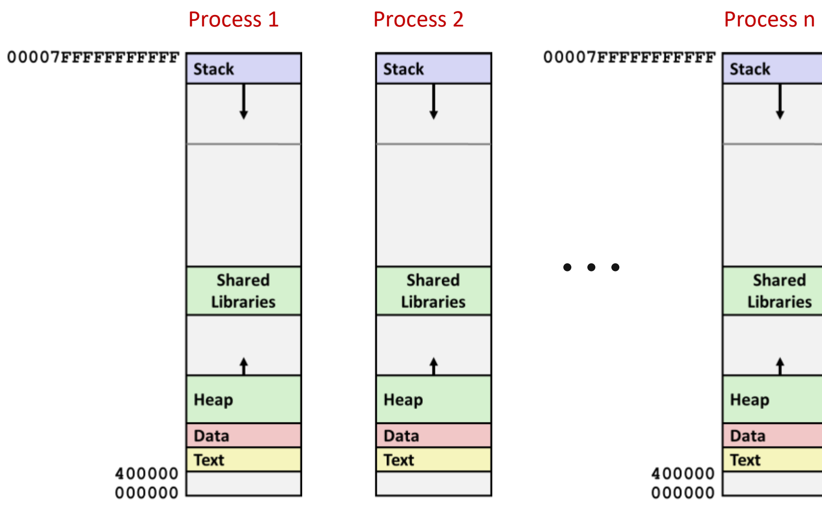
Chapter 3

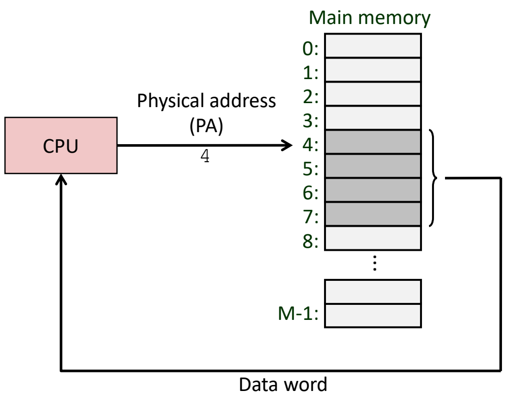
COMPUTER MEMORY
Part 2
(Virtual memory)

Hmmm, How Does This Work?!



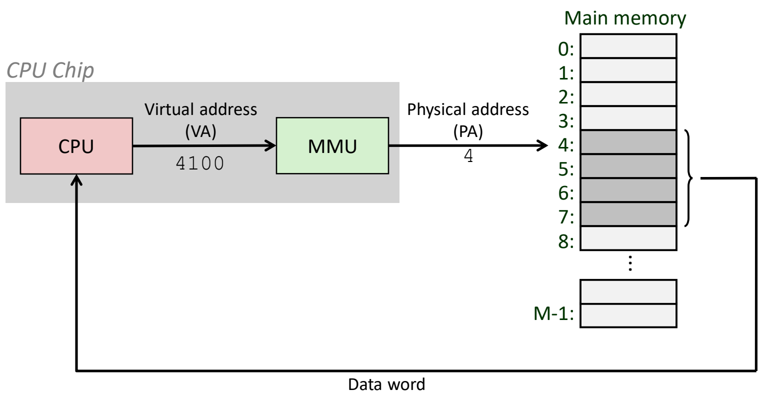
Solution: Virtual Memory (today and next lecture)

A System Using Physical Addressing



- Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Address Spaces

- Linear address space: Ordered set of contiguous non-negative integer addresses:

  {0, 1, 2, 3 … }

- Virtual address space: Set of $N = 2^n$ virtual addresses

  {0, 1, 2, 3, …, N-1}

- Physical address space: Set of $M = 2^m$ physical addresses

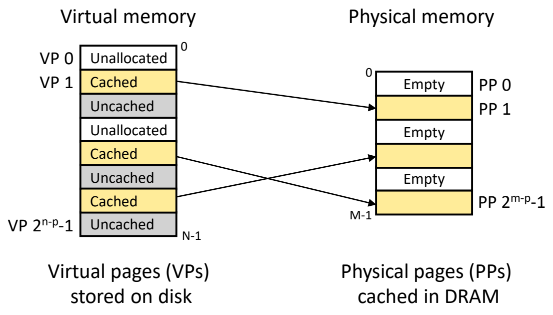
  {0, 1, 2, 3, …, M-1}

Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

VM as a Tool for Caching

- Conceptually, *virtual memory* is an array of N contiguous bytes stored on disk.
- The contents of the array on disk are cached in *physical memory* (*DRAM cache*)
  - These cache blocks are called *pages* (size is P = 2^p bytes)

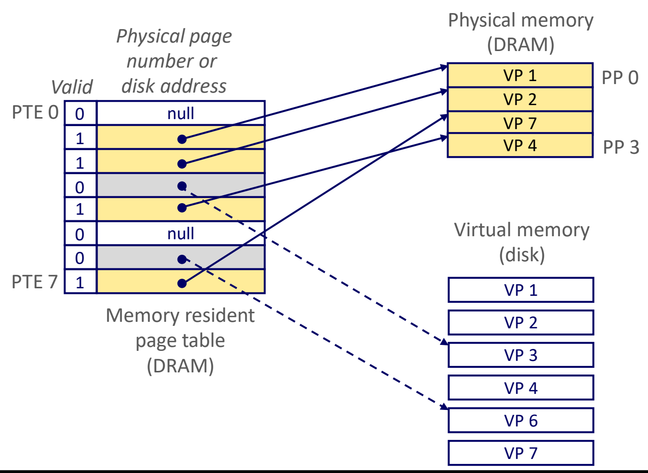


DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM
  - Time to load block from disk > 1ms (> 1 million clock cycles)
    - CPU can do a lot of computation during that time
- Consequences
  - Large page (block) size: typically 4 KB
    - Linux “huge pages” are 2 MB (default) to 1 GB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

