OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing System & Organisation

Lecture 7: Memory Management

Content

1. Memory management
2. Address space of a process
3. Segmentation
4. Paging

Memory Management

Memory Management

- In multiprogramming systems, the user part of memory is subdivided to accommodate multiple processes
- The task of subdivision is carried out by the operating system and is known as ***memory management***
- Memory needs to be allocated efficiently to pack as many processes into memory as possible

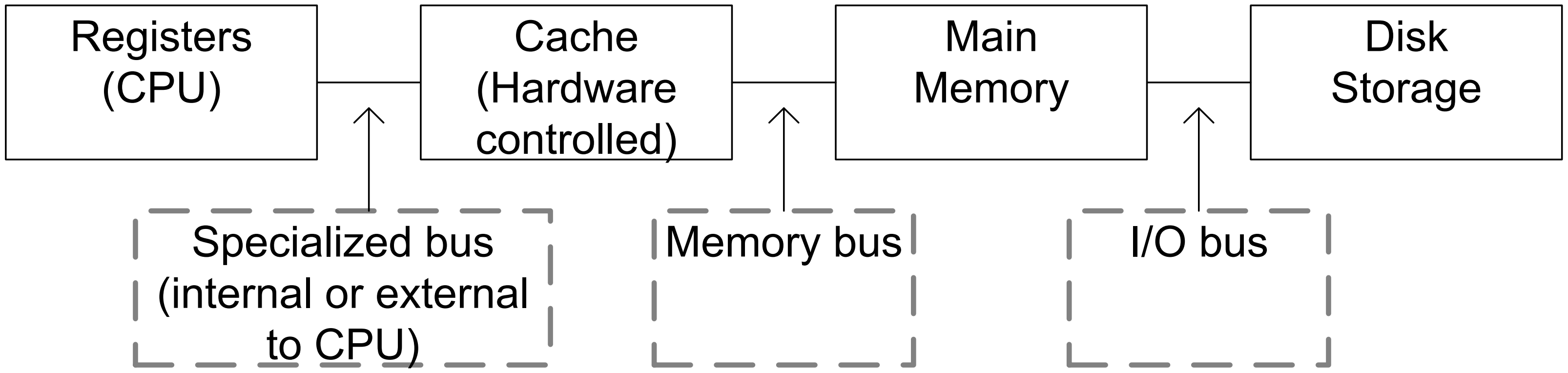
Memory Management Requirements

- **Relocation**
  - Loading dynamically the program into an arbitrary memory space, whose address limits are known only at execution time
- **Protection**
  - Each process should be protected against unwanted interference from other processes
- **Sharing**
  - Any protection mechanism should be flexible enough to allow several processes to access the same portion in the main memory

Memory Organisation

- **Logical organisation**
  - Most programs are organised in modules
    - Some modules are un-modifiable (read only and/or execute only)
    - Others contain data that can be modified
  - The operating system must take care of the possibility of sharing modules across processes
- **Physical organisation**
  - Memory is organised as at least a two-level hierarchy.
  - The OS should hide this fact and should perform the data movement between the main memory and secondary memory without the programmer’s concern

Memory Hierarchy Review



- It is a tradeoff between size, speed and cost
- **Register**
  - Fastest memory element; but small storage; very expensive
- **Cache**
  - Fast and small compared to main memory; acts as a buffer between the CPU and main memory: it contains the most recent used memory locations (*address* and *contents* are recorded here)
- **Main memory** is the RAM of the system
- **Disk storage** - HDD

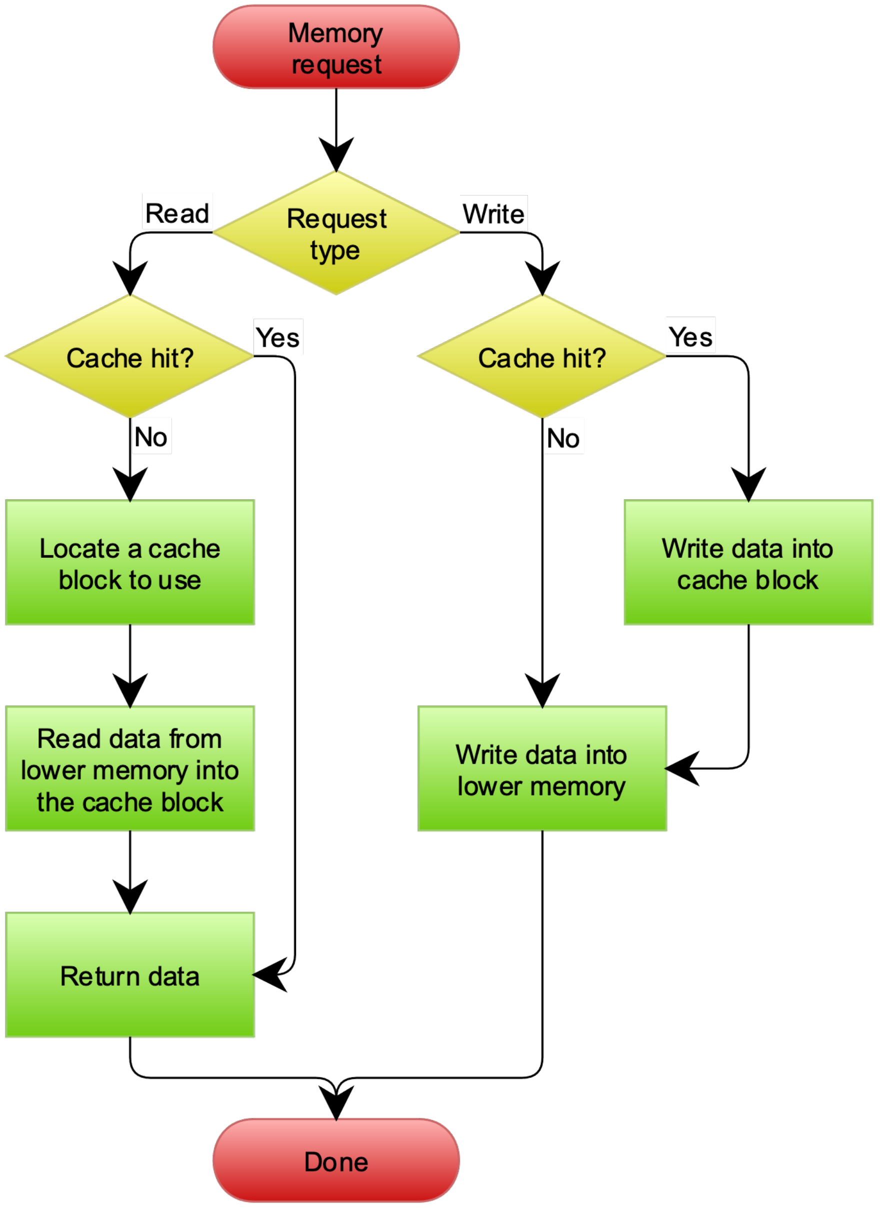
Caching

- Reading from cache is faster than recomputing a result or reading from a slower data store
  - thus, the more requests that can be served from the cache, the faster the system performs.
- When reading data from a lower memory, also store a copy in the cache
  - Future requests for that data can be served faster
- A ***cache hit*** occurs when the requested data can be found in a cache, while a ***cache miss*** occurs when it cannot.

