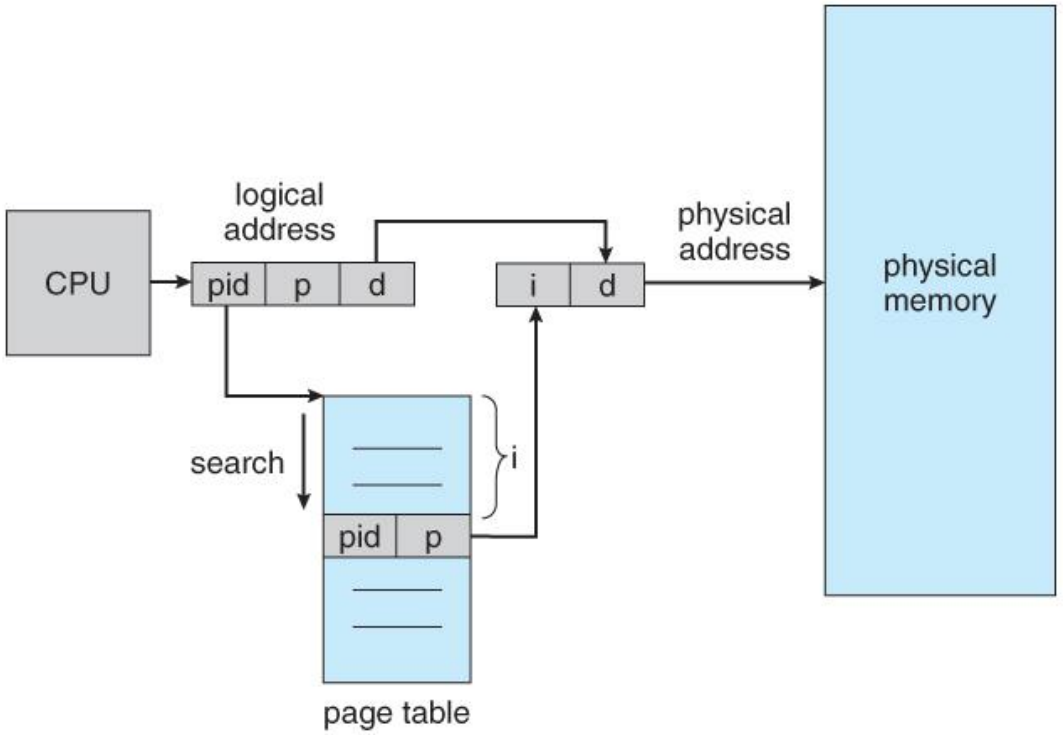


Hashed page table

Structure of the Page Table

Inverted Page Tables

- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (I.e. there is one entry per ***frame*** instead of one entry per ***page***.)
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page (or to discover that it is not there.) Hashing the table can help speedup the search process.
- Inverted page tables prohibit the normal method of implementing shared memory, which is to map multiple logical pages to a common physical frame. (Because each frame is now mapped to one and only one process.)



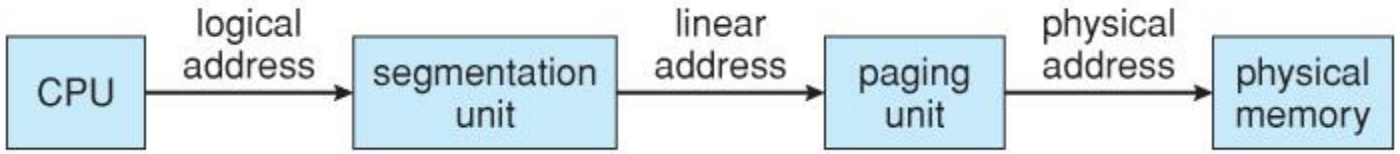
Inverted page table

Intel 32 and 64 bit Architectures

Intel 32 and 64-bit Architectures

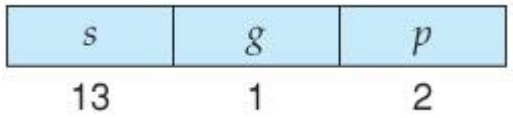
IA-32 Segmentation

- The Pentium CPU provides both pure segmentation and segmentation with paging. In the latter case, the CPU generates a logical address (segment-offset pair), which the segmentation unit converts into a logical linear address, which in turn is mapped to a physical frame by the paging unit, as shown in Figure

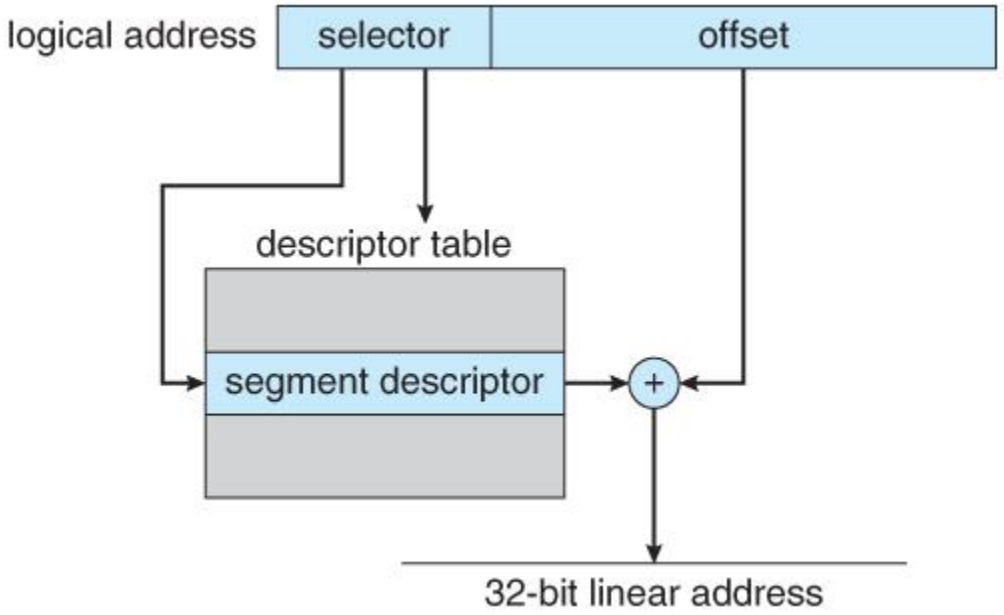


- The Pentium architecture allows segments to be as large as 4 GB, (24 bits of offset).
- Processes can have as many as 16K segments, divided into two 8K groups:
  - 8K private to that particular process, stored in the ***Local Descriptor Table, LDT.***
  - 8K shared among all processes, stored in the ***Global Descriptor Table, GDT.***
- Logical addresses are (selector, offset) pairs, where the selector is made up of 16 bits:
  - A 13 bit segment number (up to 8K)
  - A 1 bit flag for LDT vs. GDT.
  - 2 bits for protection codes.

Intel 32 and 64 bit Architectures



- The descriptor tables contain 8-byte descriptions of each segment, including base and limit registers.
- Logical linear addresses are generated by looking the selector up in the descriptor table and adding the appropriate base address to the offset, as shown in following Figure