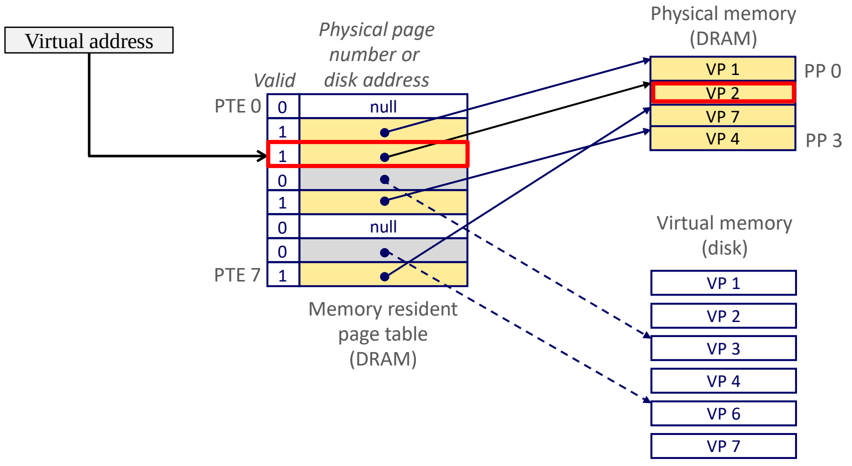
Enabling Data Structure: Page Table

- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM



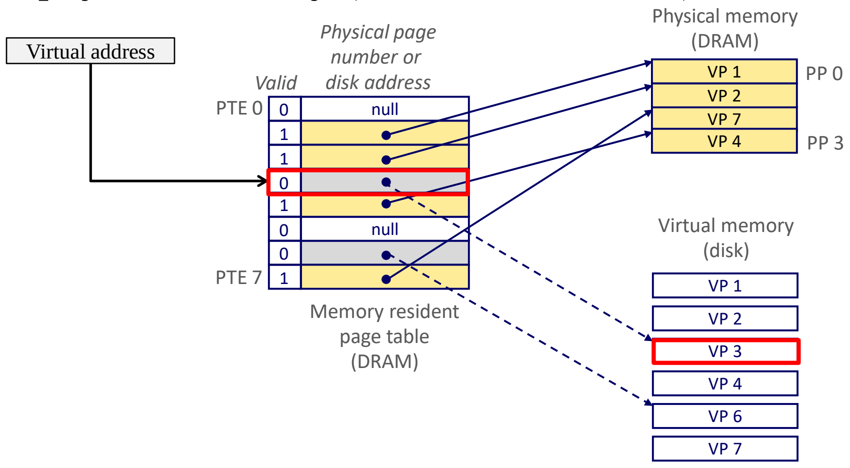
Page Hit

- *Page hit:* reference to VM word that is in physical memory (DRAM cache hit)



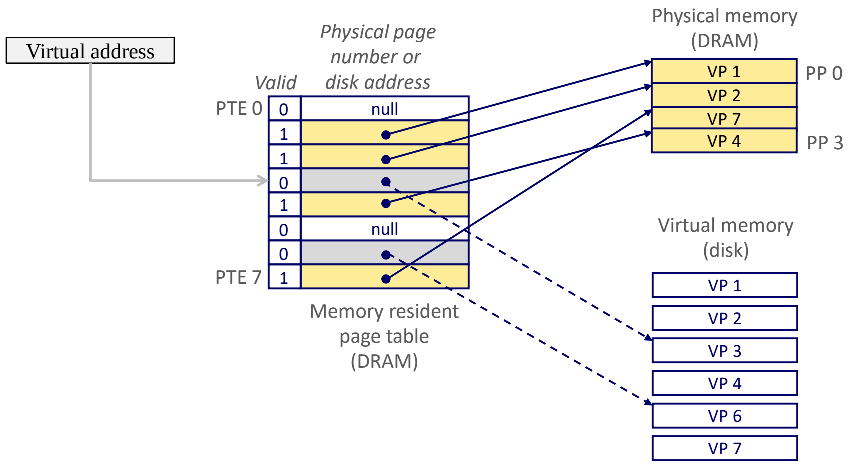
Page Fault

- *Page fault:* reference to VM word that is not in physical memory (DRAM cache miss)



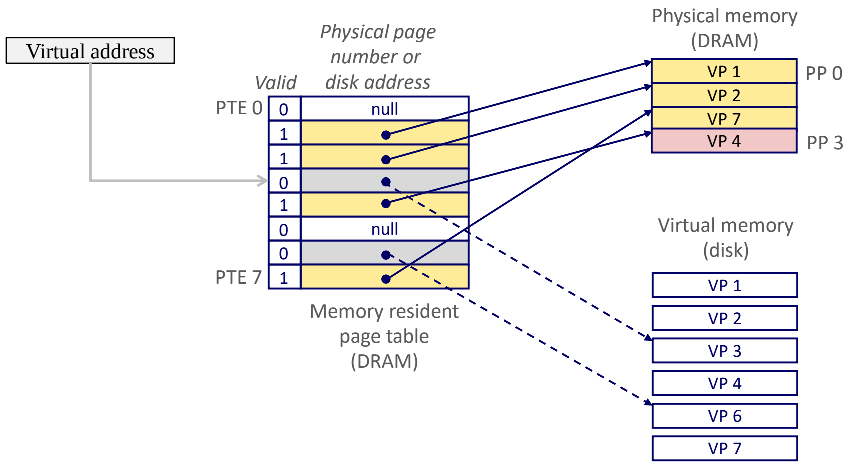
Handling Page Fault

- Page miss causes page fault (an exception)