Cache review

- Typical computer applications access data with a high degree of locality of reference:
  - ***Temporal locality***: data is requested that has been recently requested already
  - ***Spatial locality***: data is requested that is stored physically close to data that has already been requested

- When a system writes data to cache, it must at some point write that data to the main memory as well following the ***Write policies:***
  - ***Write-through***: write is done synchronously both to the cache and to main memory
  - ***Write-back***: initially, writing is done only to the cache. The write to main memory is postponed until the modified content is about to be replaced by another cache block.



https://commons.wikimedia.org/w/index.php?curid=27652294

Process Address Space

Process Address Space

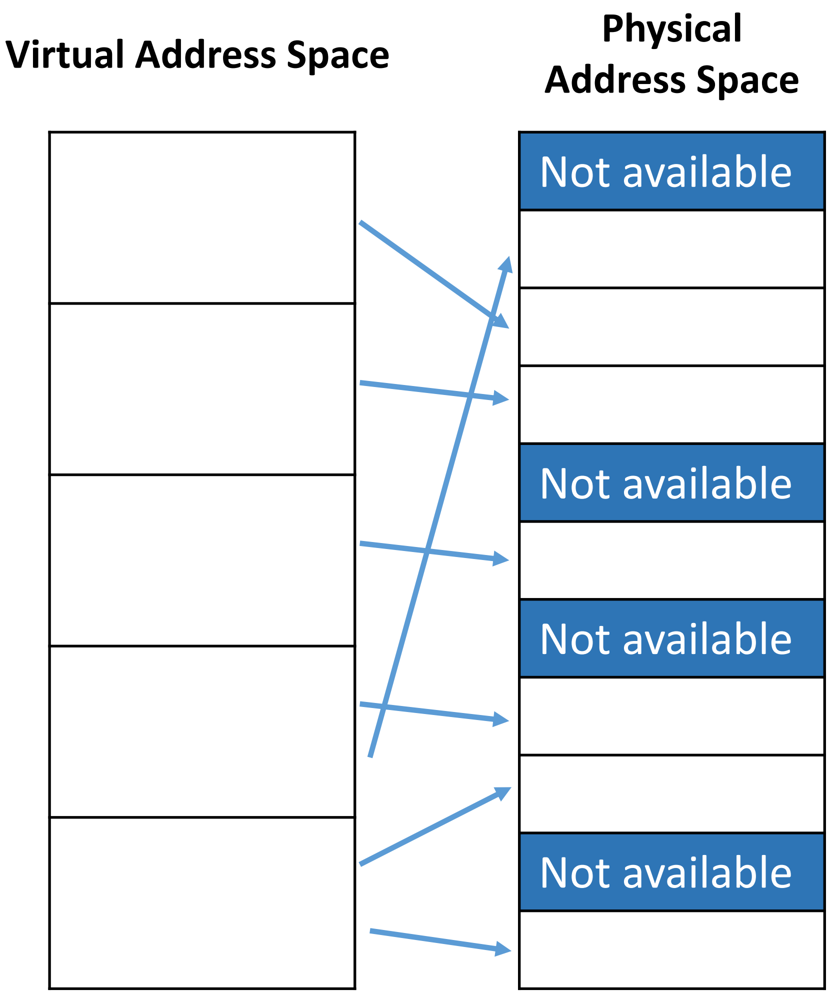
- When accessing memory, a process is said to operate within an ***address space*** (data items are accessible within the range of addresses available to the process)

- The number of bits allocated to specify the address is an ***architectural decision***
  - Many early computers had:
    - 16 bits for address (thus allowing for a space of 64KB of direct addressing -> 2^{16})
    - Then, 32 bits, which allows for 4GB of direct addressing memory space
  - Now most computers had 64 bits for addresses
    - We say that such a system gives a ***virtual address space*** of 16 ExaBytes (16 billion gigabytes)
    - Although, the amount of physical memory in such a system is likely to be less than this)

Address Binding

- An address used in an instruction can point ***anywhere*** in the virtual address space of the process
    - It still must be ***bound*** to a physical memory address
- Programs are made of modules.
- Compilers or assemblers ***do not know where the module will be loaded*** in the physical memory
    - ➢ Virtual addresses must be translated to physical addresses

➢ Address translation can be **dynamic** or **static**.



Static Address Binding

- OS is responsible for managing the memory, so it will give the loader ***a base address where to load the module***
  - The loader **converts** each virtual addresses in the module to absolute physical addresses by adding the the base address
  - This is called ***static binding***

- Simple/Easy to Implement
- But,
  - Once loaded, the code or data of the program cannot be moved into another part of memory without change in the static binding
  - All the processes executing in such a system would share the same physical address space
    - no protection from one another if addressing errors occur
    - even the OS code is exposed to addressing errors

Dynamic Address Binding

- Dynamic address binding:
  - Keeps loaded addresses ***relative*** to the start of a process
- Advantages of dynamic address binding:
  - A given program can ***run anywhere*** in the physical memory and ***can be moved around*** by the operating system
  - All of the addresses that it is using are relative to its own virtual address space, so it is ***unaware of the physical locations*** at which it happens to have been placed
  - It is possible to protect processes from each other and protect the operating system from application processes by a mechanism we employ for isolating the addresses seen by the processes
- Disadvantage:
  - A mechanism is needed to bind the virtual addresses within the loaded instructions to physical addresses when the instructions are executed

Hardware Assisted Relocation and Protection

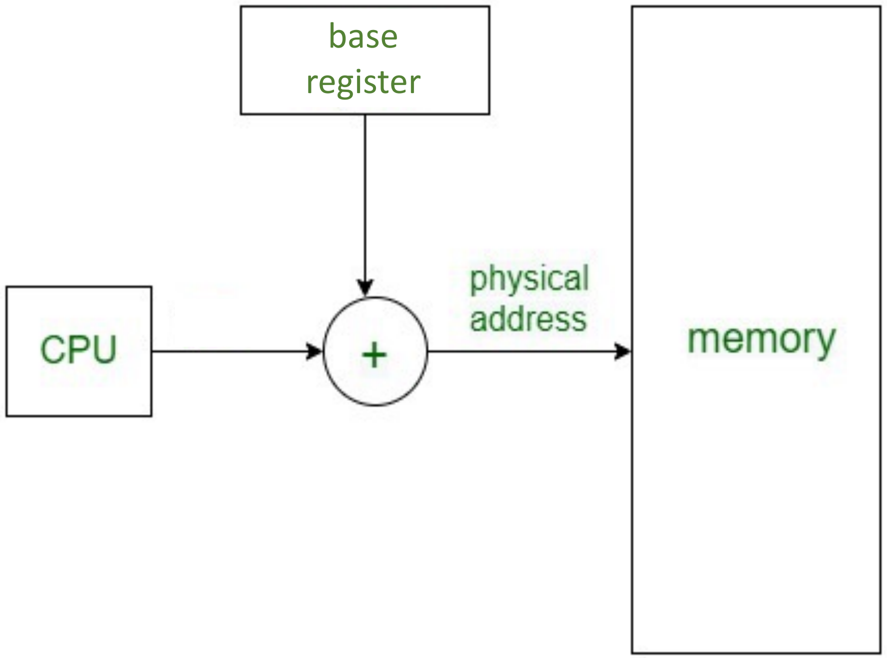
- Dynamic binding must be ***implemented in hardware***, since it introduces translation as part of every memory access

- If the basic requirement for modules is to be held **contiguously** in physical memory and contain addresses relative to their first location:
  - The first location is called the ***base*** of the process
- Suppose that an instruction is fetched and decoded and contains an address reference
  - This address reference is relative to the base, so the value of the base must be added to it (***base + address reference)*** in order to obtain the correct physical address to be sent to the memory controller