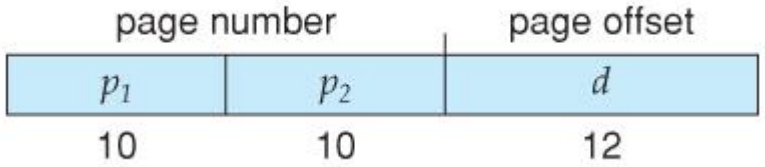


IA-32 segmentation

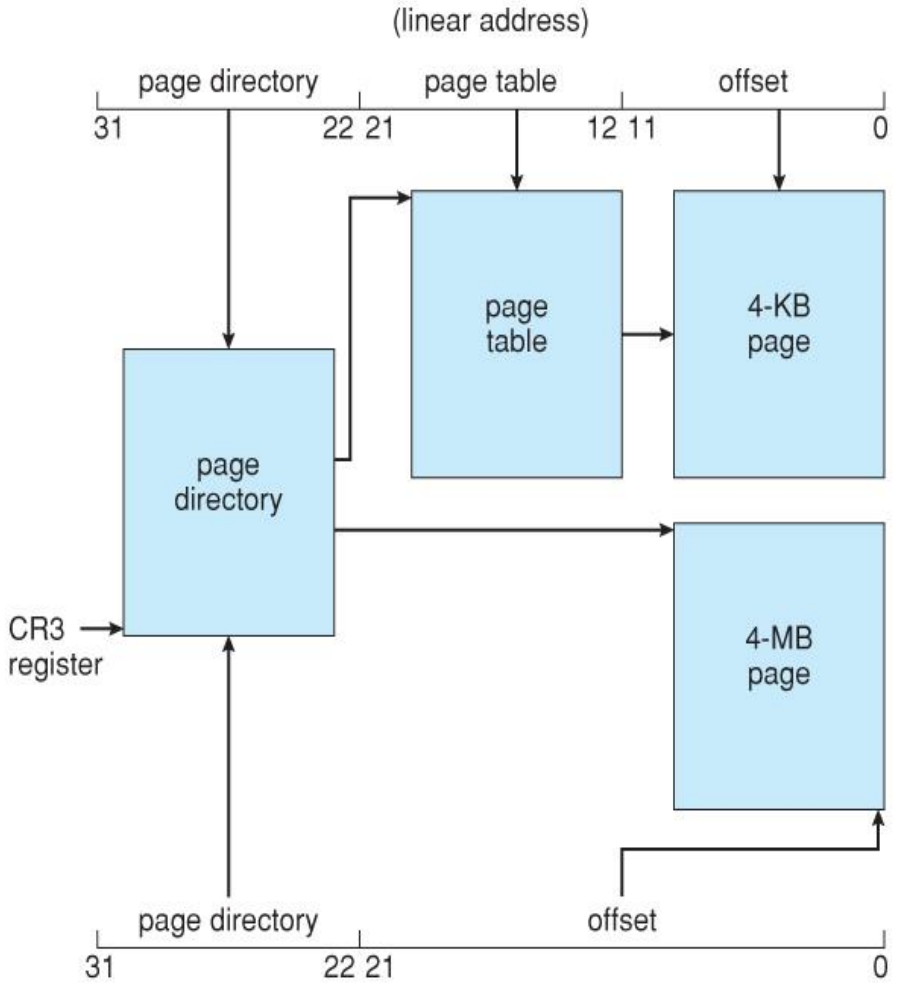
Intel 32 and 64 bit Architectures

IA-32 Paging

- Pentium paging normally uses a two-tier paging scheme, with the first 10 bits being a page number for an outer page table (a.k.a. page directory), and the next 10 bits being a page number within one of the 1024 inner page tables, leaving the remaining 12 bits as an offset into a 4K page.



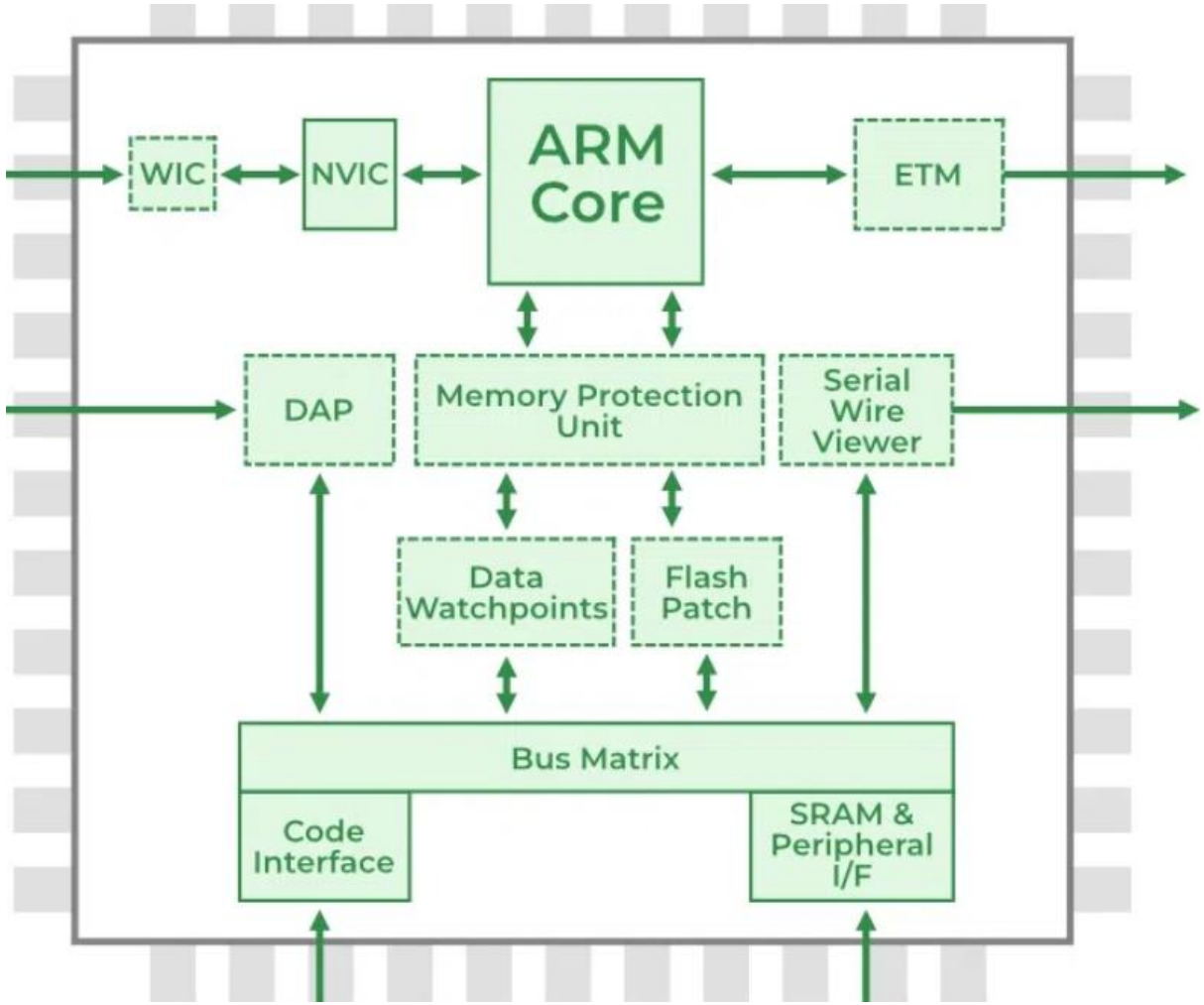
- A special bit in the page directory can indicate that this page is a 4MB page, in which case the remaining 22 bits are all used as offset and the inner tier of page tables is not used.
- The CR3 register points to the page directory for the current process, as shown in this Figure.
- If the inner page table is currently swapped out to disk, then the page directory will have an "invalid bit" set, and the remaining 31 bits provide information on where to find the swapped out page table on the disk.



Paging in the IA-32 architecture.

ARM Architecture

- Advanced RISC Machine or Acorn RISC Machine is the architecture with different computing architectures set to be used in different environments. 32-bit and 64-bit can be used here in different computer processors.
- It was developed by Arm Holdings and the architecture is updated in between.
- This architecture is specified to be used with CPU, different chips in the system, and in different registers.
- Reduced Instruction Set Computing helps in creating instructions for the system to be used for several purposes.
- Smartphones, microcomputers, and embedded devices also use ARM architecture for the instruction set in the registers



ARM

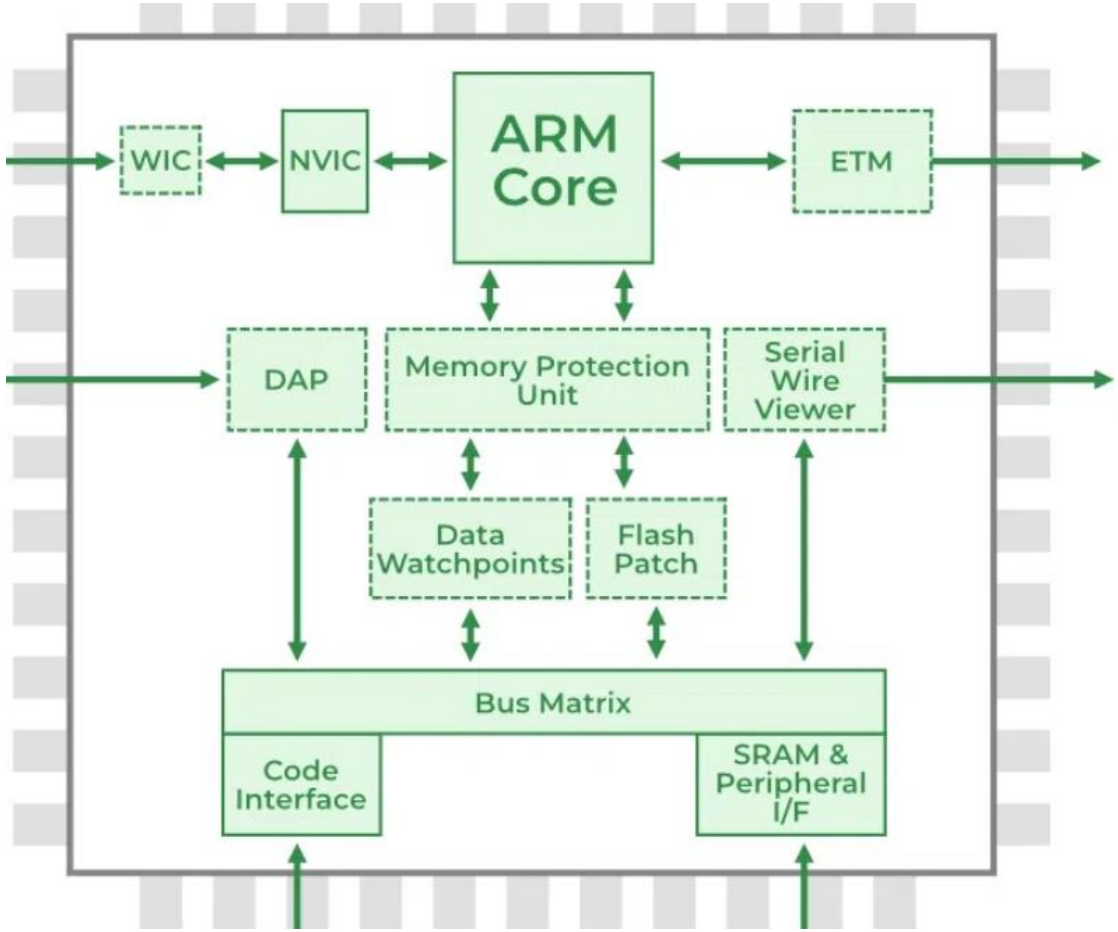
ARM Architecture

Features of ARM Processor

- Multiprocessing System
- Tightly Coupled Memory
- Memory Management
- Thumb-2 Technology
- One-Cycle Execution Time
- Pipelining
- A large number of Registers

