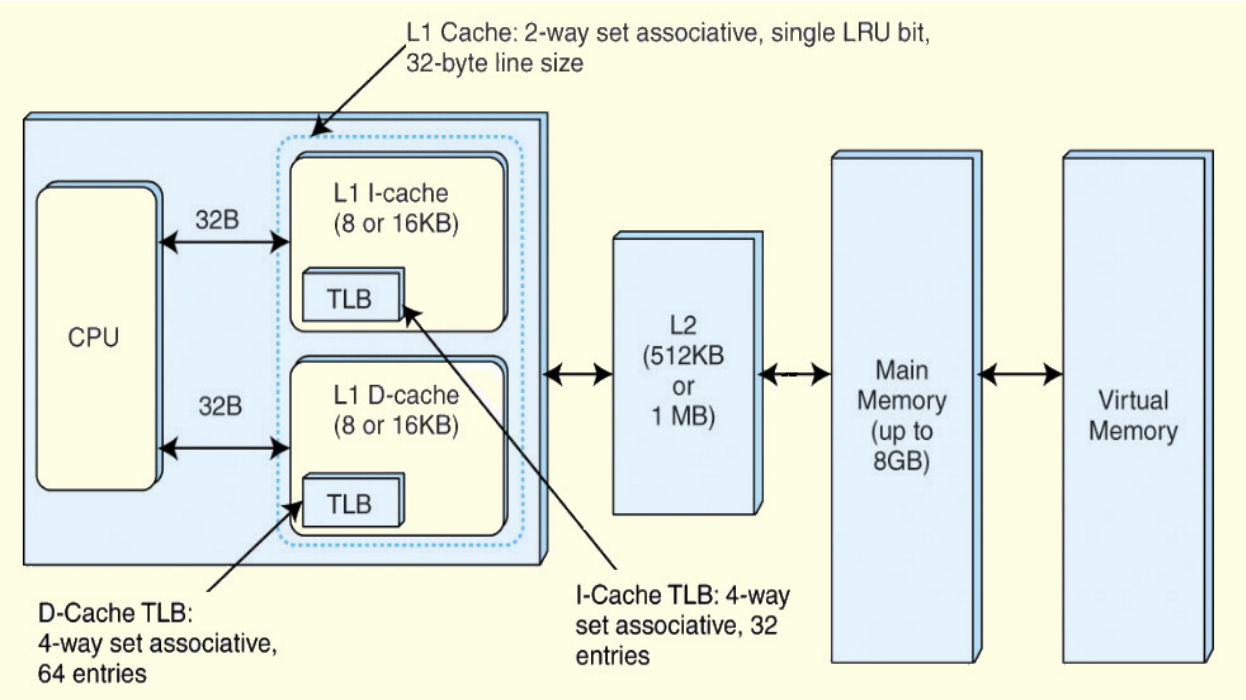
Typical system view of the memory hierarchy