Hardware Relocation and Protection

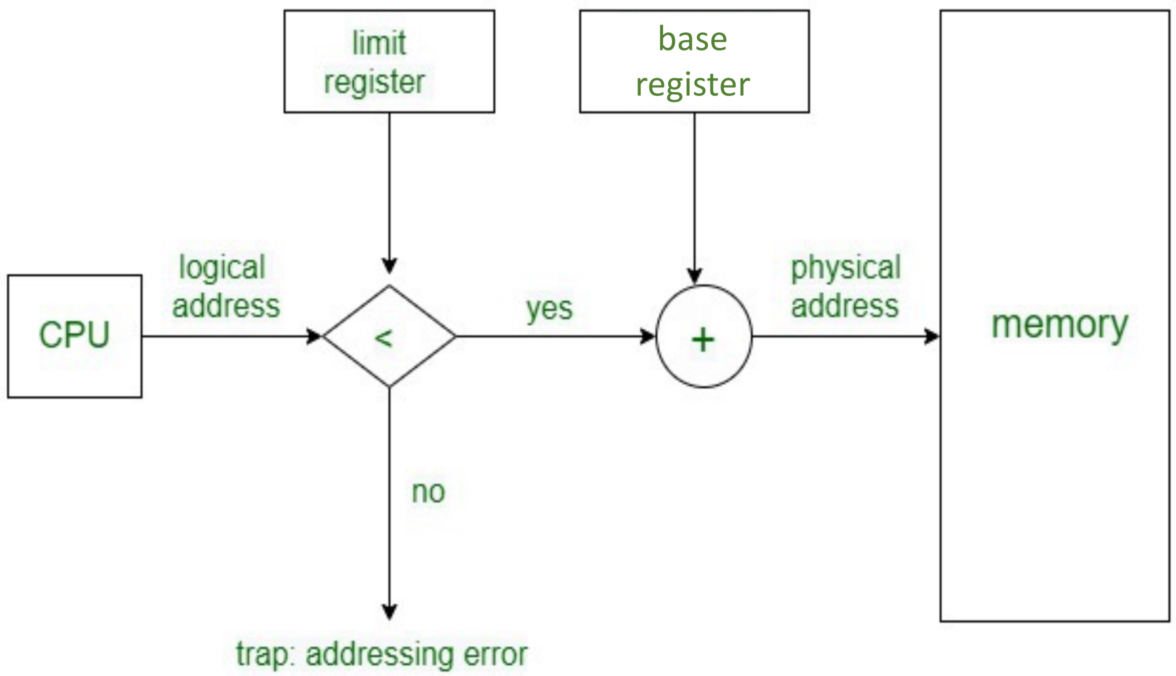
- The simplest form of dynamic binding hardware is a ***base register*** and a memory management unit (MMU*)* to perform the translation
  - The operating system must load the base register as part of setting up the state of a process before passing control to it

- Problem: This approach does not provide any protection between processes:
  - We cannot be sure that a process does not use an address that is not in its space.



Hardware Relocation and Protection

- Solution: combine the relocation and protection functions in one unit
  - By adding a second register (the ***limit register***) that delimits the upper bound of the program in the physical memory



Segmentation

Segmented Virtual Memory

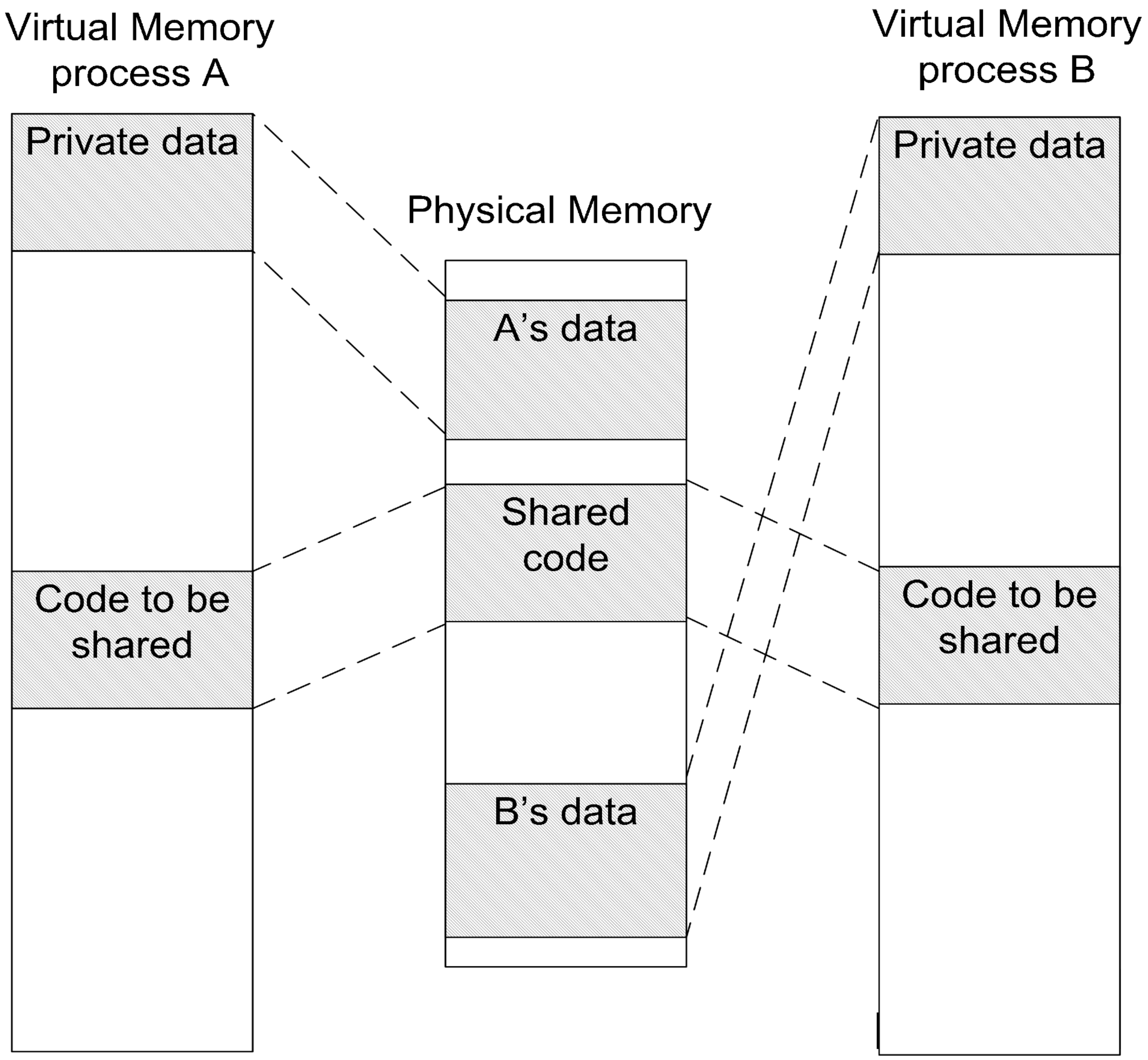
- In practice, it is **not very useful** for a program to occupy a single **contiguous** range of physical addresses

- Such as scheme would prevent two processes from sharing the code
  - i.e., using this scheme, it is difficult to arrange two executions of same program (two processes) to access different data while still being able to share same code

- This can be achieved if the system has two base registers and two limit registers, thus allowing two separate memory ranges or ***segments*** per process