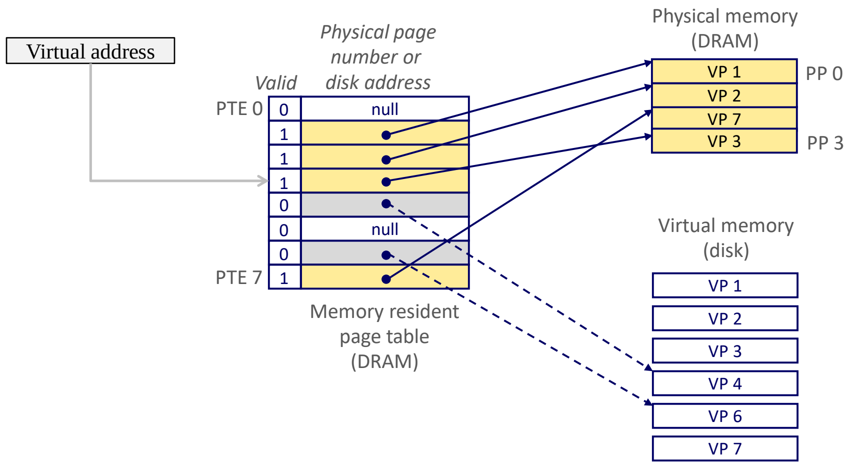
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



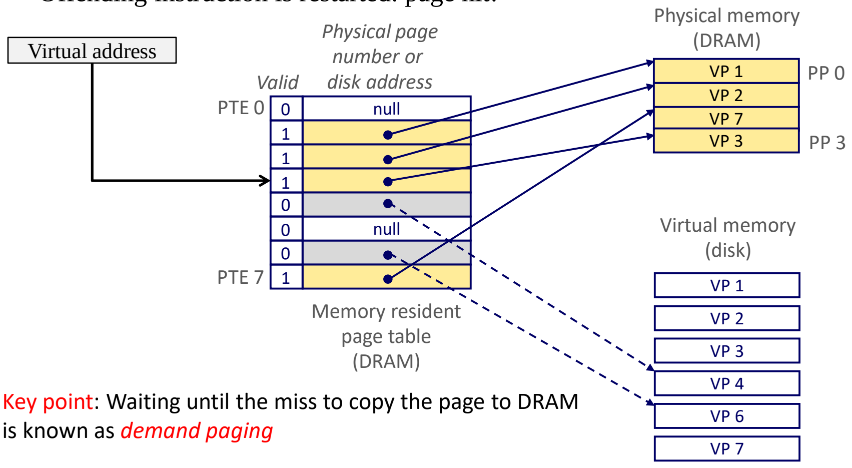
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

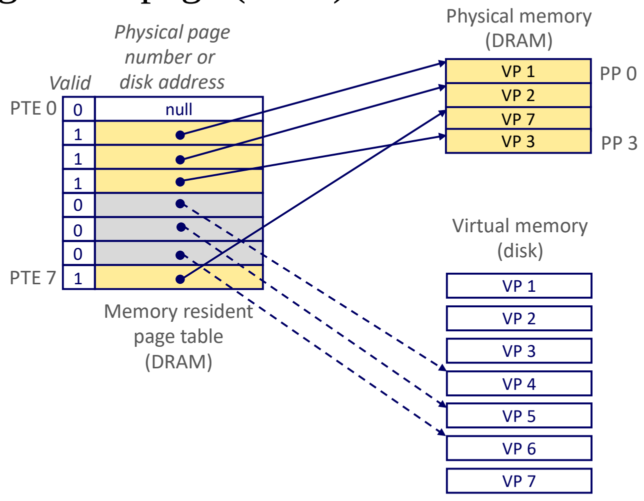
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Locality to the Rescue Again!

- Virtual memory seems terribly inefficient, but it works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - Good performance for one process (after cold misses)

Motivation for Virtual Memory

- The physical main memory (RAM) is relatively limited in capacity.
- It may not be big enough to store all the executing programs at the same time.
- A program may need memory larger than the main memory size, but the whole program doesn’t need to be kept in the main memory at the same time.
- Virtual Memory takes advantage of the fact that at any given instant of time, an executing program needs only a fraction of the memory that the whole program occupies.
- **The basic idea:** Load only pieces of each executing program which are currently needed.

Virtual Memory - Objectives

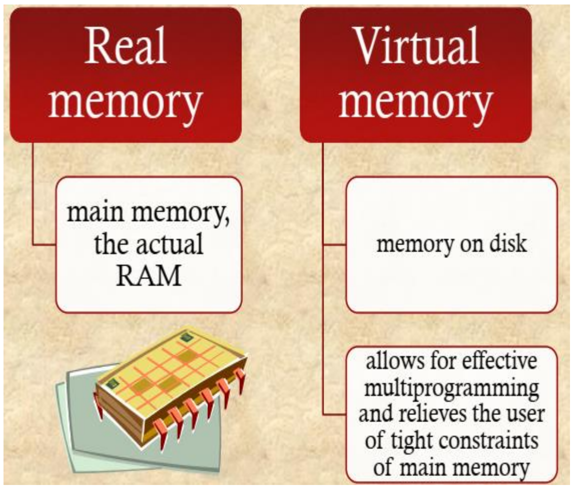
- Allow program to be written without memory constraints
  - Program can exceed the size of the main memory
- Many Programs sharing DRAM Memory so that context switches can occur
- Relocation: Parts of the program can be placed at different locations in the memory instead of a big chunk
- Virtual Memory:
  - Main Memory holds many programs running at same time (processes)
  - Use Main Memory as a kind of “cache” for disk

Characteristics of Paging and Segmentation

- Memory references are dynamically translated into physical addresses at run time
  - a process may be swapped in and out of main memory such that it occupies different regions
- A process may be broken up into pieces (pages or segments) that do not need to be located contiguously in main memory
- Hence: all pieces of a process do not need to be loaded in main memory during execution
  - computation may proceed for some time if the next instruction to be fetch (or the next data to be accessed) is in a piece located in main memory

Real and Virtual Memory

- Concept real and virtual memory


Real memory
main memory, the actual RAM
Virtual memory
memory on disk
allows for effective multiprogramming and relieves the user of tight constraints of main memory

Support Needed for Virtual Memory

- For virtual memory to be practical and effective :
  - Hardware must support paging and segmentation
  - Operating system must include software for managing the movement of pages or segments between secondary memory and main memory