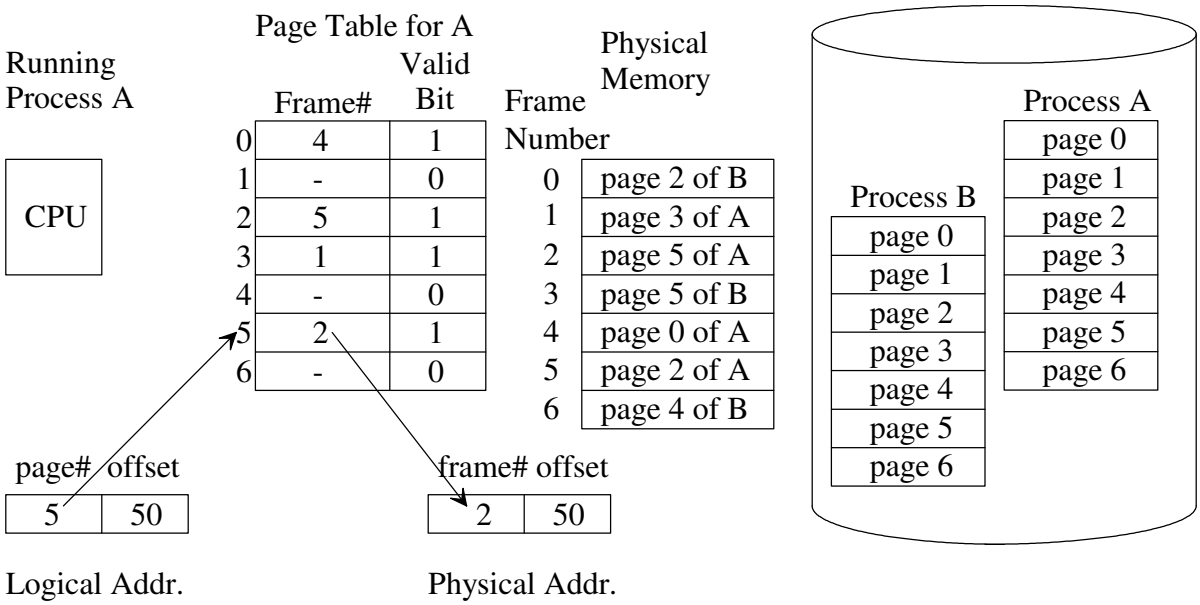


Virtual Memory - programmer views memory as large address space without concerns about the amount of physical memory or memory management. (What do the terms **32-bit** (or **64-bit) operating system** mean?)

Benefits:
1) programs can be bigger that physical memory size since only a portion of them may actually be in physical memory
2) higher degree of multiprogramming is possible since only portions of programs are in memory

An Operating System **goal** with hardware support is to make virtual memory efficient and transparent to the user.

Memory-Management Unit (MMU) for paging



Note: The "Valid" bit is indicates whether the page is loaded into memory

Demand paging is a common way for OSs to implement virtual memory. Demand paging ("lazy pager") only brings a page into physical memory when it is needed. A "Valid bit" is used in a page table entry to indicate if the page is in memory or only on disk.

A ***page fault*** occurs when the CPU generates a logical address for a page that is not in physical memory. The MMU will cause a ***page-fault trap*** (interrupt) to the OS.

Steps for OS's page-fault trap handler:
1) Check page table to see if the page is valid (exists in logical address space). If it is invalid, terminate the process; otherwise continue.

2) Find a free frame in physical memory (take one from the free-frame list or replace a page currently in memory).

3) Schedule a disk read operation to bring the page into the free page frame. (We might first need to schedule a previous disk write operation to update the virtual memory copy of a "dirty" page that we are replacing.)

4) Since the disk operations are soooooo slooooooow, the OS would context switch to another ready process selected from the ready queue.

5) After the disk (a DMA device) reads the page into memory, it involves an I/O completion interrupt. The OS will then update the PCB and page table for the process to indicate that the page in now in memory and the process is ready to run.

6) When the process is selected by the short-term scheduler to run, it repeats the instruction that caused the page fault. The memory reference that caused the page fault will now succeed.

Performance of Demand Paging
To achieve acceptable performance degradation (5-10%) of our virtual memory, we must have a very low page fault rate (probability that a page fault will occur on a memory reference).

When does a CPU perform a memory reference?
1) Fetch instructions into CPU to be executed
2) Fetch operands used in an instruction (load and store instructions on RISC machines)

Example:
Let p be the page fault rate, and ma be the memory-access time.
Assume that p = 0.02, ma = 50 ns and the time to perform a page fault is 12,200,000 ns (12.2 ms).

$$\begin{pmatrix}\text{effective memory}\\ \text{access time}\end{pmatrix} = \begin{pmatrix}\text{prob. of}\\ \text{no page fault}\end{pmatrix} * \begin{pmatrix}\text{main memory}\\ \text{access time}\end{pmatrix} + \begin{pmatrix}\text{prob. of}\\ \text{page fault}\end{pmatrix} * \begin{pmatrix}\text{page fault}\\ \text{time}\end{pmatrix}$$

$$= (1 - p) * 50\text{ns} + p * 12{,}200{,}000$$
$$= 0.98 * 50\text{ns} + 0.02 * 12{,}200{,}000$$
$$= 244{,}049\text{ns}$$

The program would appear to run very slowly!!!

If we only want say 10% slow down of our memory, then the page fault rate must be much better!

$55 = (1 - p) * 50\text{ns} + p * 12{,}200{,}000\text{ns}$

$55 = 50 - 50p + 12{,}200{,}000p$

$p = 0.0000004$ or 1 page fault in 2,439,990 references

Fortunately, programs exhibit ***locality of reference*** that helps achieve low page-fault rates. Page size is typically 4 KB.

Storage of the Page Table Issues

1) Where is it located?

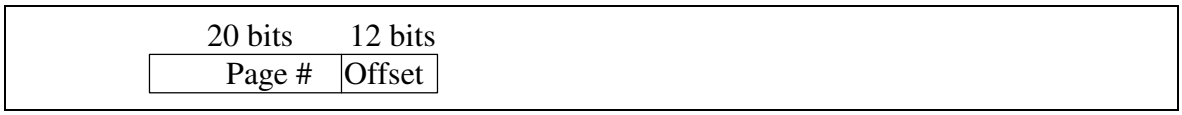
If it is in memory, then each memory reference in the program, results in two memory accesses; one for the page table entry, and another to perform the desired memory access.