Advantages of ARM Processor

- ARM processors deal with a single processor at a time, which makes it faster and it also consumes lesser power.
- ARM processors work in the case of a multiprogramming system, where more than one processor is used to process information.
- ARM processors are cheaper than other processors, which makes them usable in mobile phones.
- ARM processors are scalable, and this feature helps it in using a variety of devices



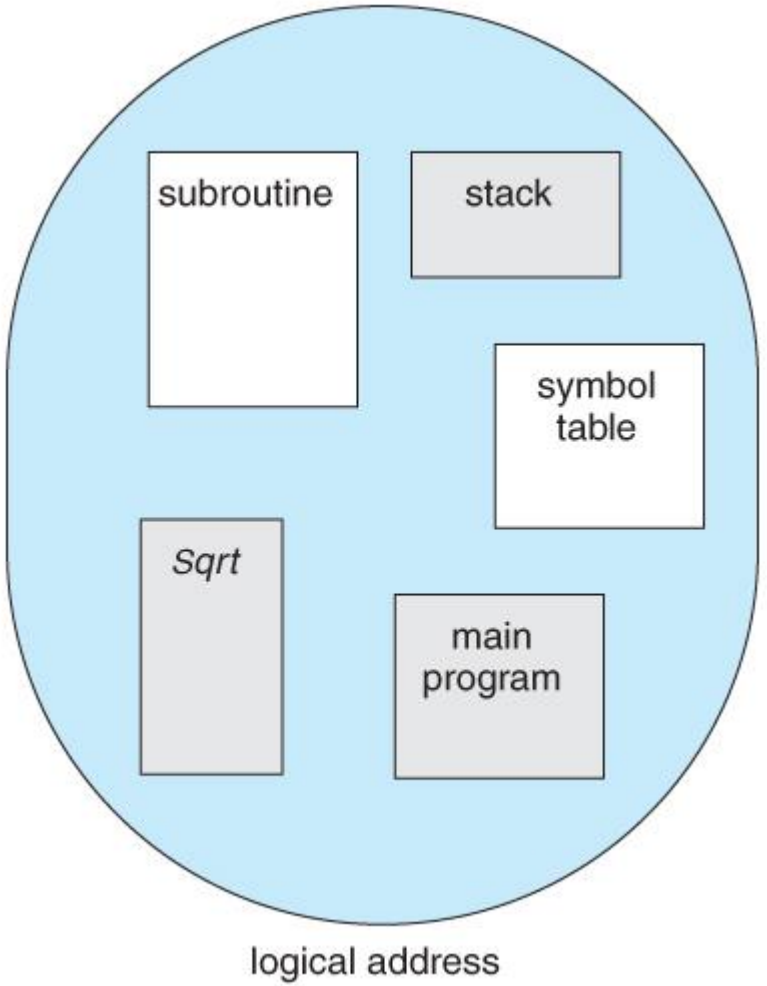
ARM

ARM Vs x86

ARM	x86
ARM uses Reduced Instruction Set Computing Architecture (RISC).	x86 uses Complex Instruction Set Architecture (CISC).
ARM works by executing single instruction per cycle.	x86 works by executing complex instructions at once and it requires more than one cycle.
Performance can be optimized by a Software-based approach.	Performance can be optimized by Hardware based approach.
ARM processors require fewer registers, but they require more memory.	x86 processors require less memory, but more registers.
Execution is faster in ARM Processes.	Execution is slower in an x86 Processor.
ARM Processor work by generating multiple instructions from a complex instruction and they are executed separately.	x86 Processors work by executing complex statements at a single time.
ARM processors use the memory which is already available to them.	x86 processors require some extra memory for calculations.
ARM processors are deployed in mobiles which deal with the consumption of power, speed, and size.	x86 processors are deployed in Servers, Laptops where performance and stability matter.

Segmentation

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure

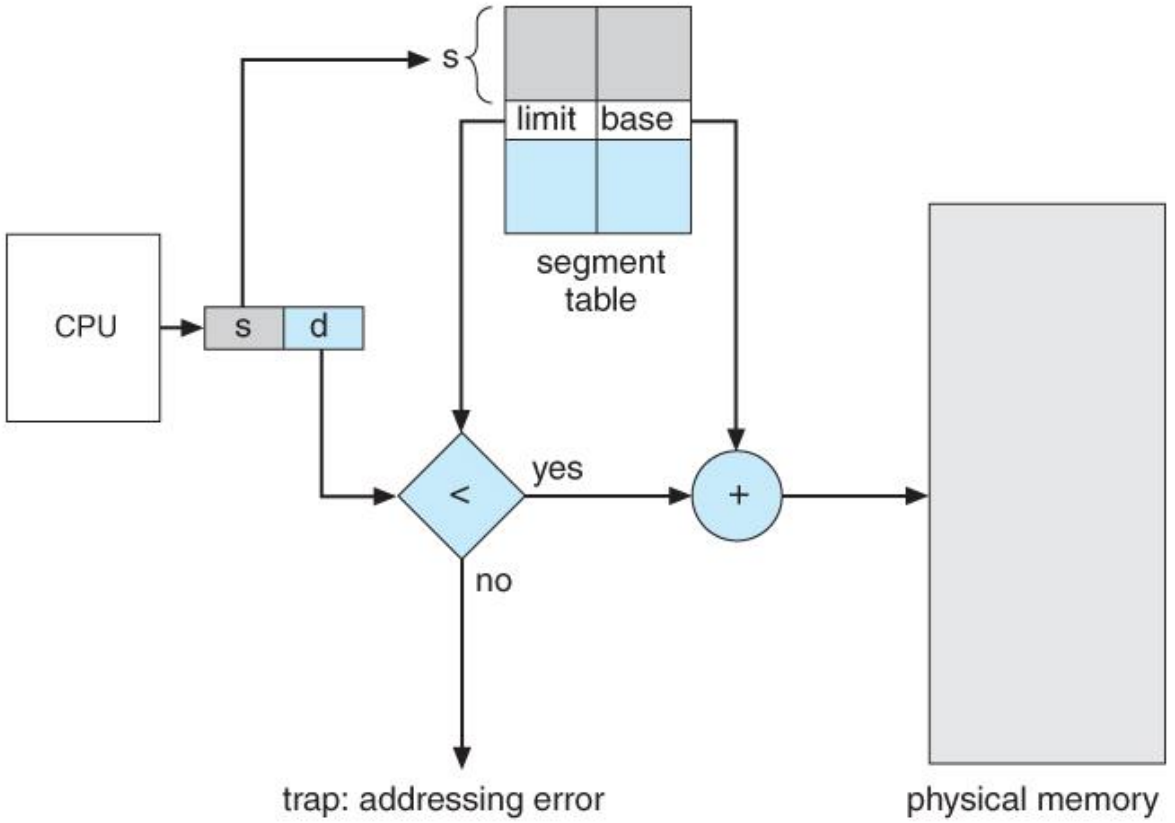


Programmer's view of a program.