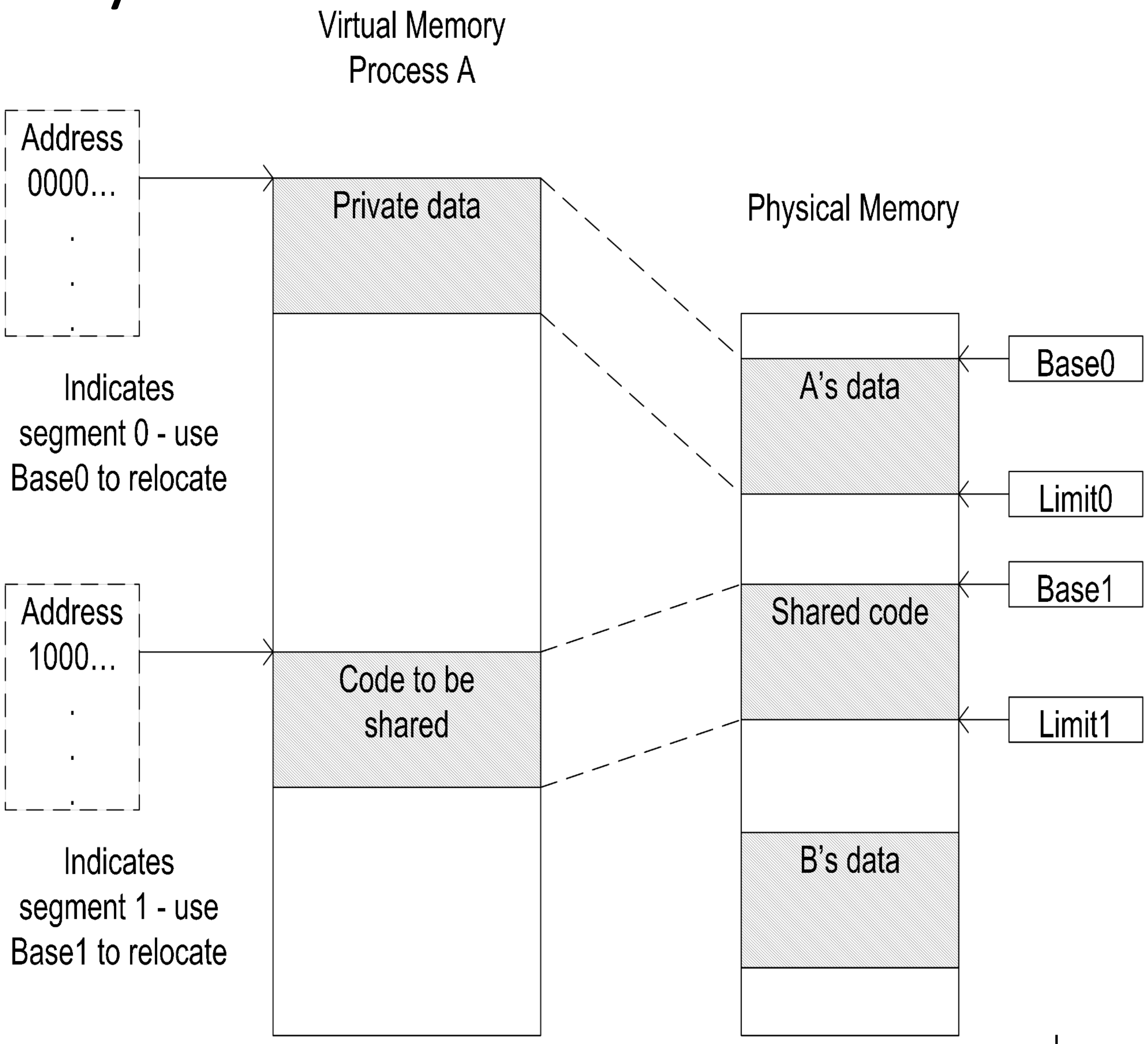
Segmented Virtual Memory

Two processes sharing a code segment but having private data segments



Segmented Virtual Memory

Most significant bit of the virtual address is taken as a ***segment identifier***, with 0 for data segment and 1 for code segment

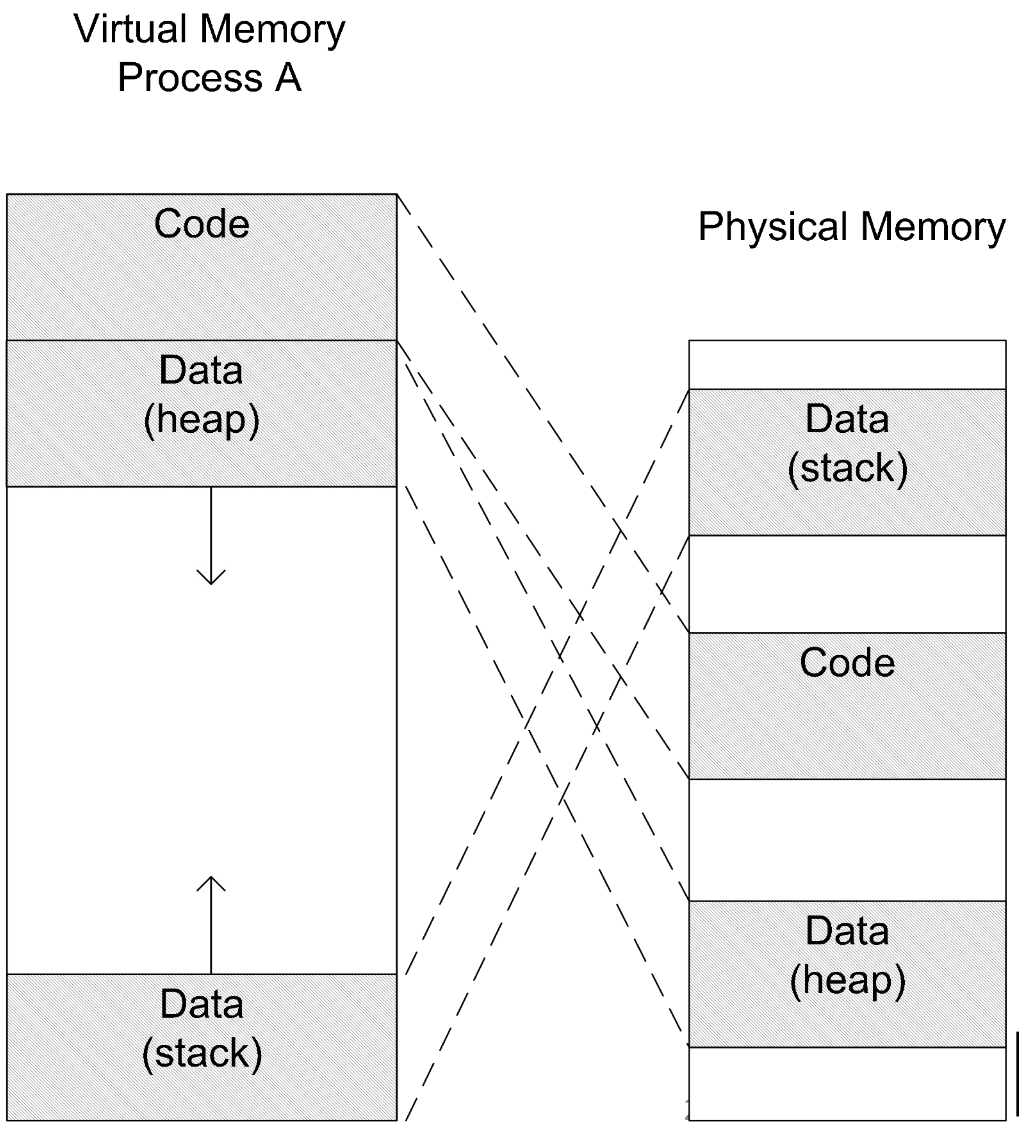


Segmented Virtual Memory

- Within a single program, it is usual to have separate areas for ***code, stack*** and ***heap***;

- Language systems have conventions on how the virtual address space is arranged
  - Code segment will not grow in size
  - Heap (may be growing)
  - Stack at the top of virtual memory, growing in opposite direction than Heap

- In order to realize the relocation (and protection), three segments would be preferable



Segmented Virtual Addresses

- The segment is the unit of protection and sharing
  - the more we have, the more flexible
- 2 ways to organise segmented address:

1. Virtual address space is split into a ***segment number*** and a ***byte number*** within a segment
   - The number of bits used for segment addressing is usually fixed by the CPU designer

Maximum number of segments is 2^x	Maximum segment size is 2^y
Segment number X bits	Byte offset in segment Y bits