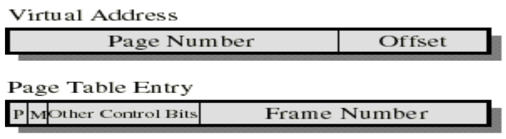
Paging of Memory

- Divide programs (processes) into equal sized, small blocks, called pages.
- Divide the primary memory into equal sized, small blocks called page frames.
- Allocate the required number of page frames to a program.
- A program does not require continuous page frames!
- The **operating system (OS)** is responsible for:
  - Maintaining a list of free frames.
  - Using a page table to keep track of the mapping between pages and page frames

Paging

- The term *virtual memory* is usually associated with systems that employ paging
- Use of paging to achieve virtual memory was first reported for the Atlas computer
- Each process has its own page table
  - Each page table entry contains the frame number of the corresponding page in main memory

- Typically, each process has its own page table



P= present bit
M = Modified bit

- Each page table entry contains a present bit to indicate whether the page is in main memory or not.
  - If it is in main memory, the entry contains the frame number of the corresponding page in main memory
  - If it is not in main memory, the entry may contain the address of that page on disk or the page number may be used to index another table (often in the PCB) to obtain the address of that page on disk

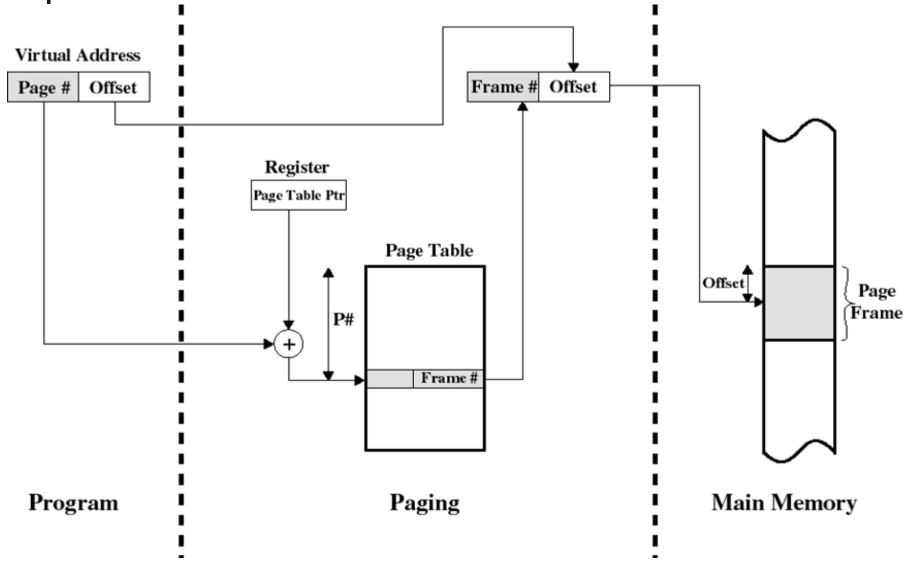
Paging

- A modified bit indicates if the page has been altered since it was last loaded into main memory
  - If no change has been made, the page does not have to be written to the disk when it needs to be swapped out
- Other control bits may be present if protection is managed at the page level
  - a read-only/read-write bit
  - protection level bit: kernel page or user page (more bits are used when the processor supports more than 2 protection levels)

Page Table Structure

- Page tables are variable in length (depends on process size)
  - then must be in main memory instead of registers
- A single register holds the starting physical address of the page table of the currently running process

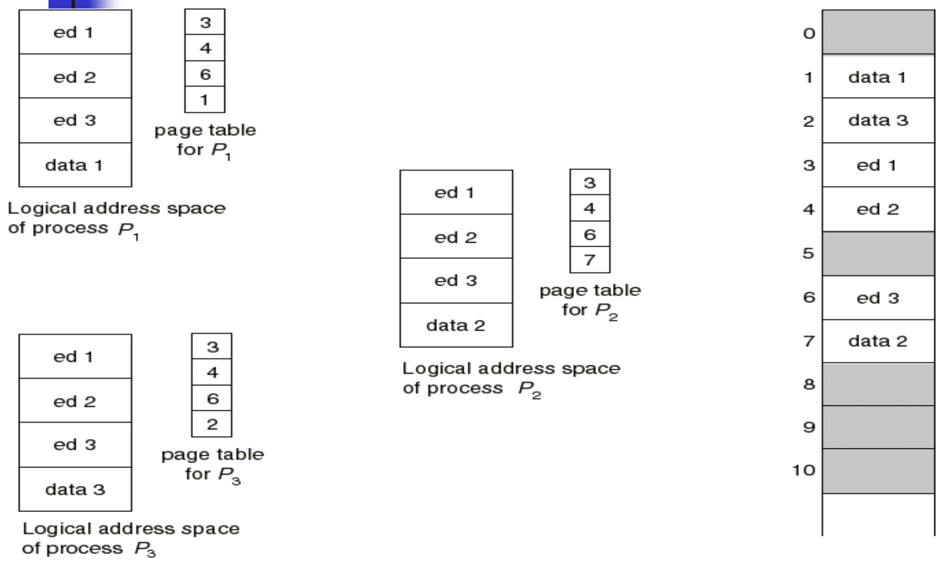
Address Translation in a Paging System



Sharing Pages

- If we share the same code among different users, it is sufficient to keep only one copy in main memory
- Shared code must be reentrant (ie: non self-modifying) so that 2 or more processes can execute the same code
- If we use paging, each sharing process will have a page table who’s entry points to the same frames: only one copy is in main memory
- But each user needs to have its own private data pages

