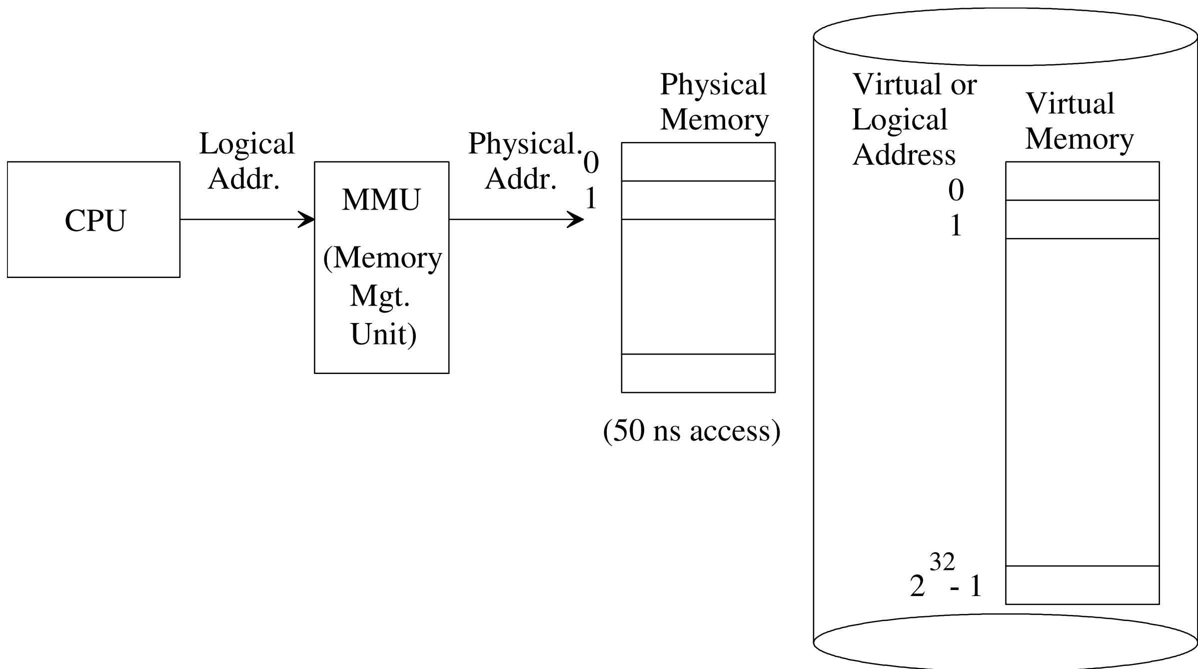
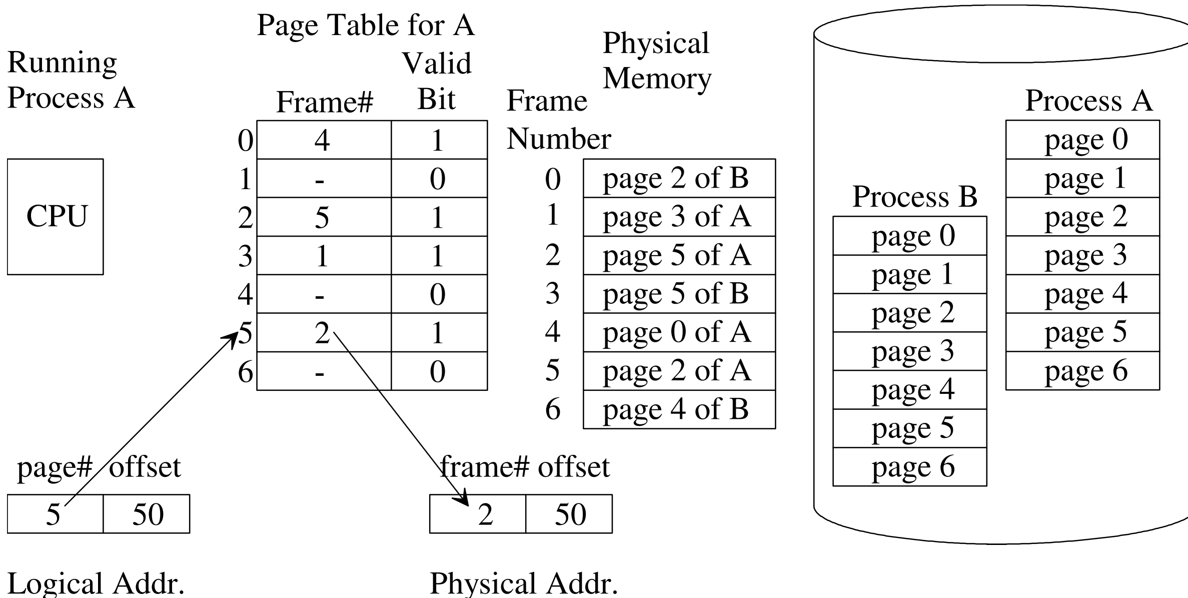
Virtual Memory - programmer views memory as large address space without concerns about the amount of physical memory or memory management. (What do the terms **32-bit** (or **64-bit) operating system** or **overlays** mean?)