Virtual Address : address field of an instruction

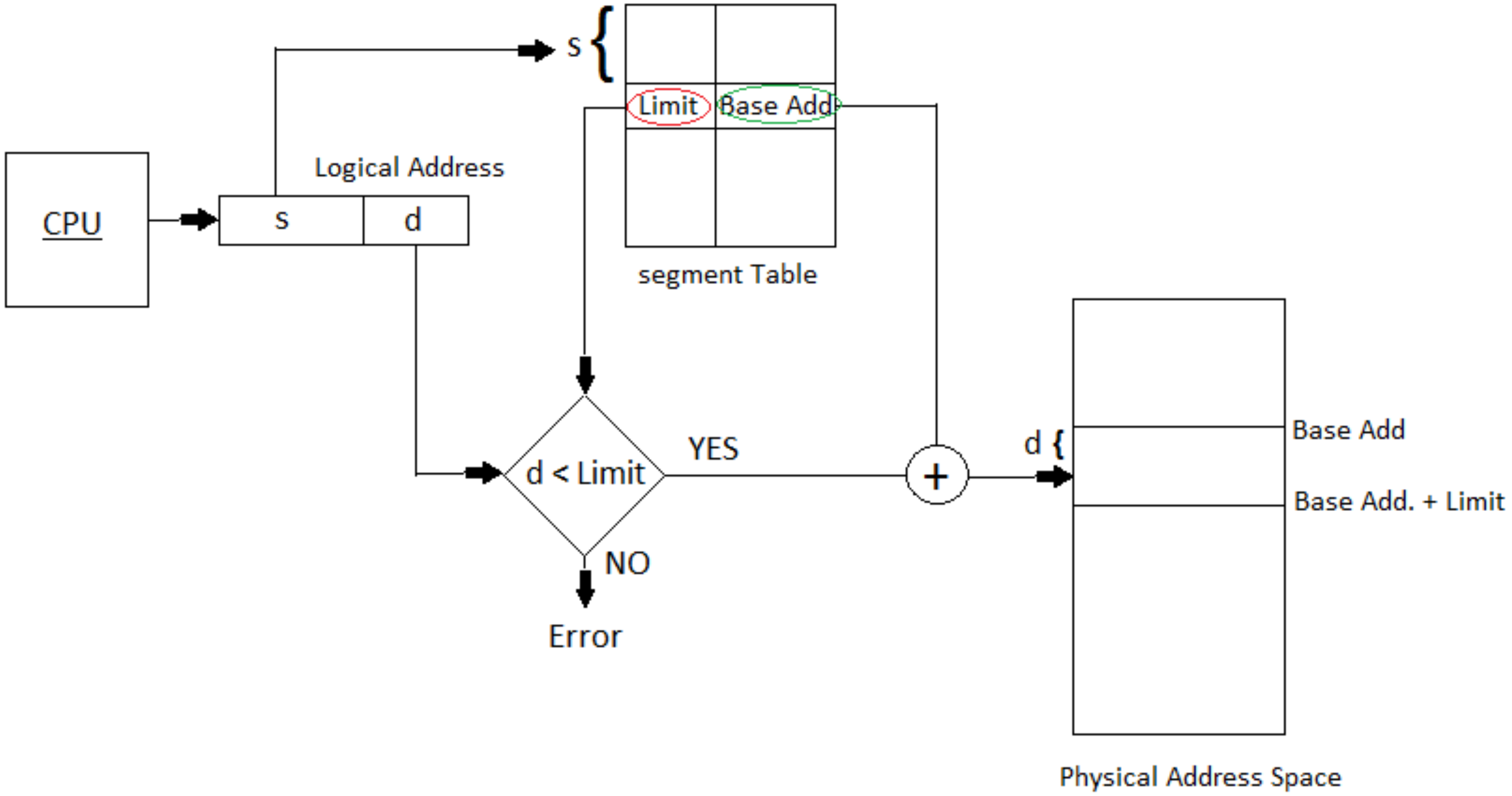
2. The segment number is supplied separated from the offset portion of the address.
   - This is done in X86 processors

Segmented Address Translation

- For dynamic address translation in the operating system
  - Hardware must keep a ***segment table*** for each process in which the location of each segment is recorded
- A process can have many segments, only those currently being used for instruction fetch and operand access need to be in main memory
  - other segments could be held on backing store until they are needed.
- If an address is presented for a segment that is not present in main memory, then the address translation hardware generates an ***addressing exception***.
  - This is handled by the operating system, causing the segment to be fetched into main memory and the mechanism restarted

	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Address Translation in Segmentation System



s = number of bits to represent the segment
d = number of bits to represent the size of the segment
limit = length of the segment
base add = initial physical address in memory

Segmentation Summary

- A process is divided into ***a number of segments*** that do not need to be equal in size
- When a process is brought into the main memory, all of its segments are usually brought into the main memory and ***a process segment table*** is setup.
- **Advantages**:
    - The virtual address space of a process is divided into logically distinct units which correspond to constituent parts of a process
    - Segments are the natural units of access control
        - Rrocesses may have different access rights for different segments and sharing code/data with other processes
- **Disadvantages**:
    - Inconvenient for operating system to manage storage allocation for variable-sized segments
    - After the system has been running for a while, the free memory available can be fragmented
    - ***External fragmentation:*** sometimes, even though the total free memory might be far greater than the size of some segment that must be loaded, there is no single area large enough to load it

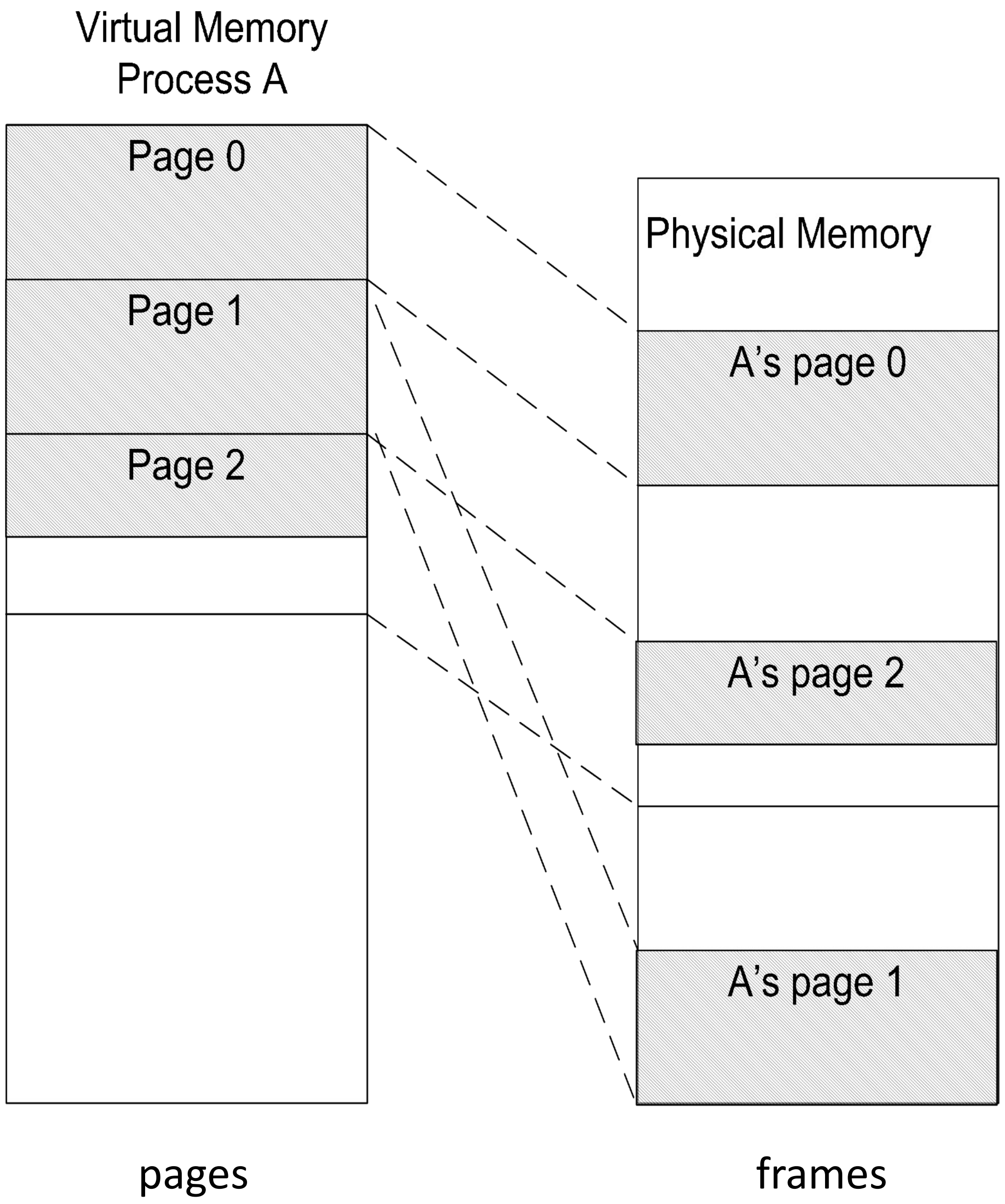
Paging

Paged Virtual Memory

- The need to keep each loaded segment contiguous in the physical memory poses a significant disadvantage:
  - It leads to fragmentation
  - It complicates the physical storage allocation problem