Segmentation

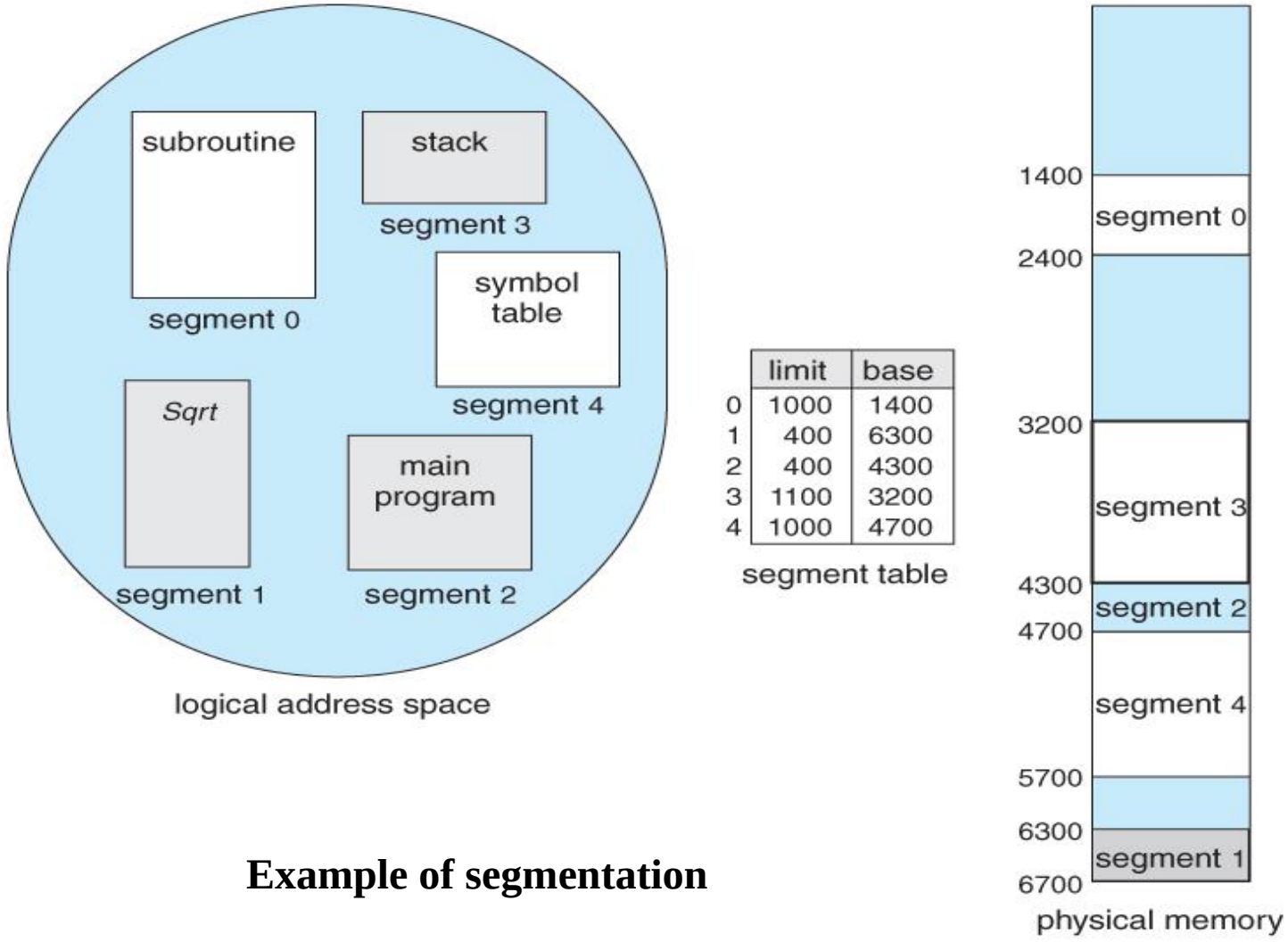
Segmentation Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously.
- Note that at this point in the discussion of segmentation, each segment is kept in contiguous memory and may be of different sizes, but that segmentation can also be combined with paging



Segmentation hardware

Segmentation



Example of segmentation

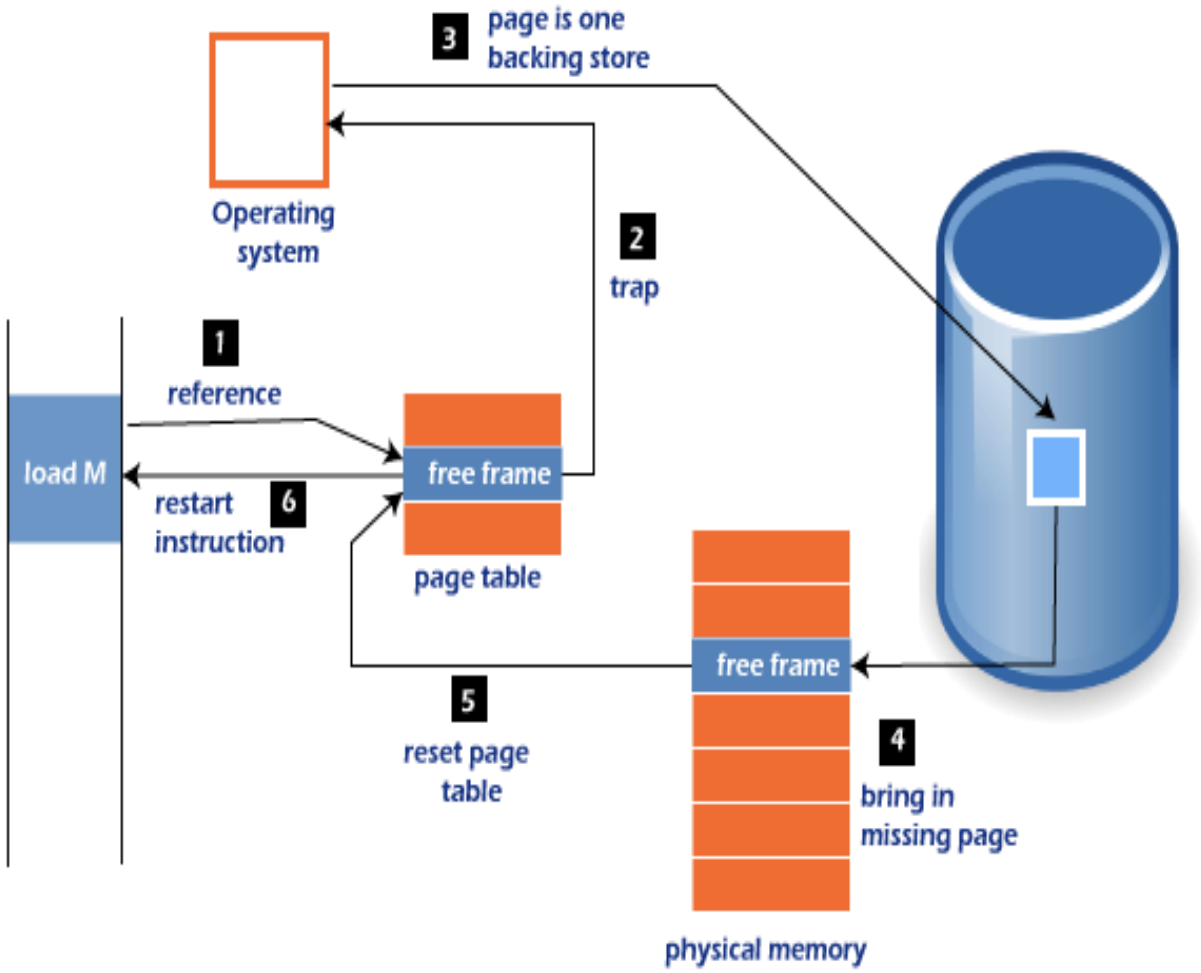
Paging Vs Segmentation

S.NO	Paging	Segmentation
1.	In paging, the program is divided into fixed or mounted size pages.	In segmentation, the program is divided into variable size sections.
2.	For the paging operating system is accountable.	For segmentation compiler is accountable.
3.	Page size is determined by hardware.	Here, the section size is given by the user.
4.	It is faster in comparison to segmentation.	Segmentation is slow.
5.	Paging could result in internal fragmentation.	Segmentation could result in external fragmentation.
6.	In paging, the logical address is split into a page number and page offset.	Here, the logical address is split into section number and section offset.
7.	Paging comprises a page table that encloses the base address of every page.	While segmentation also comprises the segment table which encloses segment number and segment offset.
8.	The page table is employed to keep up the page data.	Section Table maintains the section data.

S.NO	Paging	Segmentation
9.	In paging, the operating system must maintain a free frame list.	In segmentation, the operating system maintains a list of holes in the main memory.
10.	Paging is invisible to the user.	Segmentation is visible to the user.
11.	In paging, the processor needs the page number, and offset to calculate the absolute address.	In segmentation, the processor uses segment number, and offset to calculate the full address.
12.	It is hard to allow sharing of procedures between processes.	Facilitates sharing of procedures between the processes.
13	In paging, a programmer cannot efficiently handle data structure.	It can efficiently handle data structures.
14.	This protection is hard to apply.	Easy to apply for protection in segmentation.
15.	The size of the page needs always be equal to the size of frames.	There is no constraint on the size of segments.
16.	A page is referred to as a physical unit of information.	A segment is referred to as a logical unit of information.
17.	Paging results in a less efficient system.	Segmentation results in a more efficient system.

Page Fault Handling

- Page faults dominate more like an **error**. A page fault will happen if a program tries to access a piece of memory that does not exist in physical memory (main memory).
- The fault specifies the operating system to trace all data into virtual memory management and then relocate it from secondary memory to its primary memory, such as a hard disk.
- A page fault trap occurs if the requested page is not loaded into memory.
- The page fault primarily causes an exception, which is used to notify the operating system to retrieve the **"pages"** from virtual memory to continue operation.
- Once all of the data has been placed into physical memory, the program resumes normal operation.
- The Page fault process occurs in the background, and thus the user is unaware of it.



Page Fault Handling

- A Page Fault happens when you access a page that has been marked as invalid.
- The paging hardware would notice that the invalid bit is set while translating the address across the page table, which will cause an operating system trap.
- The trap is caused primarily by the OS's failure to load the needed page into memory.

Now, let's understand the procedure of page fault handling in the OS:

1. Firstly, an internal table for this process to assess whether the reference was valid or invalid memory access.
2. If the reference becomes invalid, the system process would be terminated. Otherwise, the page will be paged in.
3. After that, the free-frame list finds the free frame in the system.
4. Now, the disk operation would be scheduled to get the required page from the disk.
5. When the I/O operation is completed, the process's page table will be updated with a new frame number, and the invalid bit will be changed. Now, it is a valid page reference.
6. If any page fault is found, restart these steps from starting.

