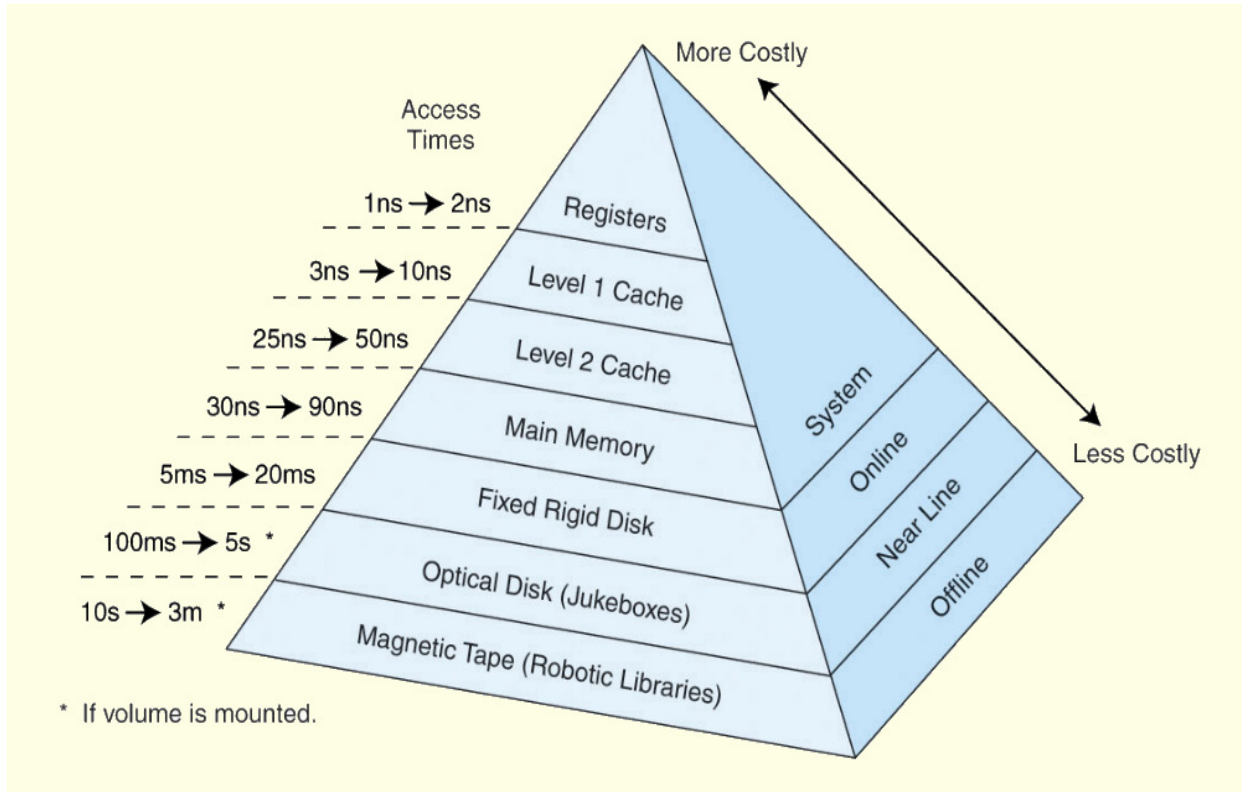
Memory Hierarchy

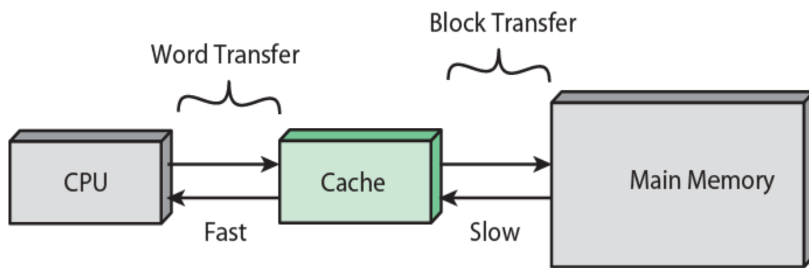
Goal: "Fast", "unlimited" storage at a reasonable cost per bit.