Sharing Pages: a text editor

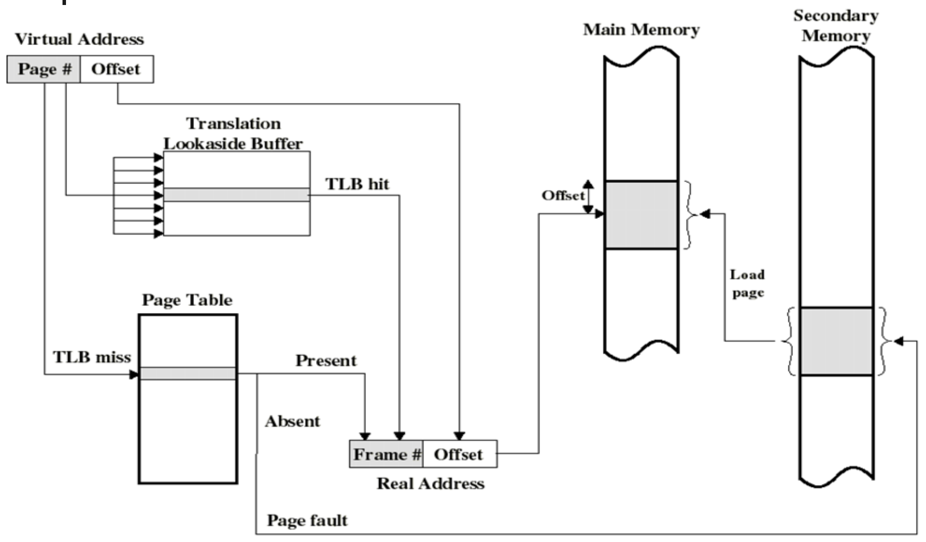


Translation Lookaside Buffer

- Because the page table is in main memory, each virtual memory reference causes at least two physical memory accesses
  - one to fetch the page table entry
  - one to fetch the data
- To overcome this problem a special cache is set up for page table entries
  - called the TLB - Translation Lookaside Buffer
    - Contains page table entries that have been most recently used
    - Works similar to main memory cache

- Given a logical address, the processor examines the TLB
- If page table entry is present (a hit), the frame number is retrieved and the real (physical) address is formed
- If page table entry is not found in the TLB (a miss), the page number is used to index the process page table
  - if present bit is set then the corresponding frame is accessed
  - if not, a page fault is issued to bring in the referenced page in main memory
- The TLB is updated to include the new page entry

Use of a Translation Lookaside Buffer



TLB: further comments

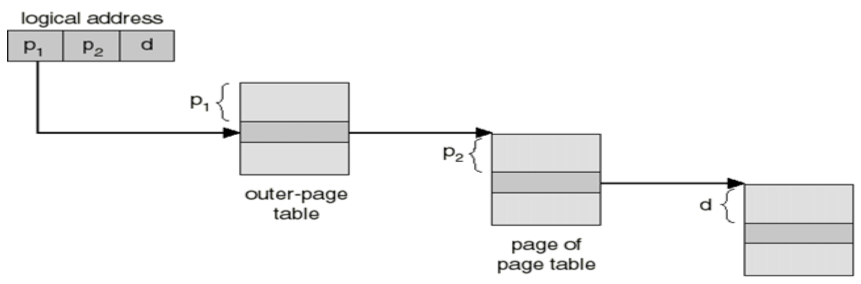
- TLB use associative mapping hardware to simultaneously interrogates all TLB entries to find a match on page number
- The TLB must be flushed each time a new process enters the Running state
- The CPU uses two levels of cache on each virtual memory reference
  - first the TLB: to convert the logical address to the physical address
  - once the physical address is formed, the CPU then looks in the cache for the referenced word

Page Tables and Virtual Memory

- Most computer systems support a very large virtual address space
  - 32 to 64 bits are used for logical addresses
  - If (only) 32 bits are used with 4KB pages, a page table may have 2^{20} entries
- The entire page table may take up too much main memory. Hence, page tables are often also stored in virtual memory and subjected to paging
  - When a process is running, part of its page table must be in main memory (including the page table entry of the currently executing page)

Multilevel Page Tables

- Since a page table will generally require several pages to be stored. One solution is to organize page tables into a multilevel hierarchy
  - When 2 levels are used (ex: 386, Pentium), the page number is split into two numbers p1 and p2
  - p1 indexes the outer paged table (directory) in main memory who’s entries points to a page containing page table entries which is itself indexed by p2. Page tables, other than the directory, are swapped in and out as needed

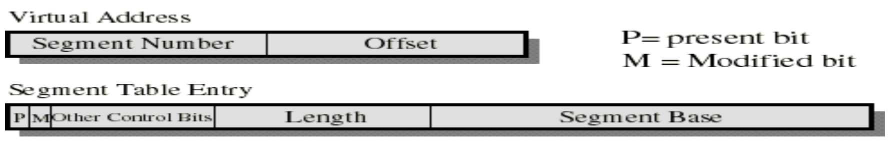


Segmentation

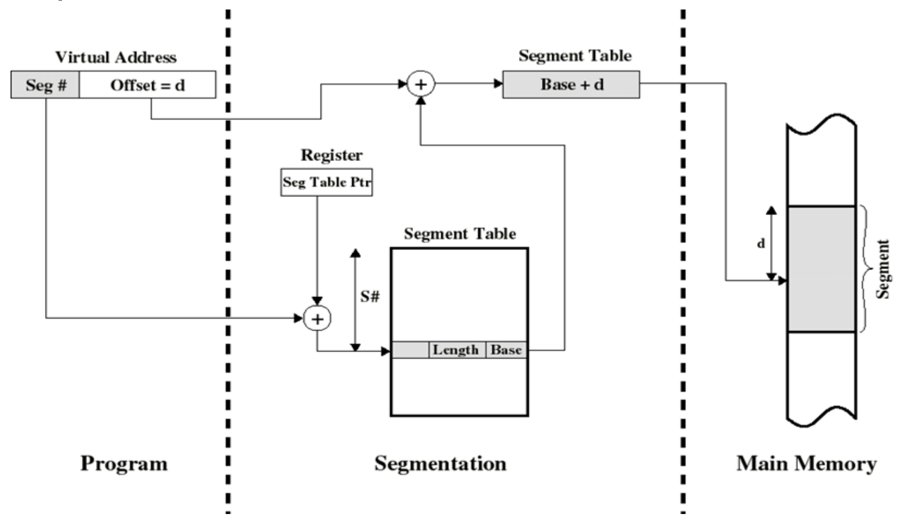
- Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments
- Advantages:
  - simplifies handling of growing data structures
  - allows programs to be altered and recompiled Independently
  - lends itself to sharing data among processes
  - lends itself to protection