Page Fault Handling

Page Fault Terminology

- There are various page fault terminologies in the operating system.
- Some terminologies of page fault are as follows:

1. Page Hit

- When the CPU attempts to obtain a needed page from main memory and the page exists in main memory (RAM), it is referred to as a "PAGE HIT".

2. Page Miss

- If the needed page has not existed in the main memory (RAM), it is known as "PAGE MISS".

3. Page Fault Time

- The time it takes to get a page from secondary memory and recover it from the main memory after loading the required page is known as "PAGE FAULT TIME".

4. Page Fault Delay

- The rate at which threads locate page faults in memory is referred to as the "PAGE FAULT RATE". The page fault rate is measured per second.

Page Fault Handling

5. Hard Page Fault
If a required page exists in the hard disk's page file, it is referred to as a **"HARD PAGE FAULT".**

6. Soft Page Fault
If a required page is not located on the hard disk but is found somewhere else in memory, it is referred to as a **"SOFT PAGE FAULT".**

7. Minor Page Fault
If a process needs data and that data exists in memory but is being allotted to another process at the same moment, it is referred to as a **"MINOR PAGE FAULT".**

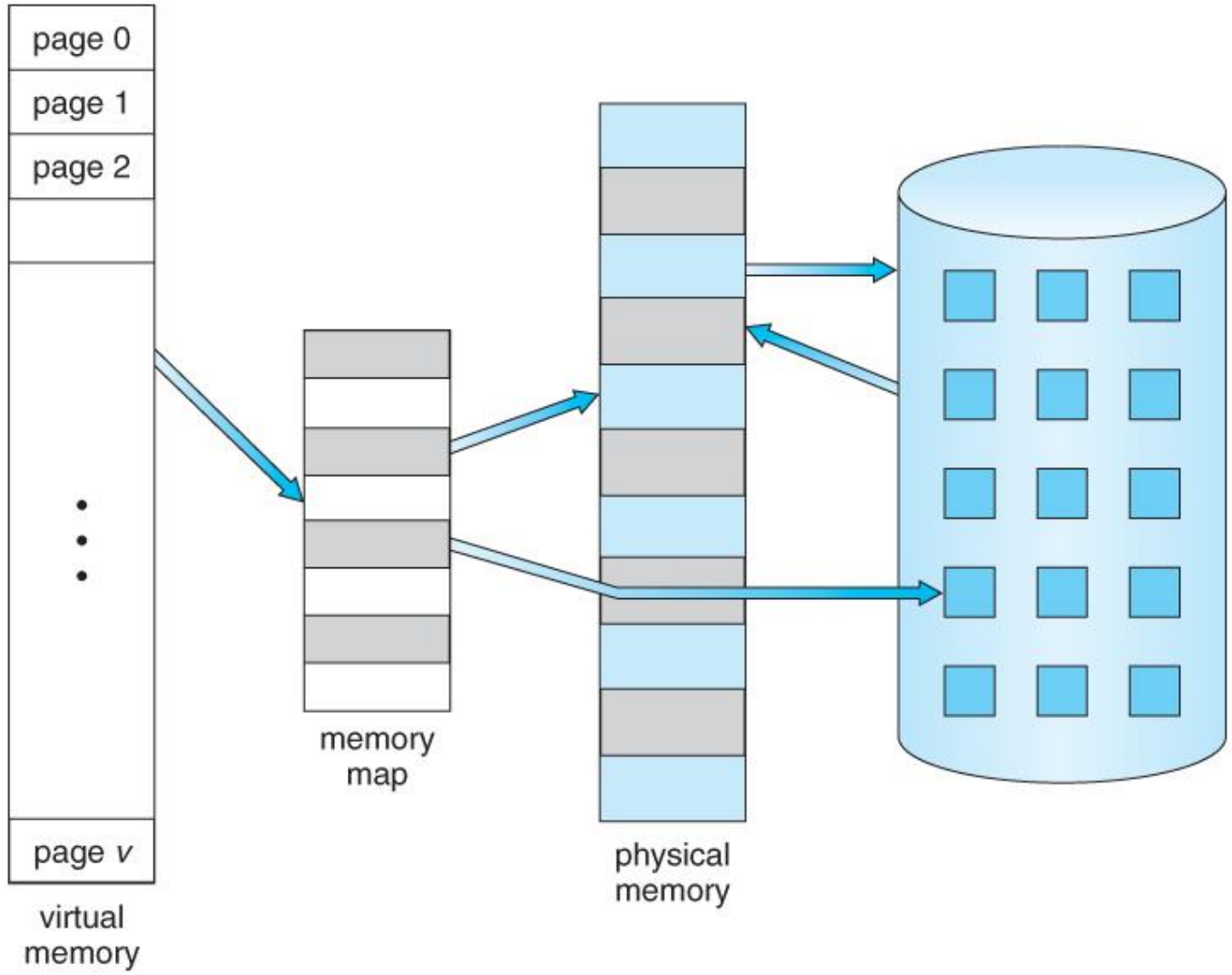
Virtual Memory - Overview

- Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the main memory.
- The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses.
- The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory is available not by the actual number of the main storage locations.
- It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
2. A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

Virtual Memory - Overview

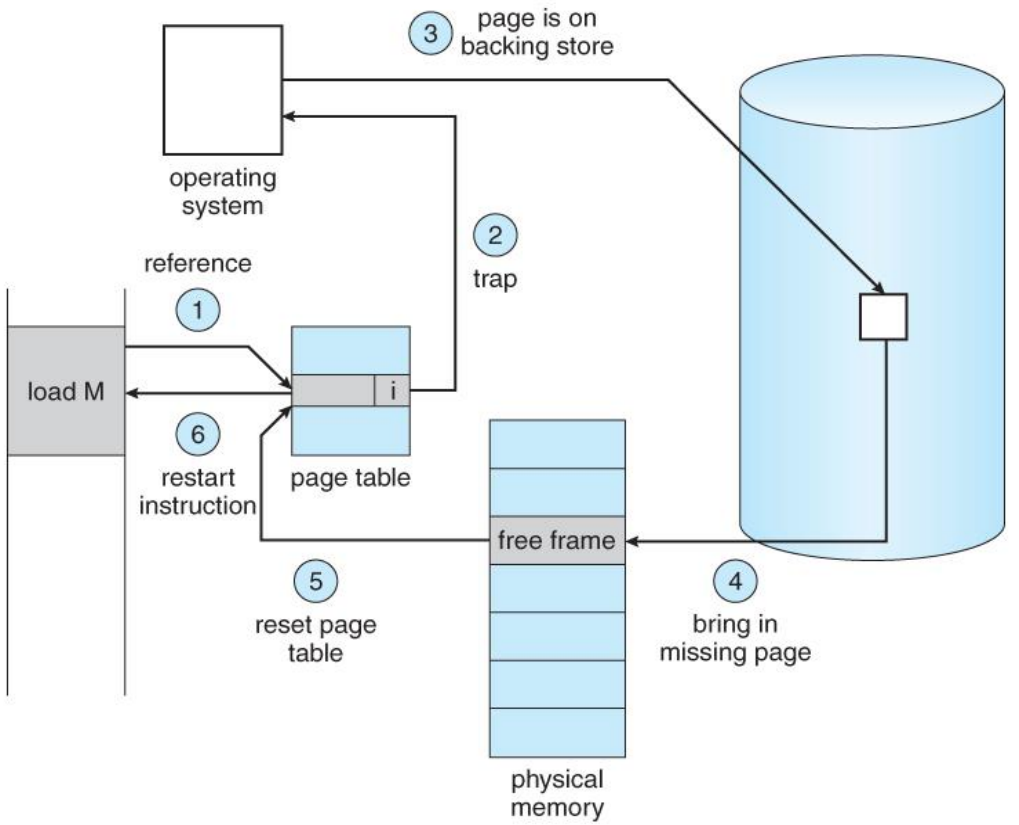
- The ability to load only the portions of processes that were actually needed (and only *when* they were needed) has several benefits:
  - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
  - Because each process is only using a fraction of its total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
  - Less I/O is needed for swapping processes in and out of RAM, speeding things up.



Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once.
- Rather they are swapped in only when the process needs them. (on demand.)
- This is termed a ***lazy swapper***, although a ***pager*** is a more accurate term.
- If a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:

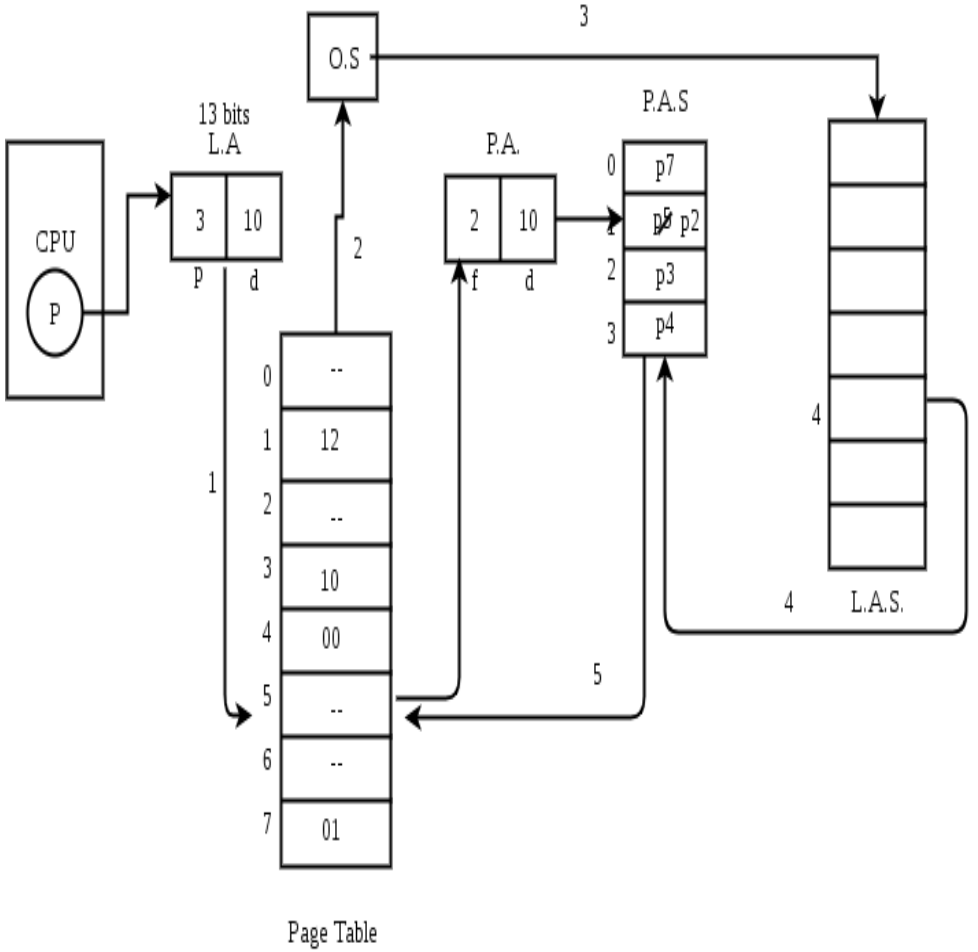
1. The memory address requested is first checked, to make sure it was a valid memory request.
2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)



Demand Paging

- The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging. The process includes the following steps are as follows:

1. If the CPU tries to refer to a page that is currently not available in the main memory, it generates an interrupt indicating a memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision-making of replacing the page in physical address space.
5. The page table will be updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into the ready state.



Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Demand Paging

Advantages

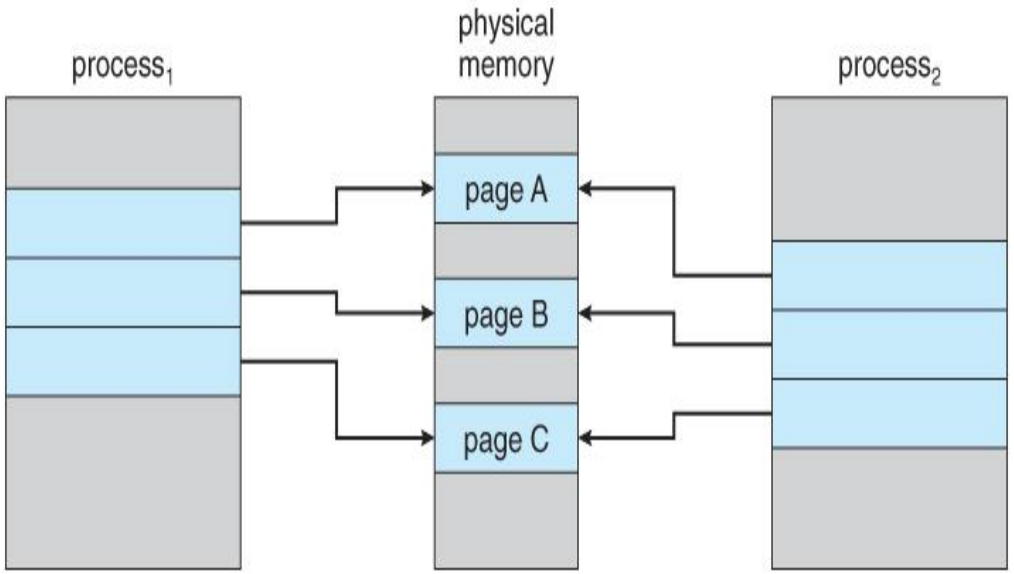
- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- A process may be larger than all of the main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

Disadvantages

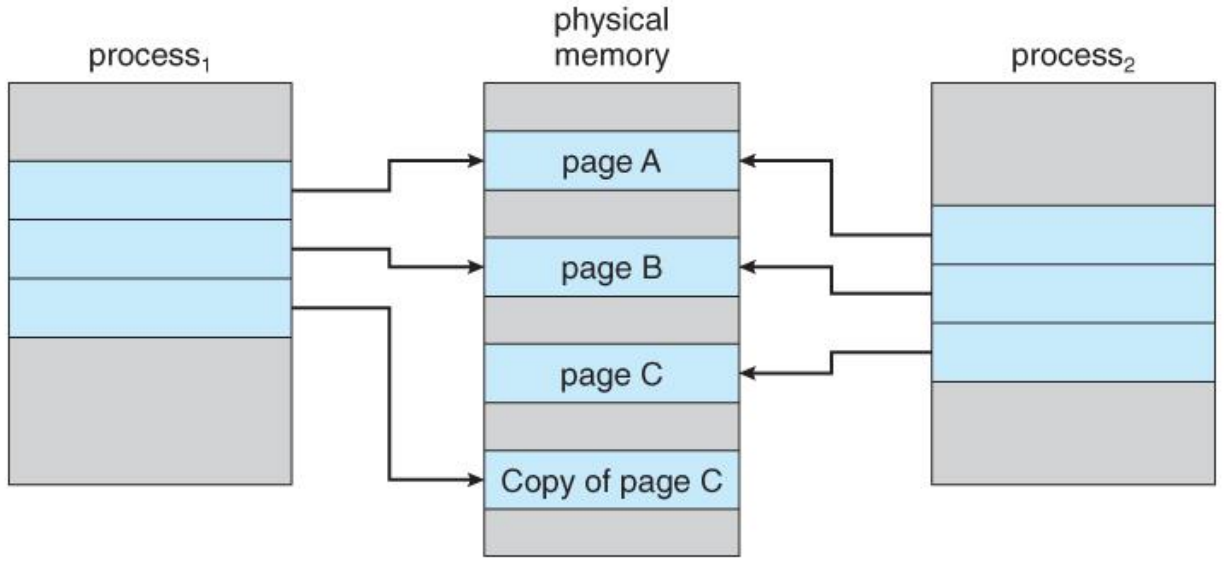
- It can slow down the system performance, as data needs to be constantly transferred between the physical memory and the hard disk.
- It can increase the risk of data loss or corruption, as data can be lost if the hard disk fails or if there is a power outage while data is being transferred to or from the hard disk.
- It can increase the complexity of the memory management system, as the operating system needs to manage both physical and virtual memory.

Copy on Write

- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page.
- They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to.
- This is a reasonable approach, since the child process usually issues an exec( ) system call immediately after the fork.