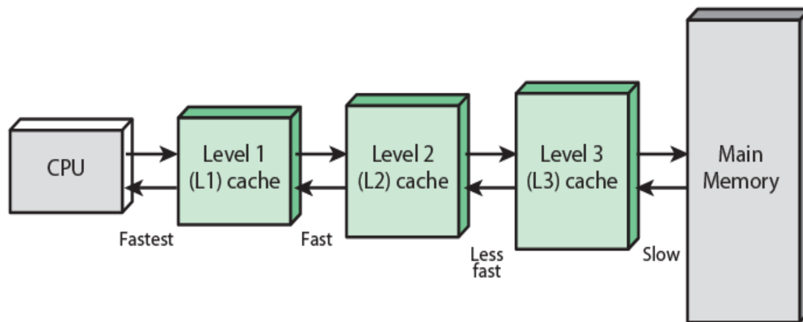
Recall the von Neumann bottleneck - single, relatively slow path between the CPU and main memory.

Typical system view of the memory hierarchy

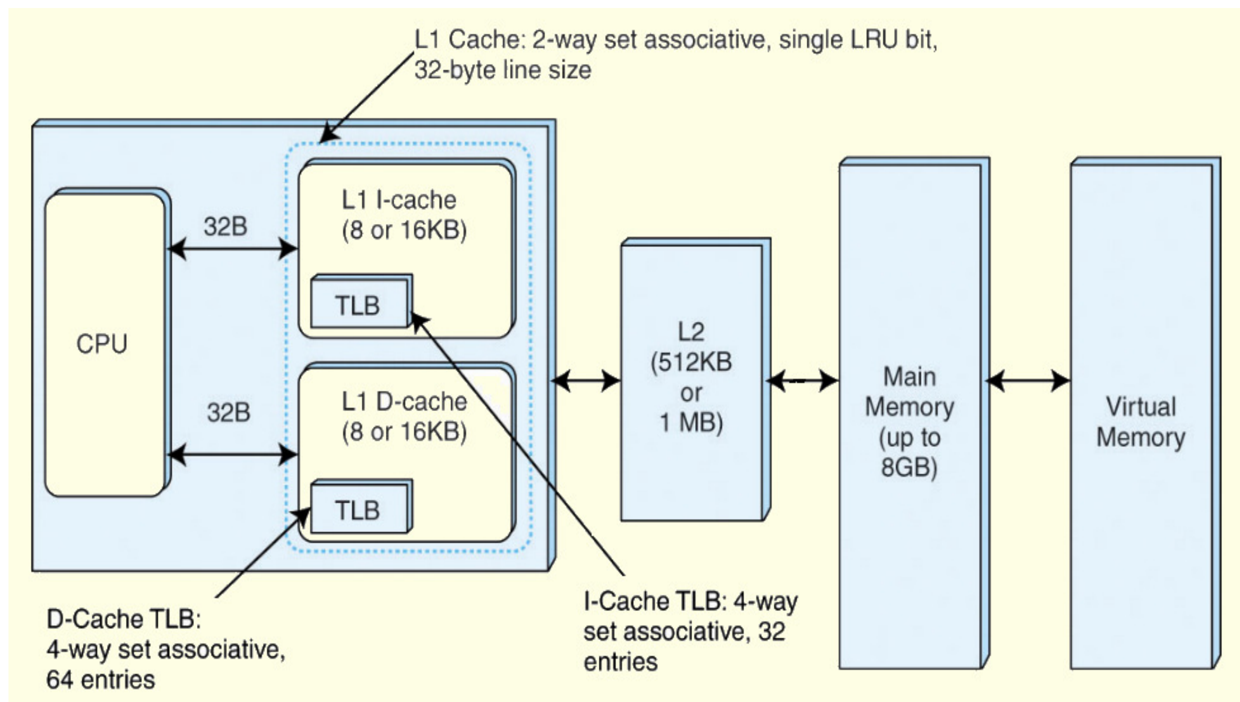
Figure 4.3 Cache and Main Memory



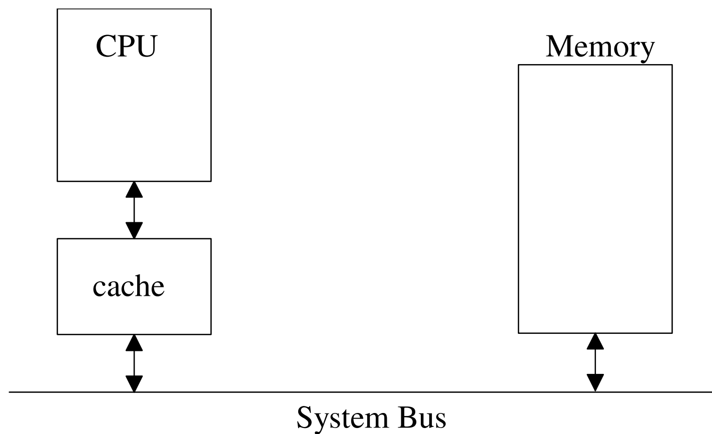
(a) Single cache



(b) Three-level cache organization



Main Idea of a Cache - keep a copy of frequently used information as "close" (w.r.t access time) to the processor as possible.



Steps when the CPU generates a memory request:
1) check the (faster) cache first
2) If the addressed memory value is in the cache (called a ***hit***), then no need to access memory
3) If the addressed memory value is NOT in the cache (called a ***miss***), then transfer the block of memory containing the reference to cache. (The CPU is stalled waiting while this occurs)
4) The cache supplies the memory value from the cache.

Effective Memory Access Time
Suppose that the hit time is 5 ns, the cache miss penalty is 160 ns, and the hit rate is 99%.

Effective Access Time ≈ (hit time * hit probability) + (miss penalty * miss probability)

Effective Access Time = 5 * 0.99 + 160 * (1 - 0.99) = 4.95 + 1.6 = 6.55 ns

(One way to reduce the miss penalty is to not have the cache wait for the whole block to be read from memory before supplying the accessed memory word.)

Fortunately, programs exhibit ***locality of reference*** that helps achieve high hit-rates:

1) ***spatial locality*** - if a (logical) memory address is referenced, nearby memory addresses will tend to be referenced soon.

2) ***temporal locality*** - if a memory address is referenced, it will tend to be referenced again soon.