Segmentation

- Typically, each process has its own segment table
- Similarly to paging, each segment table entry contains a present bit and a modified bit
- If the segment is in main memory, the entry contains the starting address and the length of that segment
- Other control bits may be present if protection and sharing is managed at the segment level
- Logical to physical address translation is similar to paging except that the offset is added to the starting address (instead of being appended)



Address Translation in a Segmentation System



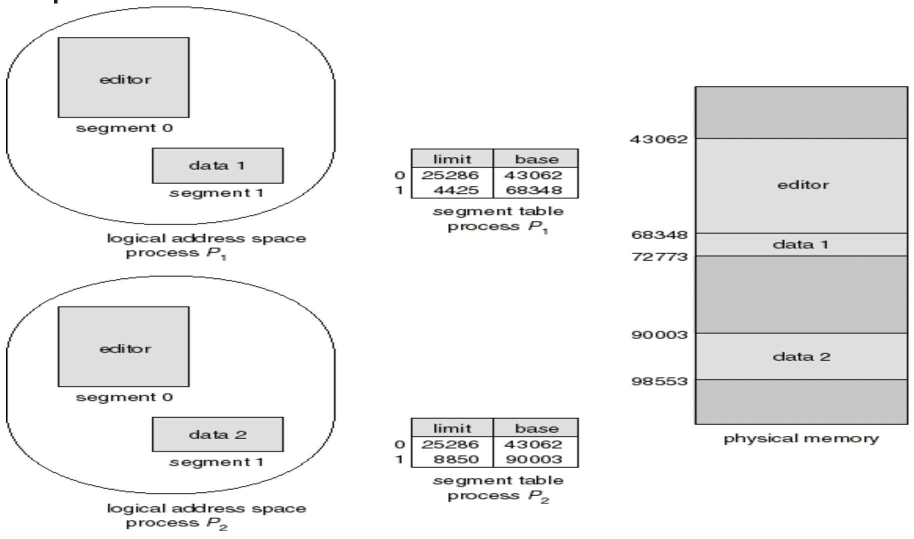
Segmentation: comments

- In each segment table entry we have both the starting address and length of the segment
  - the segment can thus dynamically grow or shrink as needed
  - address validity easily checked with the length field
- But variable length segments introduce external fragmentation and are more difficult to swap in and out...
- It is natural to provide protection and sharing at the segment level since segments are visible to the programmer (pages are not)
- Useful protection bits in segment table entry:
  - read-only/read-write bit
  - Supervisor/User bit

Sharing in Segmentation Systems

- Segments are shared when entries in the segment tables of 2 different processes point to the same physical locations
- Ex: the same code of a text editor can be shared by many users
  - Only one copy is kept in main memory
- but each user would still need to have its own private data segment

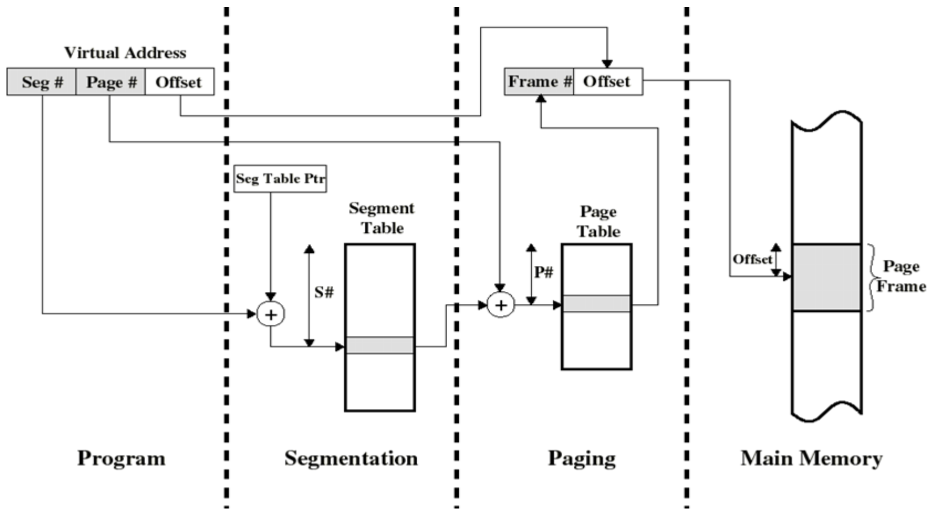
Sharing of Segments: text editor example



Combined Segmentation and Paging

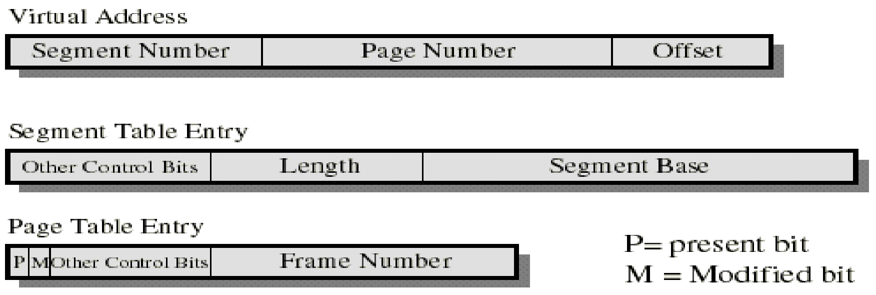
- To combine their advantages some processors and OS page the segments.
- Several combinations exists. Here is a simple one
- Each process has:
  - one segment table
  - several page tables: one page table per segment
- The virtual address consist of:
  - a segment number: used to index the segment table who's entry gives the starting address of the page table for that segment
  - a page number: used to index that page table to obtain the corresponding frame number
  - an offset: used to locate the word within the frame

Address Translation in a (simple) combined Segmentation/Paging System



Simple Combined Segmentation and Paging

- The Segment Base is the physical address of the page table of that segment
- Present and modified bits are present only in page table entry
- Protection and sharing info most naturally resides in segment table entry
  - Ex: a read-only/read-write bit, a kernel/user bit...