- Solution: ***paging***, where blocks of a fixed size are used for memory allocation (so that if there is any free space, it is of the right size)

- Memory is divided into page ***frames***, and the user program is divided into ***pages*** of the same size



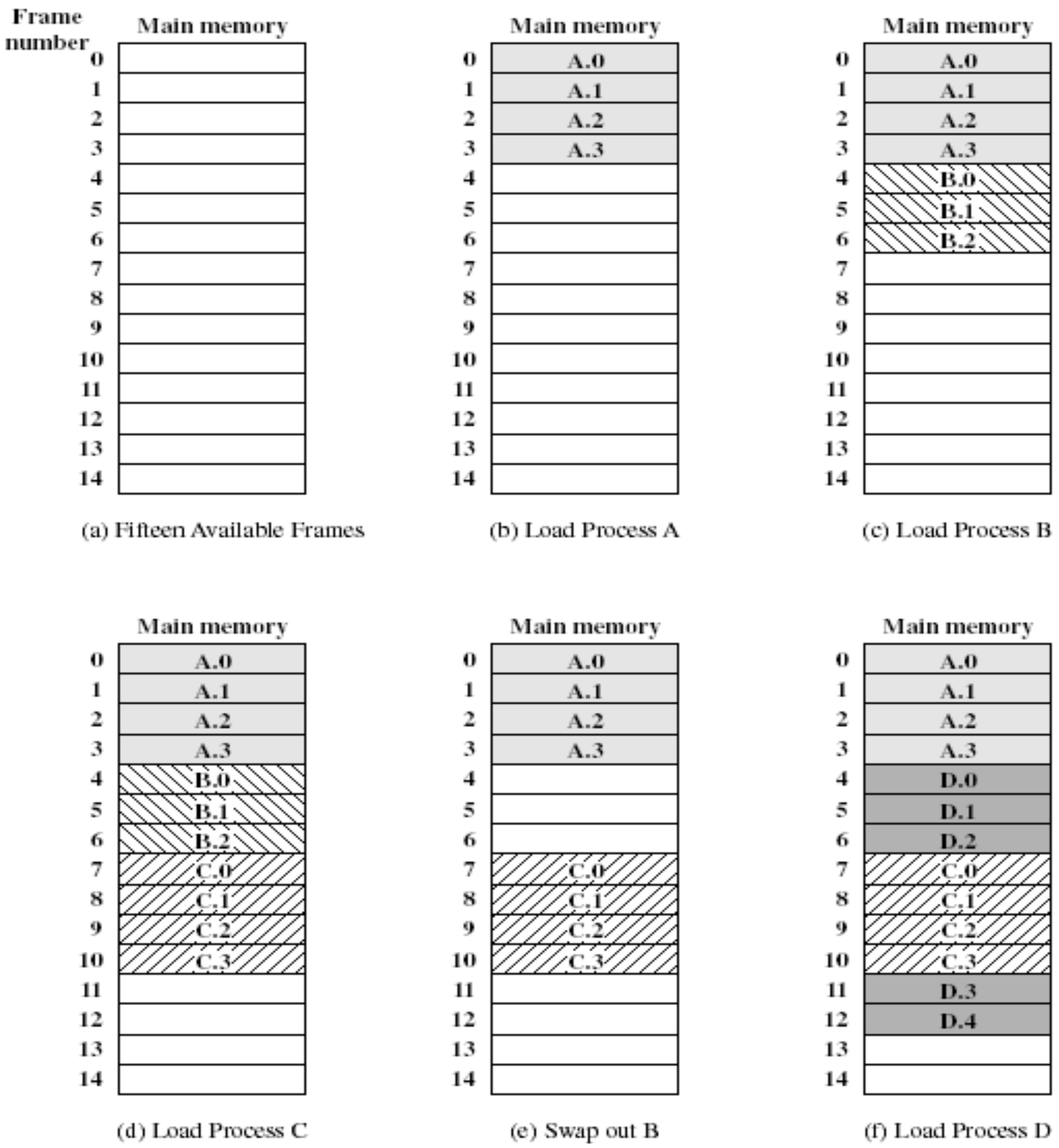
Paged Virtual Memory

- Typical page size is small (1 to 4KB)
  - In paged systems, a process would require many pages
- The limited size of physical memory can cause problems. Therefore,
  - a portion of the disk storage could be used as extension to the main memory (backing store)
  - and the pages of a process may be in the main memory and/or in this backing store
- The operating system ***must manage the two levels of storage and the transfer of pages*** between them
- It must keep a ***page table*** for each process to record information about the pages
  - A ***present bit*** is needed to indicate whether the page is in main memory or not
  - A ***modify bit*** indicates if the page has been altered since last loaded into main memory
  - If not modified, the page does not have to be written to the disk when swapped out

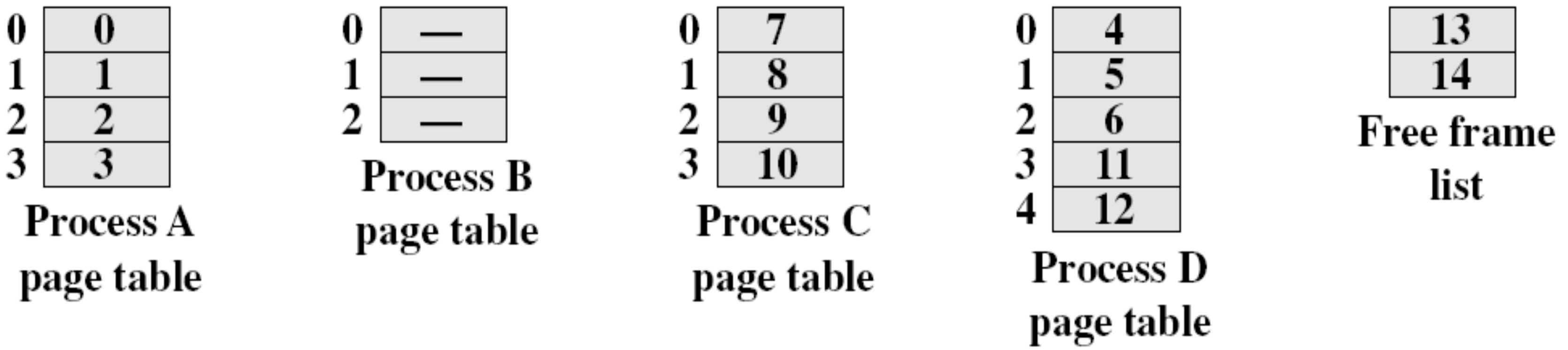
Paging Example

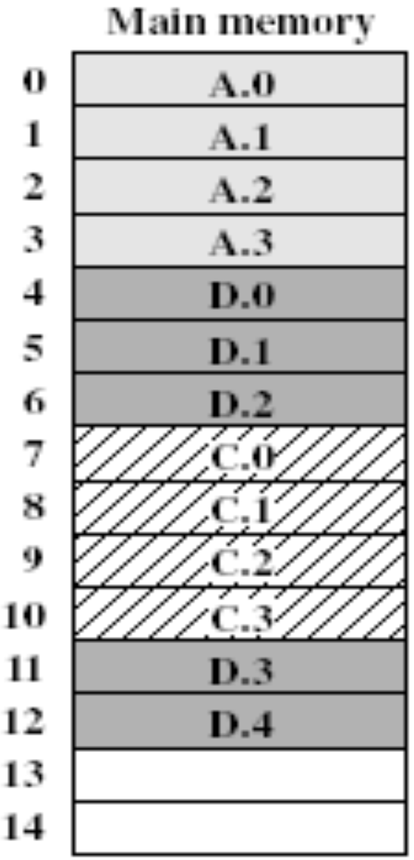
All the processes (A, B, C and D) are stored on disk and are about to be loaded in the memory (by the operating system)

- Process A has four pages
- Process B has three pages
- Process C has four pages
- Process D has five pages