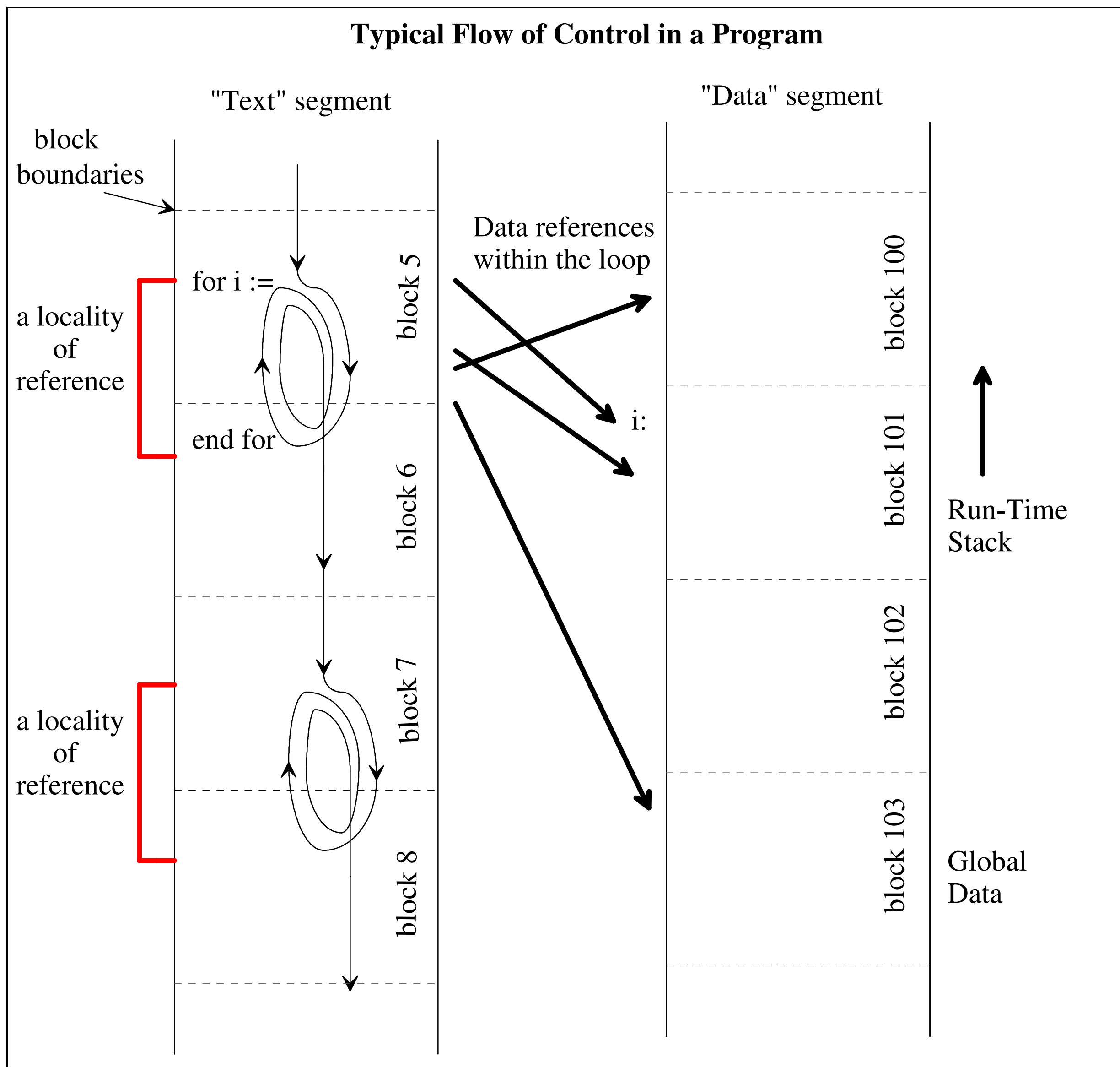
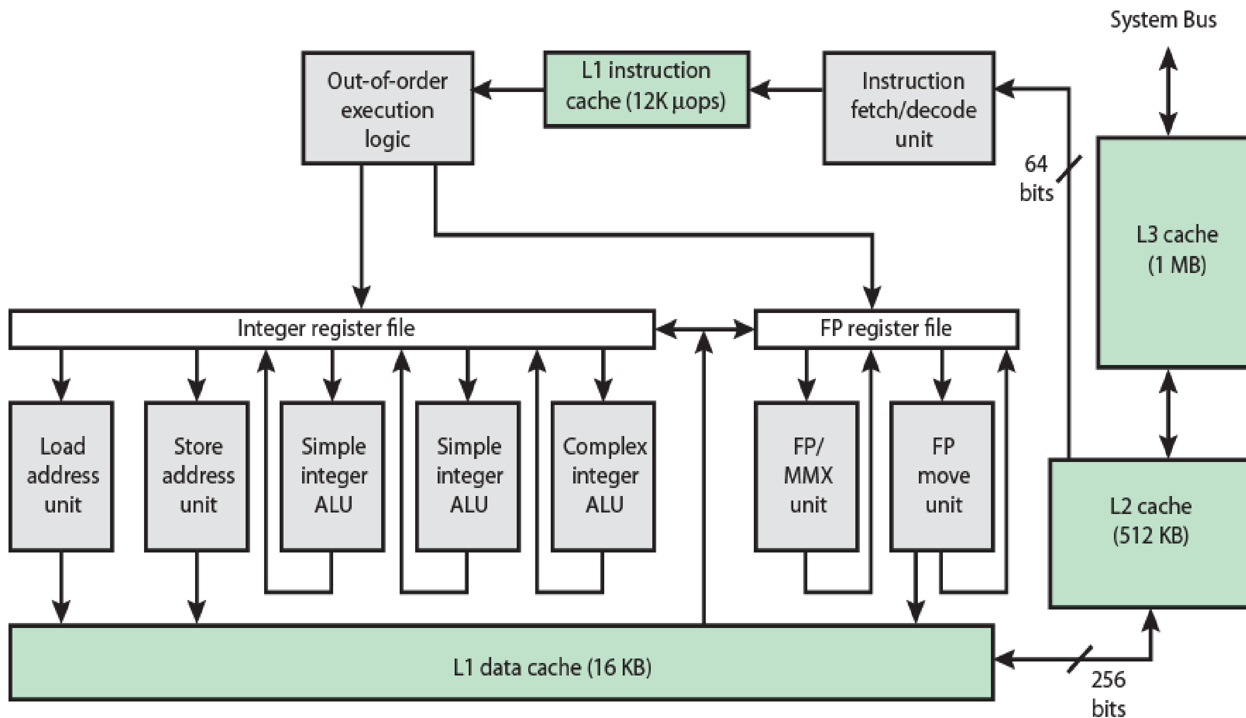