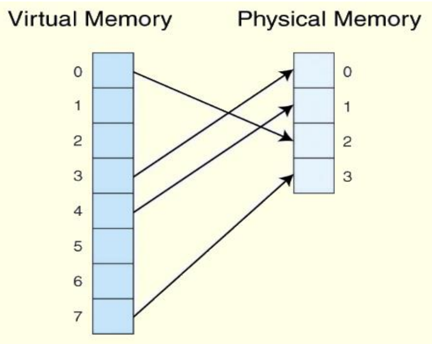
Introduction

- Cache memory enhances performance by providing faster memory access speed.
- Virtual memory enhances performance by providing greater memory capacity, without the expense of adding main memory.
- Instead, a portion of a disk drive serves as an extension of main memory.
- If a system uses paging, virtual memory partitions main memory into individually managed *page frames*, that are written *(or paged)* to disk when they are not immediately needed.

- A *physical address* is the actual memory address of physical memory.
- Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager.
- *Page faults* occur when a logical address requires that a page be brought in from disk.
- *Memory fragmentation* occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

- Main memory and virtual memory are divided into equal sized pages.
- The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- Further, the pages allocated to a process do not need to be stored contiguously -- either on disk or in memory.
- In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

- Information concerning the location of each page, whether on disk or in memory, is maintained in a data structure called a *page table* (shown below).
- There is one page table for each active process.



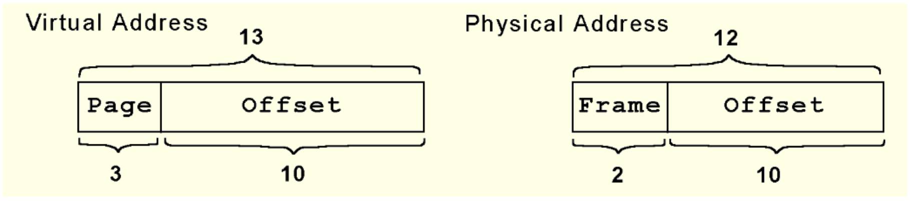
Page Table

Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

- When a process generates a virtual address, the operating system translates it into a physical memory address.
- To accomplish this, the virtual address is divided into two fields: A *page* field, and an *offset* field.
- The page field determines the page location of the address, and the offset indicates the location of the address within the page.
- The logical page number is translated into a physical page frame through a lookup in the page table.

- If the valid bit is zero in the page table entry for the logical address, this means that the page is not in memory and must be fetched from disk.
  - This is a page fault.
  - If necessary, a page is evicted from memory and is replaced by the page retrieved from disk, and the valid bit is set to 1.
- If the valid bit is 1, the virtual page number is replaced by the physical frame number.
- The data is then accessed by adding the offset to the physical frame number.

- As an example, suppose a system has a virtual address space of 8K, each page has 1K, and a physical address space of 4K bytes. The system uses byte addressing.
  - We have $2^{13}/2^{10} = 2^3$ virtual pages.
- A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first two bits for the page frame and the trailing 10 bits the offset.



Virtual memory

- Suppose we have the page table shown below.
- What happens when CPU generates address $5459_{10} = 1010101010011_2$? (in page 5, the first 3 bits is 101)

Page Table

Page	Frame	Valid Bit
Page 0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Page	Addresses
Page 0 :	0 – 1023
1 :	1024 – 2047
2 :	2048 – 3071
3 :	3072 – 4095
4 :	4096 – 5119
5 :	5120 – 6143
6 :	6144 – 7167
7 :	7168 – 8191

- The address 1010101010011_2 is converted to physical address 010101010011 because the page field 101 is replaced by frame number 01 (in frame #1) through a lookup in the page table.

Page Table

Page	Frame	Valid Bit
Page 0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Page	Addresses
Page 0 :	0 – 1023
1 :	1024 – 2047
2 :	2048 – 3071
3 :	3072 – 4095
4 :	4096 – 5119
5 :	5120 – 6143
6 :	6144 – 7167
7 :	7168 – 8191

- What happens when the CPU generates address 1000000000100_2? (first 3 bits is 100, in page #4)

Page Table

Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

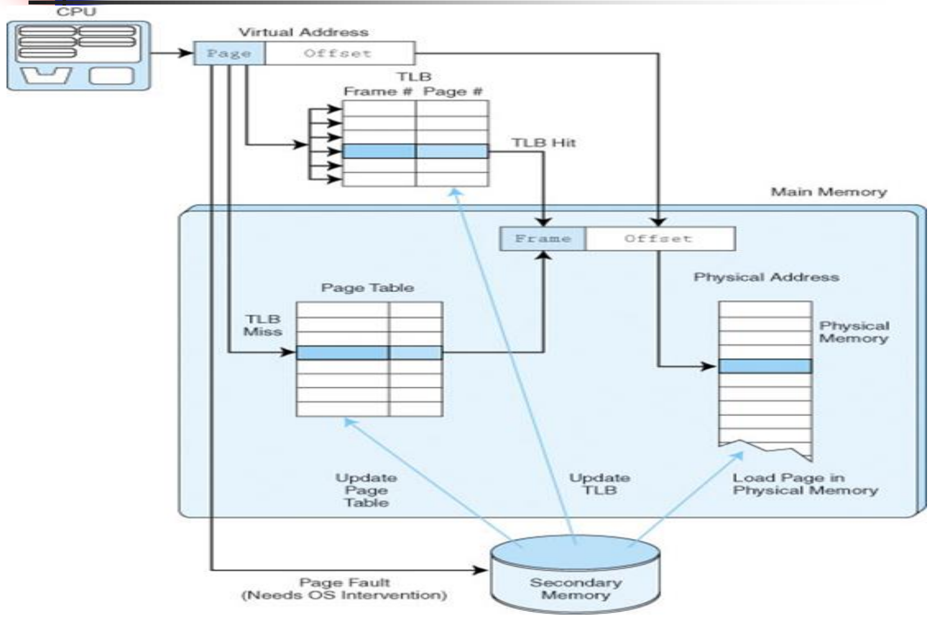
Page	Addresses
0 :	0 - 1023
1 :	1024 - 2047
2 :	2048 - 3071
3 :	3072 - 4095
4 :	4096 - 5119
5 :	5120 - 6143
6 :	6144 - 7167
7 :	7168 - 8191