Solution: TLB (Translation-lookaside Buffer) - small, fully-associative cache to hold PT entries
Ideally, when the CPU generates a memory reference, the PT entry is found in the TLB, the page is in memory, and the block with the page is in the cache, so NO memory accesses are needed.
However, each CPU memory reference involves two cache lookups **and** these cache lookups must be done sequentially, i.e., first check TLB to get physical frame # used to build the physical address, then use the physical address to check the tag of the L1 cache.

Alternatively, the L1 cache can contain virtual addresses (called a *virtual cache*). This allows the TLB and cache access to be done in parallel. If the cache hits, the result of the TLB is not used. If the cache misses, then the address translation is under way and used by the L2 cache.

2) Ways to handle large page tables:

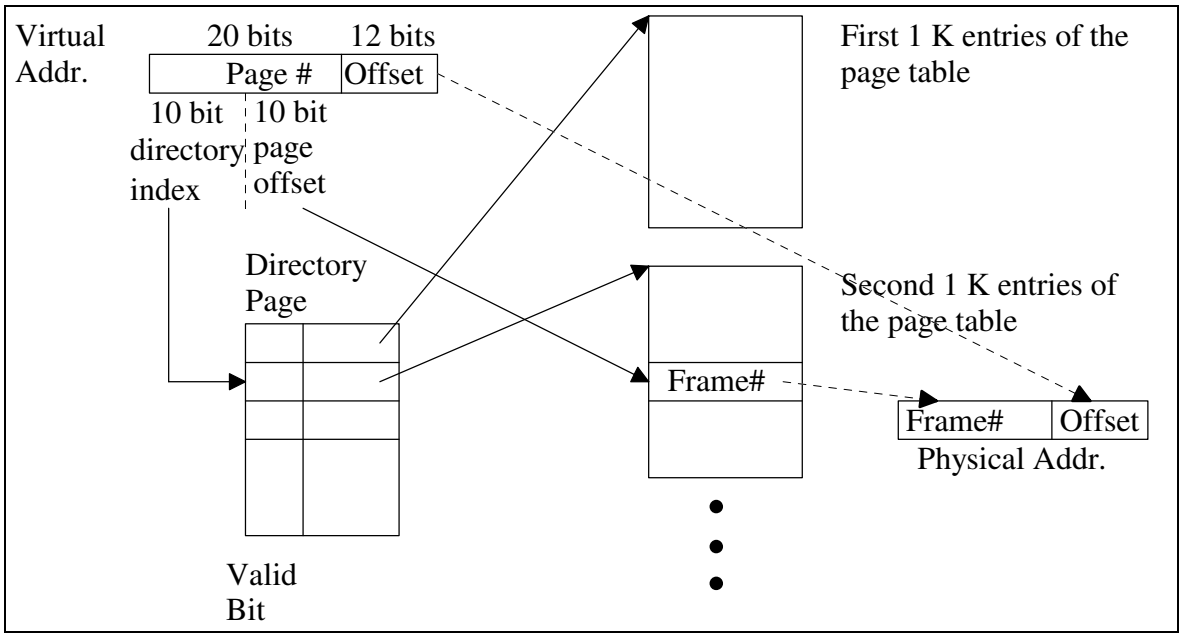
Page table for each process can be large
e.g., 32-bit address, 4 KB (2^{12} bytes) pages, byte-addressable memory, 4 byte PT entry



1 M (2^{20}) of page table entries, or 4MB for the whole page table with 4 byte page table entries

A solution:

a) two-level page table - the first level (the "directory") acts as an index into the page table which is scattered across several pages. Consider a 32-bit example with 4KB pages and 4 byte page table entries.