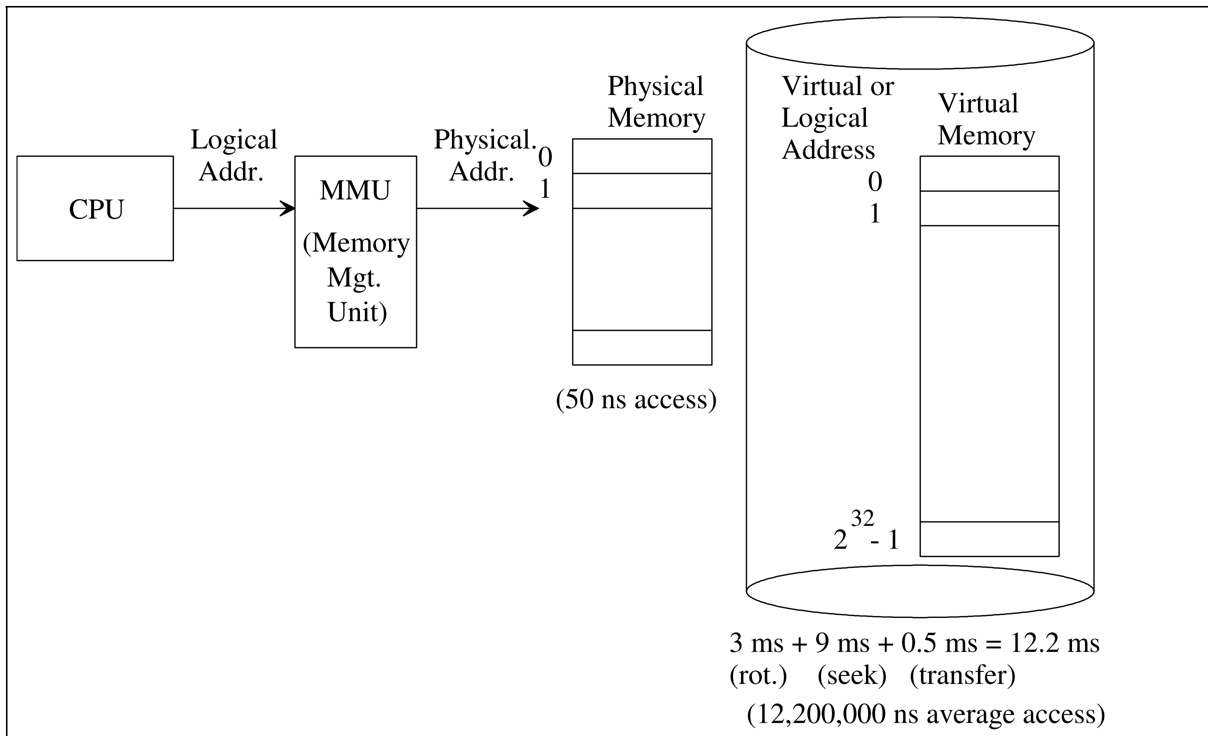