- We said earlier that effective access time (EAT) takes all levels of memory into consideration.
- Thus, virtual memory is also a factor in the calculation, and we also have to consider page table access time.
- Suppose a main memory access takes 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk. We have:

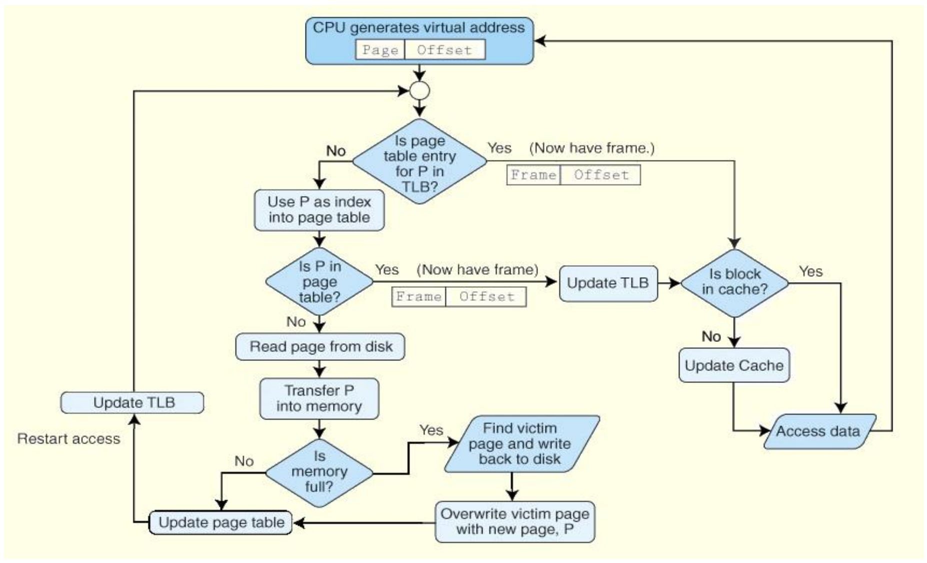
EAT = 0.99(200ns + 200ns) + 0.01(10ms) = 100,396ns.

- Even if we had no page faults, the EAT would be 400ns because memory is always read twice: First to access the page table, and second to load the page from memory.
- Because page tables are read constantly, it makes sense to keep most recent page lookup values in a special cache called a *translation look-aside buffer* (TLB).
- TLBs are a special associative cache that stores the mapping of virtual pages to physical pages.

Example of virtual address



Flowchart of virtual address



- Another approach to virtual memory is the use of *segmentation*.
- Instead of dividing memory into equal-sized pages, virtual address space is divided into variable-length segments, often under the control of the programmer.
- A segment is located through its entry in a segment table, which contains the segment's memory location and a bounds limit that indicates its size.
- After a page fault, the operating system searches for a location in memory large enough to hold the segment that is retrieved from disk.

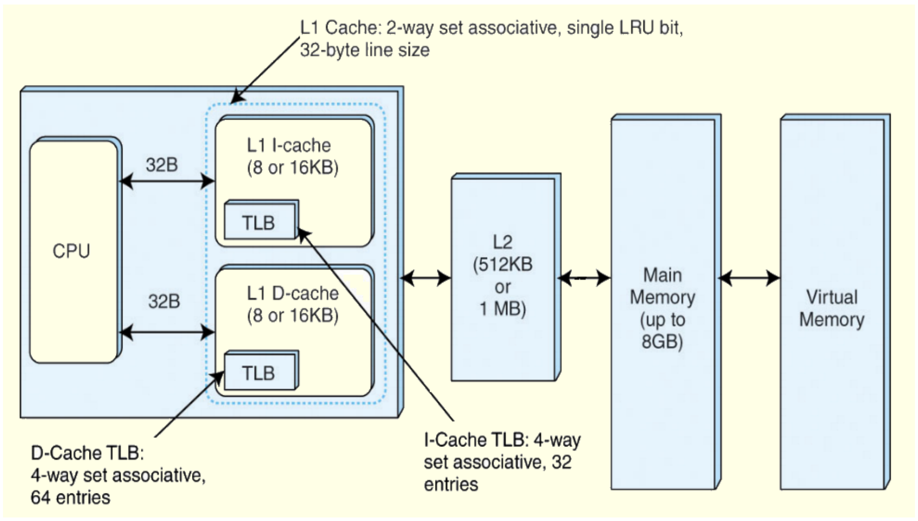
- Both paging and segmentation can cause fragmentation.
- Paging is subject to *internal* fragmentation because a process may not need the entire range of addresses contained within the page. Thus, there may be many pages containing unused fragments of memory.
- Segmentation is subject to *external* fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over time.

- Large page tables are cumbersome and slow, but with its uniform memory mapping, page operations are fast. Segmentation allows fast access to the segment table, but segment loading is labor-intensive.
- Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.
- Each segment has a page table. This means that a memory address will have three fields, one for the segment, another for the page, and a third for the offset.

Real-World Example

- The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpaged unsegmented, segmented unpaged, unsegmented paged, and segmented paged memory (pp263-264).
- The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes.
- The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- The L1 cache is in two parts: and instruction cache (I-cache) and a data cache (D-cache).

Real world example


L1 Cache: 2-way set associative, single LRU bit, 32-byte line size
CPU
32B
L1 I-cache (8 or 16KB)
TLB
32B
L1 D-cache (8 or 16KB)
TLB
L2 (512KB or 1 MB)
Main Memory (up to 8GB)
Virtual Memory
D-Cache TLB: 4-way set associative, 64 entries
I-Cache TLB: 4-way set associative, 32 entries