Before process 1 modifies page C



After process 1 modifies page C

Copy on Write

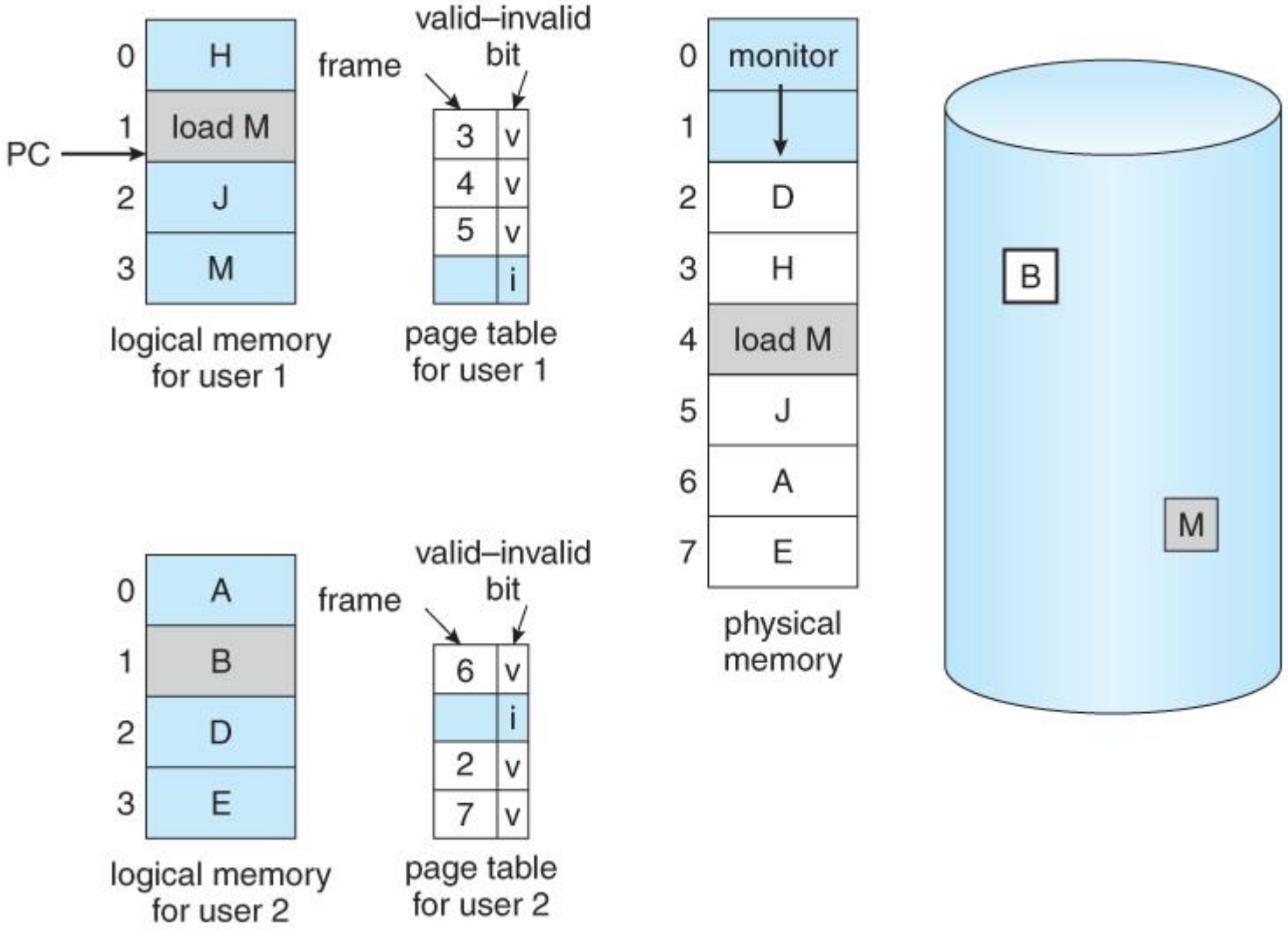
- Obviously only pages that can be modified even need to be labeled as copy-on-write. Code segments can simply be shared.
- Pages used to satisfy copy-on-write duplications are typically allocated using ***zero-fill-on-demand***, meaning that their previous contents are zeroed out before the copy proceeds.
- Some systems provide an alternative to the fork( ) system call called a ***virtual memory fork, vfork( )***.
- In this case the parent is suspended, and the child uses the parent's memory pages.
- This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the exec( ) system call.
- (In essence this addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime.

Page Replacements

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available?
- There are several possible solutions to consider:

1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
2. Put the process requesting more pages into a wait queue until some free frames become available.
3. Swap some process out of memory completely, freeing up its page frames.
4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as page replacement, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.

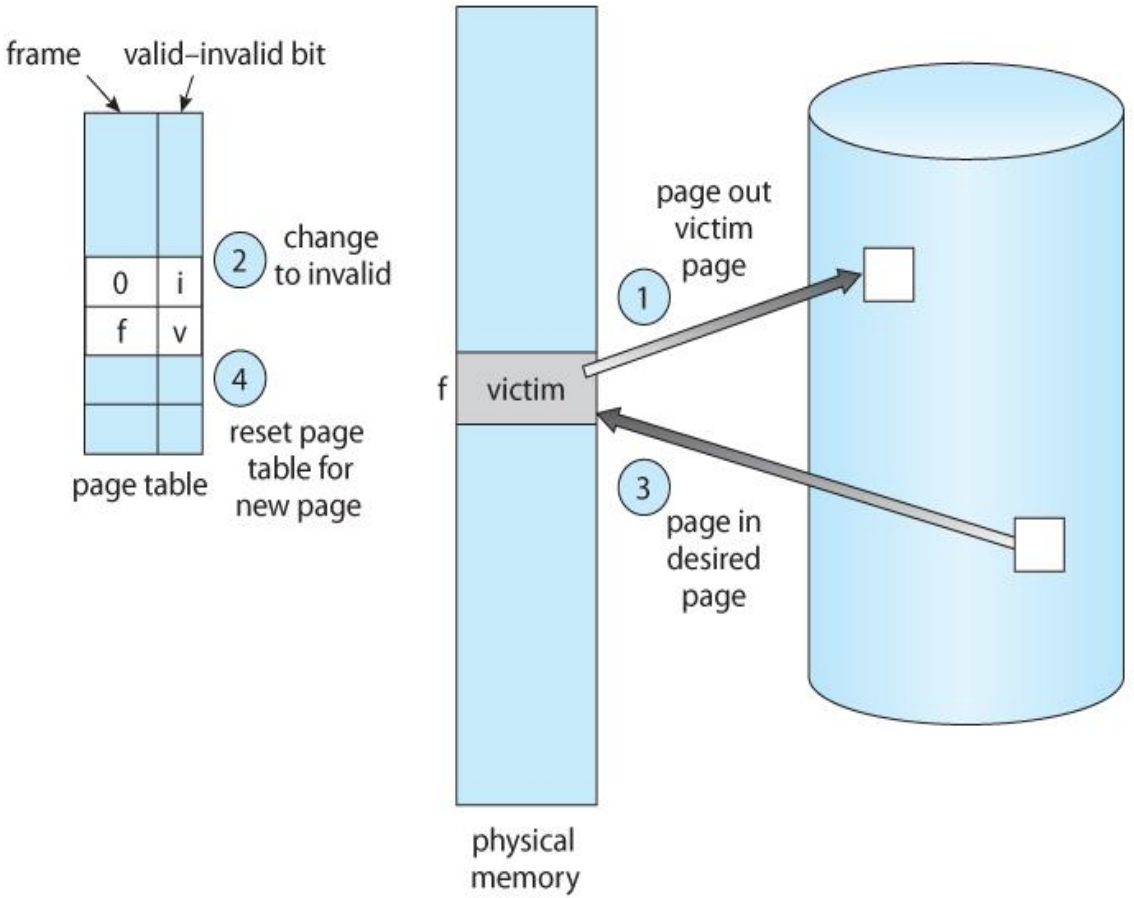
Page Replacements



Ned for page replacement

Basic Page Replacements

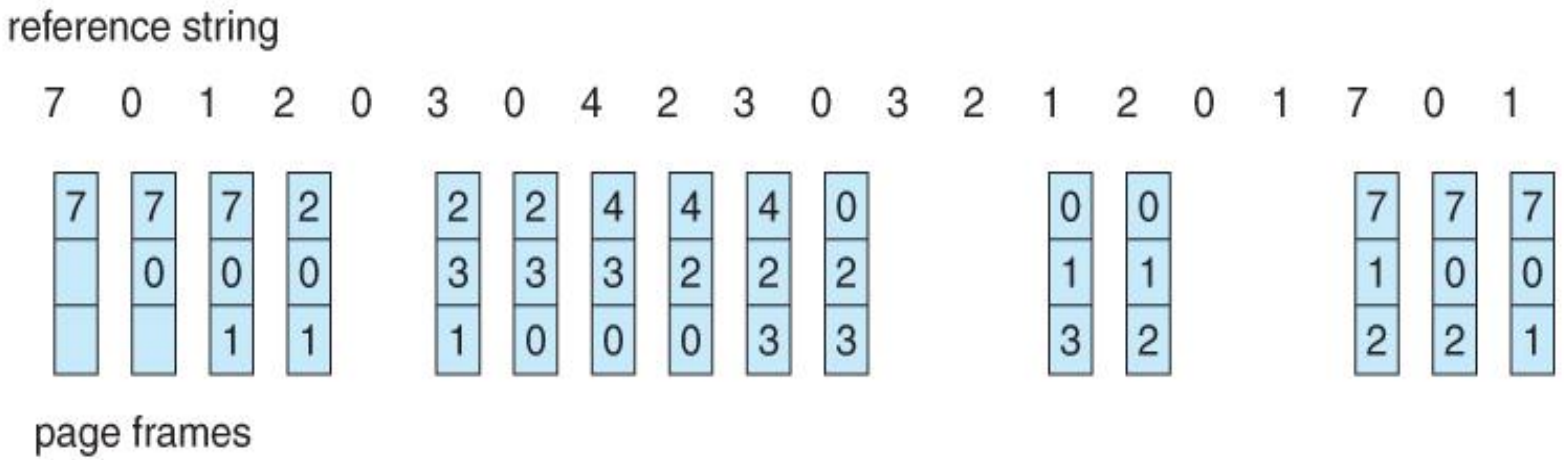
- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list.
- Now the page-fault handling must be modified to free up a frame if necessary, as follows:
  1. Find the location of the desired page on the disk, either in swap space or in the file system.
  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 an existing frame to be replaced, known as the ***victim frame***.
     c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
  3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
  4. Restart the process that was waiting for this page.