Figure 4.18 Pentium 4 Block Diagram



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** or **overlays** mean?)

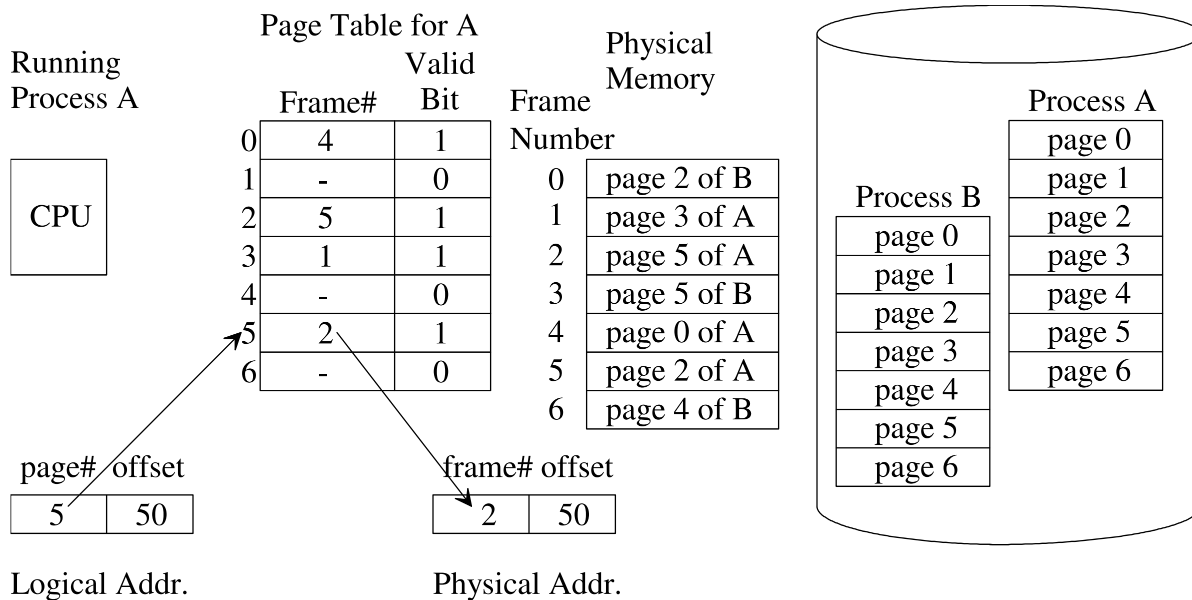


Benefits:

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

Operating System's **goals** with hardware support are to make virtual memory efficient and transparent to the user.

Memory-Management Unit (MMU) for paging



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!