Paging Example





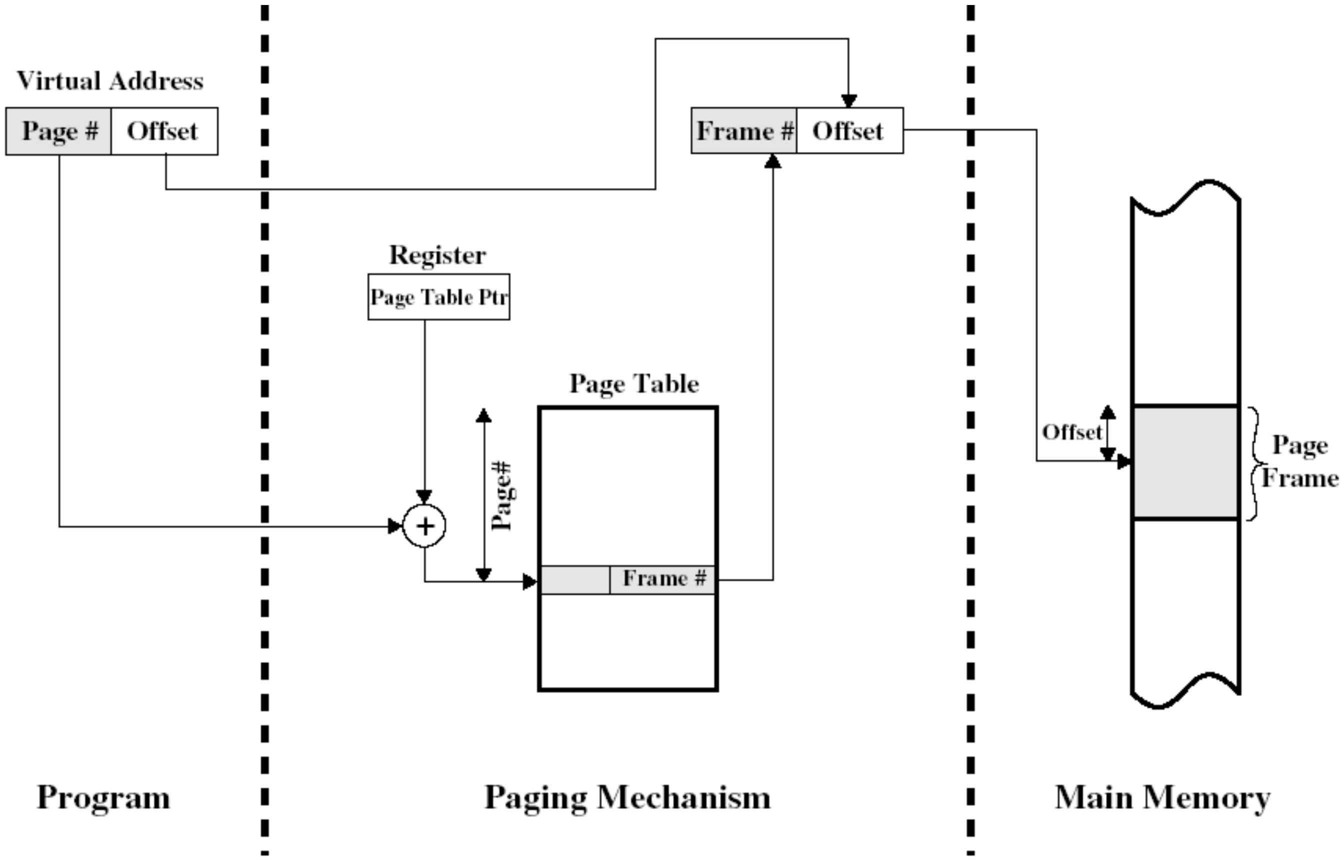
(f) Load Process D

- Various page tables at the time
  - Each Page Table Entry (PTE) contains the number of the frame in main memory (if any) that holds that page
  - In addition, typically, the operating system maintains a list of all frames in main memory that are currently unoccupied and available for pages

Ollscoil na Gaillimhe
University of Galway

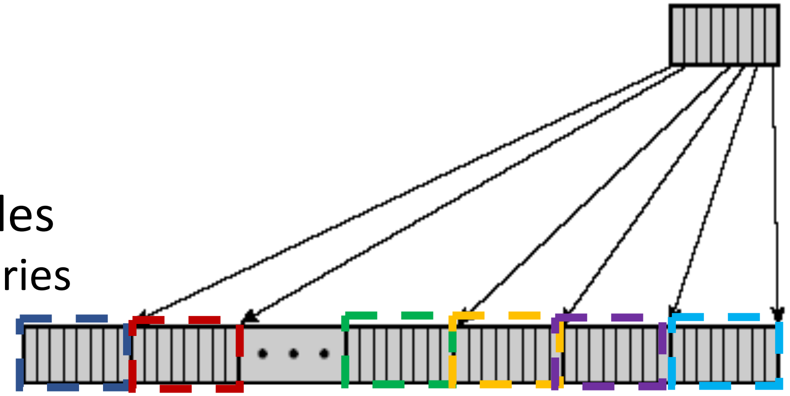
Paged Virtual Memory Address Translation

- Translation of a virtual address (***page*** + offset) into a physical address (***frame*** + offset)
  - using a page table
- Page table is stored in the main memory
  - Each process maintains a pointer in one of its registers, to the page table
- The page number is used to index that table and lookup the corresponding frame number
- Combining the frame number with the offset from the virtual address gives the real physical address



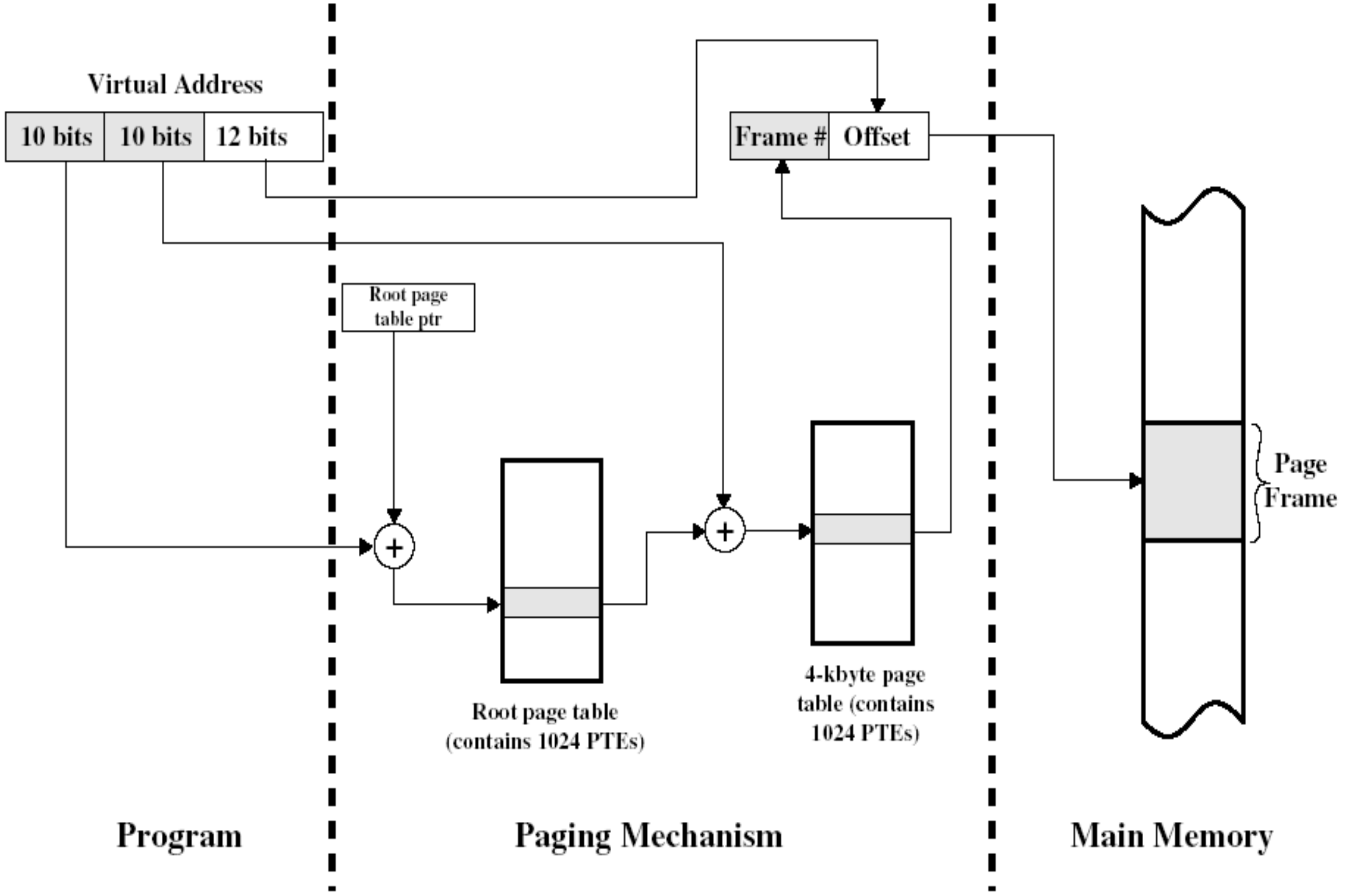
Paged Virtual Memory Address Translation