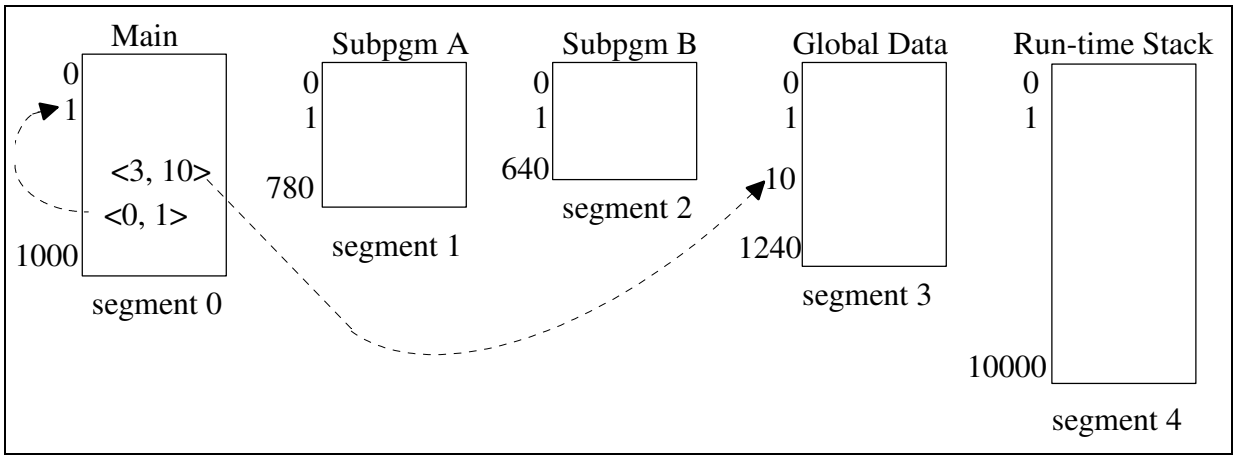


Problem with paging:

1) Protection unit is a page, i.e., each Page Table Entry can contain protection information, but the virtual address space is divided into pages along arbitrary boundaries.

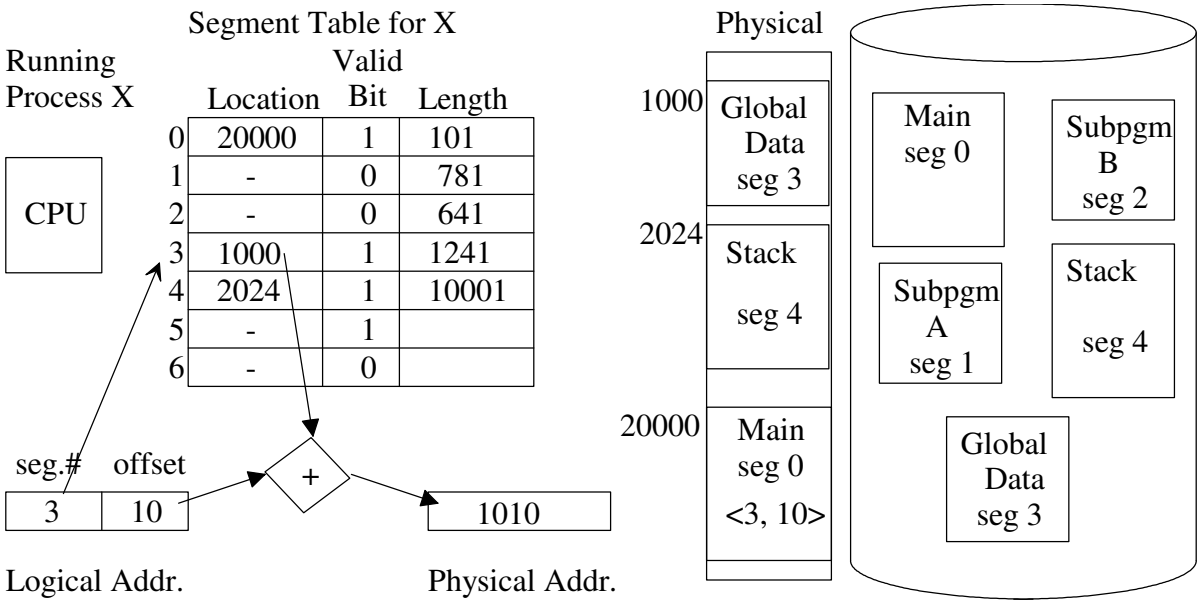
Segmentation - divides virtual address space in terms of meaningful program modules which allows each to be associated with different protection. For example, a segment containing a matrix multiplication subprogram could be shared by several programs.

Programmer views memory as multiple address spaces, i.e., segments. Memory references consist of two parts: < segment #, offset within segment >.



As in paging, the operating system with hardware support can move segments into and out of memory as needed by the program.

Each process (running program) has its own segment table similar to a page table for performing address translations.



Problems with Segmentation:

1) hard to manage memory efficiently due to external fragmentation
2) segments can be large in size so not many can be loaded into memory at one time

Solution: Combination of paging with segmentation by paging each segment.

Each segment has its own page table which is referenced from the segment table. Assume a page size of 1024 bytes