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

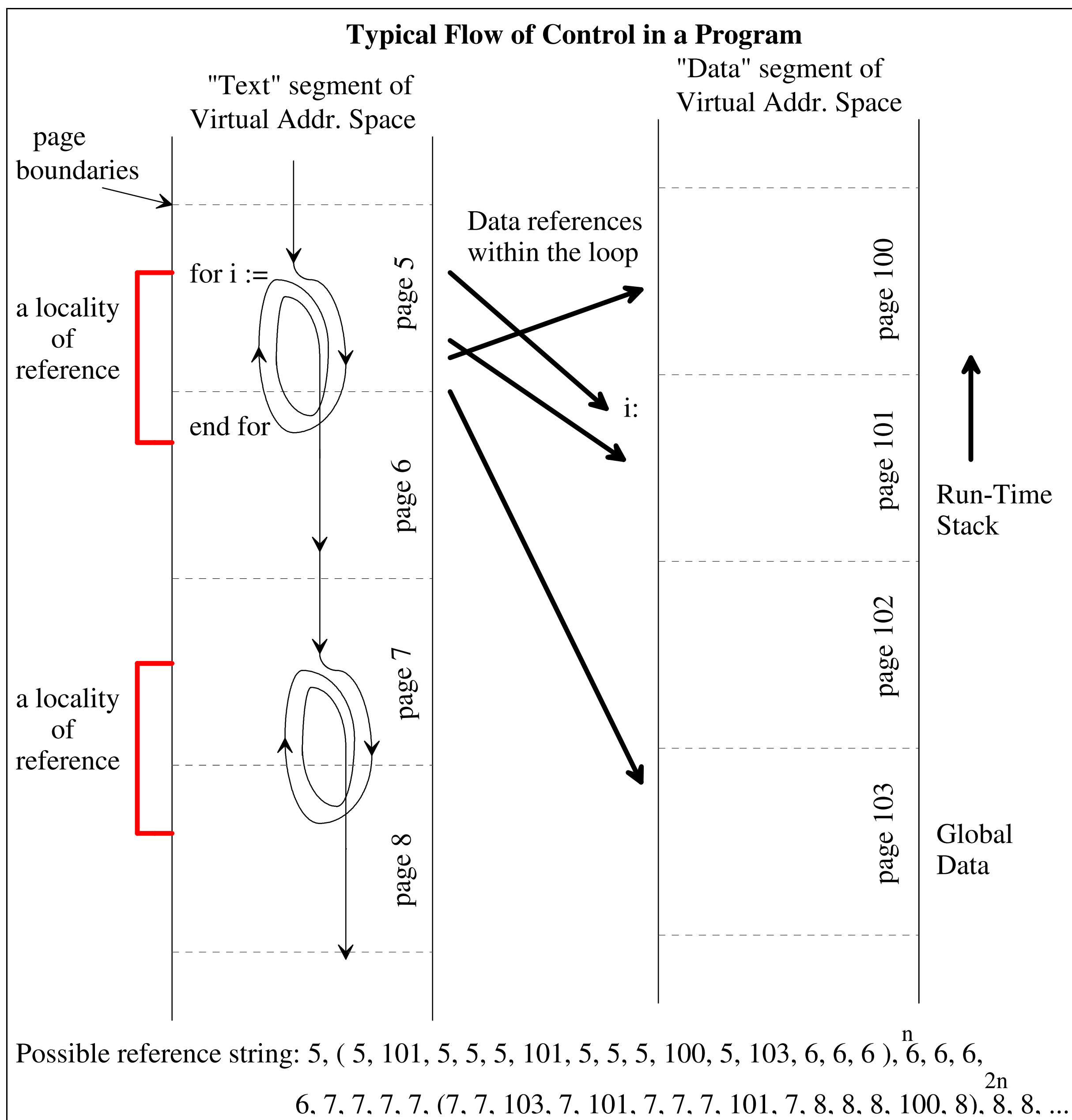
$55 = 50 - 50p + 12{,}200{,}000p$

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

Fortunately, programs exhibit ***locality of reference*** that helps achieve low page-fault rates

1) ***spatial locality*** - if a (logical) memory address is referenced, nearby memory addresses will tend to be referenced soon.

2) ***temporal locality*** - if a memory address is referenced, it will tend to be referenced again soon.



Terms:

reference string - the sequence of page #'s that a process references

locality - the set of pages that are actively used together

working set - the set of all pages accessed in the current locality of reference