- Processes could occupy ***huge amounts of virtual memory***
  - E.g., in a 32bit addressing system with pages of size 4KB:
    - 12 bits for offset
    - 20 bits for number of pages
  - This means 2^{20} entries could be in each page table
    - If each entry occupies 4Bytes (32bit address)
    - Then *each page table would take 4MB*
      - ➢ Unacceptably high!
- Solution: a ***two-level scheme*** to organise large page tables
  - Root Page Table with 2^{10} (1024 entries, 4 Bytes each) entries occupying 4KB of main memory
    - Root page always remains in the main memory
  - User Page Tables can reside in either the main memory or in disk



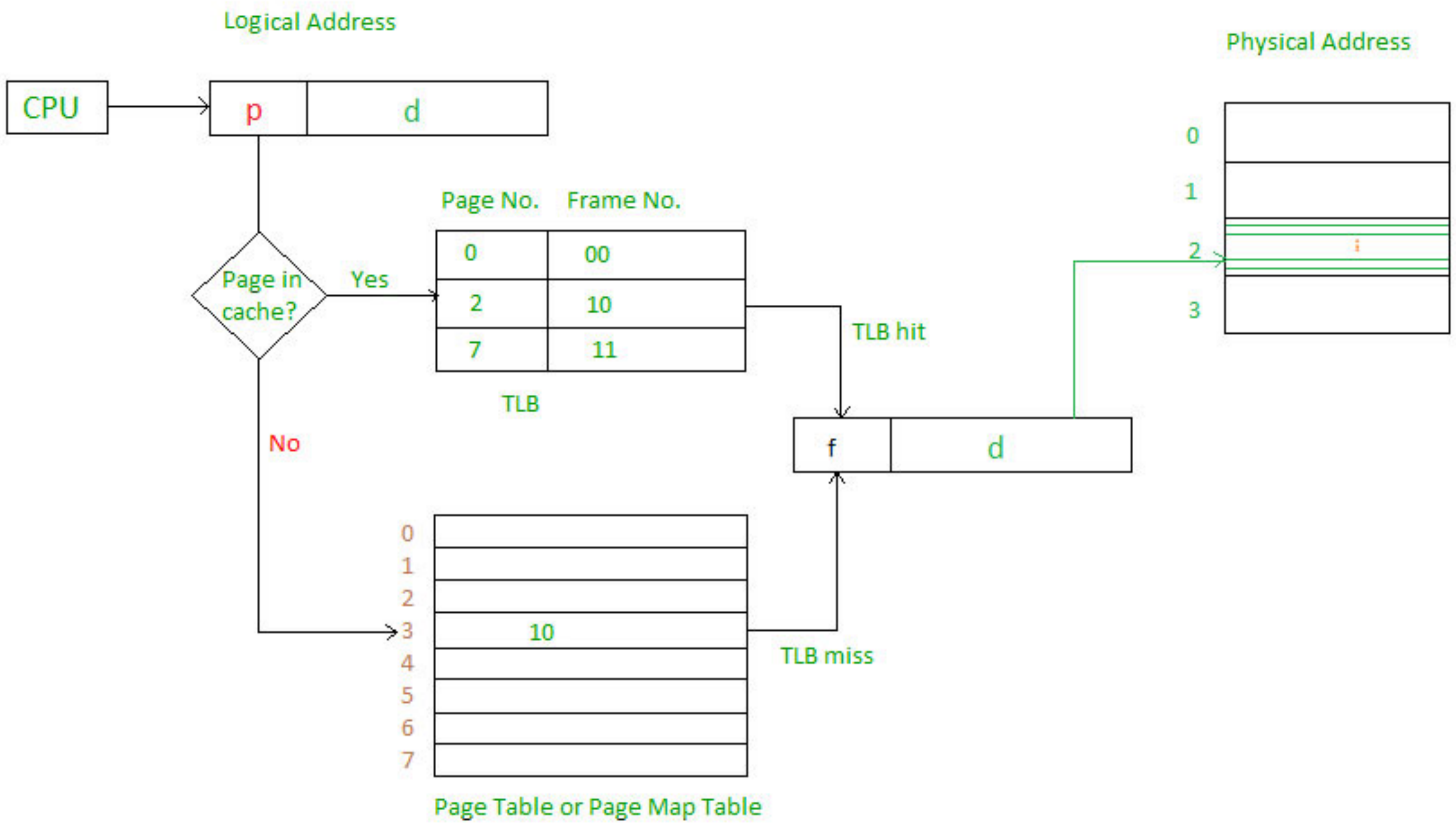
Paged Virtual Memory Address Translation

- The first 10 bits of a virtual address are used to find a PTE to the user page table
- The next 10 bits of the virtual memory address are used find the PTE for the page that is referenced by the virtual address
- Every virtual memory reference causes two physical memory accesses:
  - one to fetch the appropriate User Page Table entry
  - the other to fetch the desired page
- To overcome this, most of the virtual memory schemas make use of a ***special high-speed cache*** for page entries



Translation Lookaside Buffer (TLB)

- **A kind of cache memory:** it contains the page entries that have been most recently used
- TLB is searched for each address reference
- TLB is nearly always present in any processor that utilizes paged or segmented virtual memory
  - Including in most desktops, laptops, and servers.



Translation Lookaside Buffer (TLB)

- The virtual page number is extracted from the virtual address and a lookup is initiated
  - If multiple processes, then special care needs to be taken, so the ***page from one process would not be confused with another's***

- If a match is found (***TLB hit***), then an access check is made, based on the information stored in the flags
  - The physical page base, taken form TLB is appended to the offset from the virtual address to form the complete physical address
  - The flags field will indicate the access rights and other information (i.e. if a write is being attempted to a page that is read only etc)

- If an address reference is made to a page that ***is in the main memory but not in the TLB***, then address translation fails (***TLB miss***) and a new entry in the TLB needs to be created for that page

- If an address reference is made to a page that ***is not in the main memory***, the address translation will fail again. No match will be found in the address table and the addressing hardware will raise an exception, called ***page fault***
  - The operating system will handle this exception

Paging Summary

- **Advantages** – by using fixed size pages in virtual address space and fixed size pages in physical address space, it ***addresses some of the problems with segmentation***:
  - ***External fragmentatio***n is no longer a problem (all frames in physical memory are same size)
  - Transfers to/from disks can be performed at granularity of individual pages

- **Disadvantages**
  - The page size is a choice made by CPU or OS designer
    - It may ***not fit the size of program data structures*** and lead to internal fragmentation in which storage allocation request must be rounded to an integral number of pages
  - There may be no correspondence between ***page protection settings*** and ***application data structures***
    - If two processes are to share data structures, they may do so at the level of sharing entire pages
  - Requiring page tables per process , it is likely that the OS require **more storage** for its internal data structures

References

- “Operating Systems”, William Stallings, ISBN 0-13-032986-X
- “Operating Systems”, Jean Bacon and Tim Harris, ISBN 0-321-11789-1