Page replacement

FIFO Page Replacement

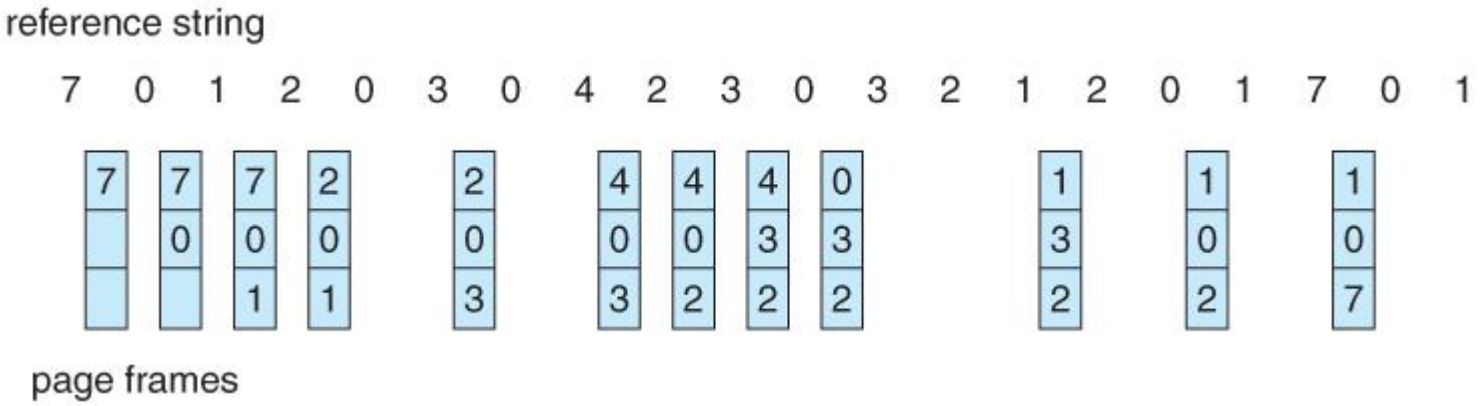
- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:



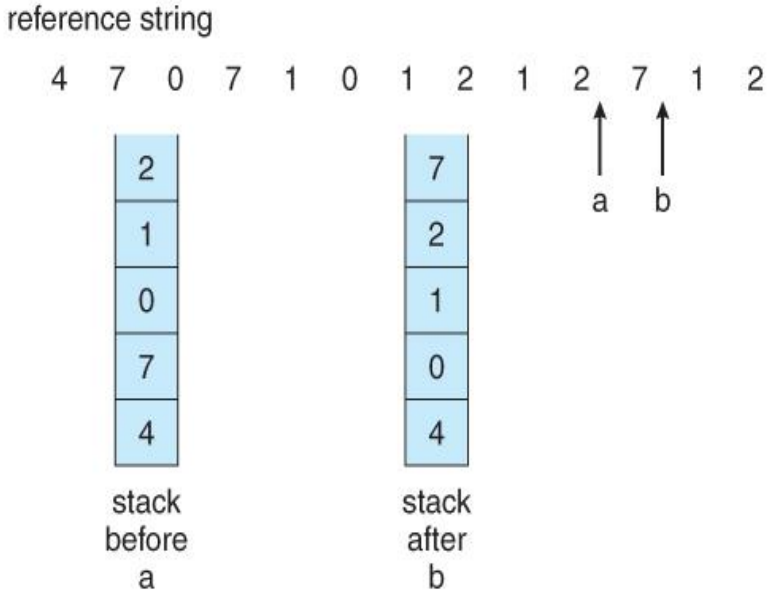
- Although FIFO is simple and easy, it is not always optimal, or even efficient.
- An interesting effect that can occur with FIFO is ***Belady's anomaly***, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur!
- Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames.
- Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:

LRU Page Replacement

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest ***load*** time, and the latter looks at the oldest ***use*** time.)
- Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)
- Following Figure illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)



LRU page-replacement algorithm



Use of a stack to record the most recent page references

LRU Page Replacement

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it.
- There are two simple approaches commonly used:
  - **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.
  - **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.
- Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for ***every*** memory access.
- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called ***stack algorithms,*** which can never exhibit Belady's anomaly.
- A stack algorithm is one in which the pages kept in memory for a frame set of size N will always be a subset of the pages kept for a frame size of N + 1. In the case of LRU, (and particularly the stack implementation thereof), the top N pages of the stack will be the same for all frame set sizes of N or anything larger.

Allocation of Frames

- We said earlier that there were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

Minimum Number of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction.
- If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch.
- Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also.
- The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously.
- For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs. This example would still require a minimum frame allocation of 17 per process.

Allocation Algorithms

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i, and S is the sum of all S_i, then the allocation for process P_i is a_i = m * S_i / S.
- Variations on proportional allocation could consider priority of process rather than just their size.
- Obviously all allocations fluctuate over time as the number of available free frames, m, fluctuates, and all are also subject to the constraints of minimum allocation. (If the minimum allocations cannot be met, then processes must either be swapped out or not allowed to start until more free frames become available.)

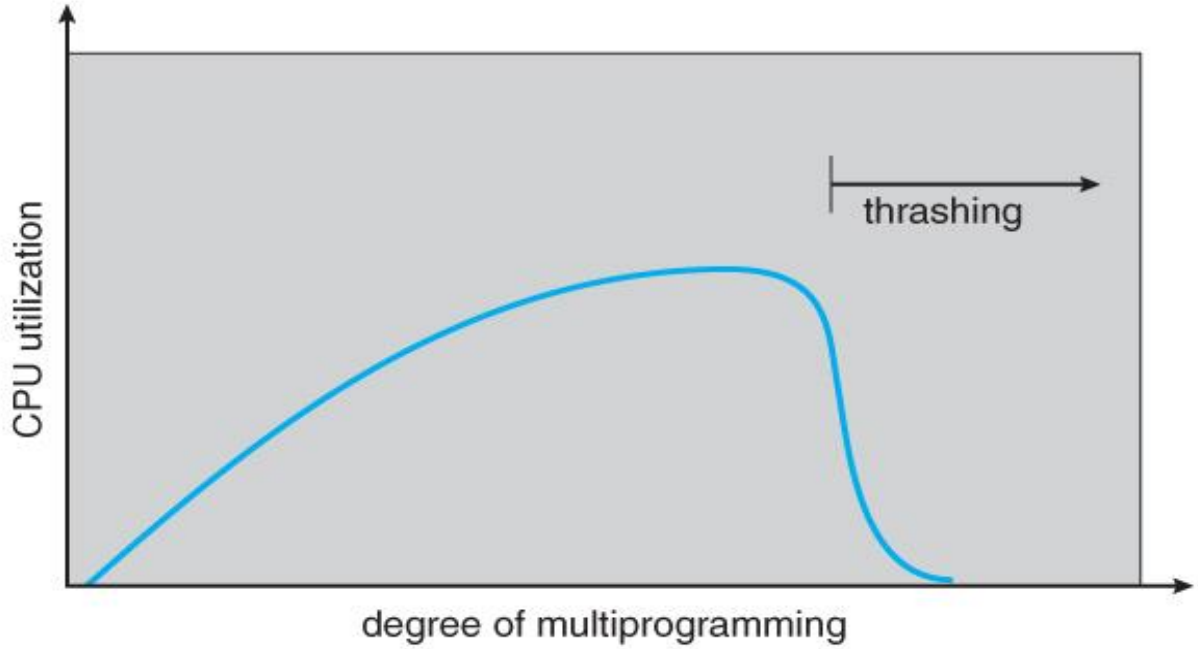
Allocation Algorithms

Global versus Local Allocation

- One big question is whether frame allocation (page replacement) occurs on a local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be ***thrashing.***



Thrashing

- **Cause of Thrashing**
  - Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
  - The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem!
  - Eventually the system would essentially grind to a halt.
  - Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)

Allocating Kernel Memory

- Previous discussions have centered on process memory, which can be conveniently broken up into page-sized chunks, and the only fragmentation that occurs is the average half-page lost to internal fragmentation for each process (segment.)
- There is also additional memory allocated to the kernel, however, which cannot be so easily paged.
- Some of it is used for I/O buffering and direct access by devices, example, and must therefore be contiguous and not affected by paging. Other memory is used for internal kernel data structures of various sizes, and since kernel memory is often locked (restricted from being ever swapped out), management of this resource must be done carefully to avoid internal fragmentation or other waste. (I.e. you would like the kernel to consume as little memory as possible, leaving as much as possible for user processes.)
- Accordingly there are several classic algorithms in place for allocating kernel memory structures.

Allocating Kernel Memory

- The Buddy ***System*** allocates memory using a ***power of two allocator***.
- Under this scheme, memory is always allocated as a power of 2 (4K, 8K, 16K, etc), rounding up to the next nearest power of two if necessary.
- If a block of the correct size is not currently available, then one is formed by splitting the next larger block in two, forming two matched buddies. (And if that larger size is not available, then the next largest available size is split, and so on.)
- One nice feature of the buddy system is that if the address of a block is exclusively ORed with the size of the block, the resulting address is the address of the buddy of the same size, which allows for fast and easy ***coalescing*** of free blocks back into larger blocks.
  - Free lists are maintained for every size block.
  - If the necessary block size is not available upon request, a free block from the next largest size is split into two buddies of the desired size. (Recursively splitting larger size blocks if necessary.)
  - When a block is freed, its buddy's address is calculated, and the free list for that size block is checked to see if the buddy is also free. If it is, then the two buddies are coalesced into one larger free block, and the process is repeated with successively larger free lists.

धन्यवाद