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).

$$\left(\begin{array}{c}\text{effective memory}\\\text{access time}\end{array}\right) = \left(\begin{array}{c}\text{prob. of}\\\text{no page fault}\end{array}\right) * \left(\begin{array}{c}\text{main memory}\\\text{access time}\end{array}\right) + \left(\begin{array}{c}\text{prob. of}\\\text{page fault}\end{array}\right) * \left(\begin{array}{c}\text{page fault}\\\text{time}\end{array}\right)$$

$= (1 - p) * 50ns + p * 12,200,000$

$= 0.98 * 50ns + 0.02 * 12,200,000$

$= 244,049ns$

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!

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

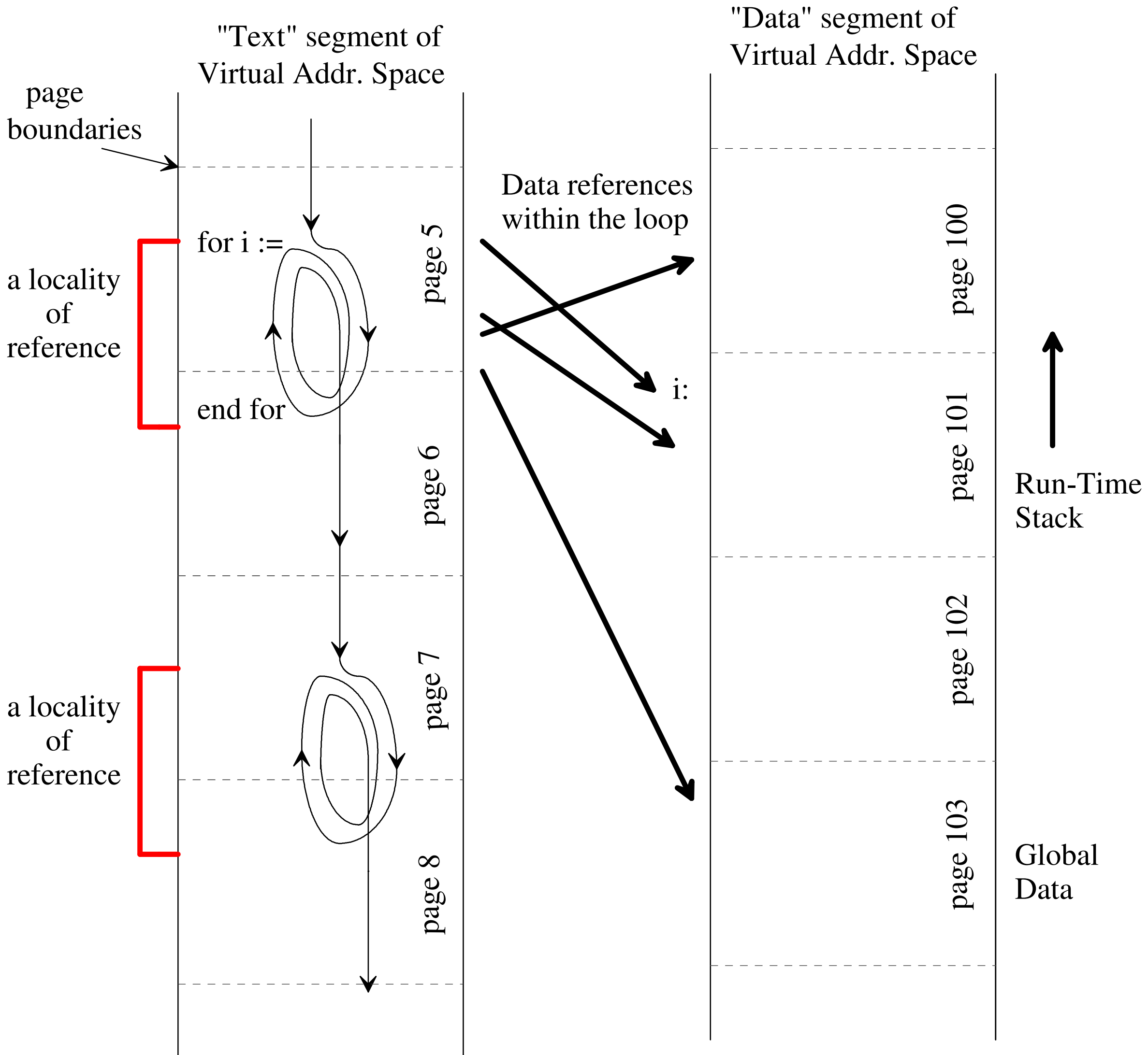
$55 = 50 - 50p + 12,200,000p$

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.

Typical Flow of Control in a Program



Possible reference string: 5, $(5, 101, 5, 5, 5, 101, 5, 5, 5, 100, 5, 103, 6, 6, 6)^{n}$, 6, 6, 6,
6, 7, 7, 7, 7, $(7, 7, 103, 7, 101, 7, 7, 7, 101, 7, 8, 8, 8, 100, 8)^{2n}$, 8, 8, ...

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

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.

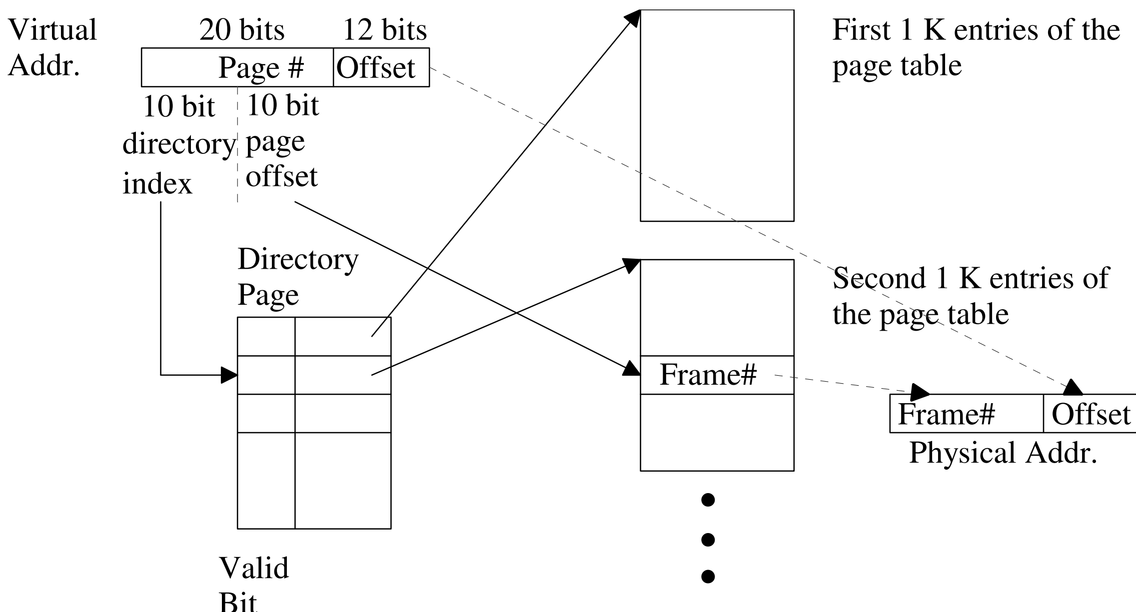
2) Ways to handle large page tables:

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

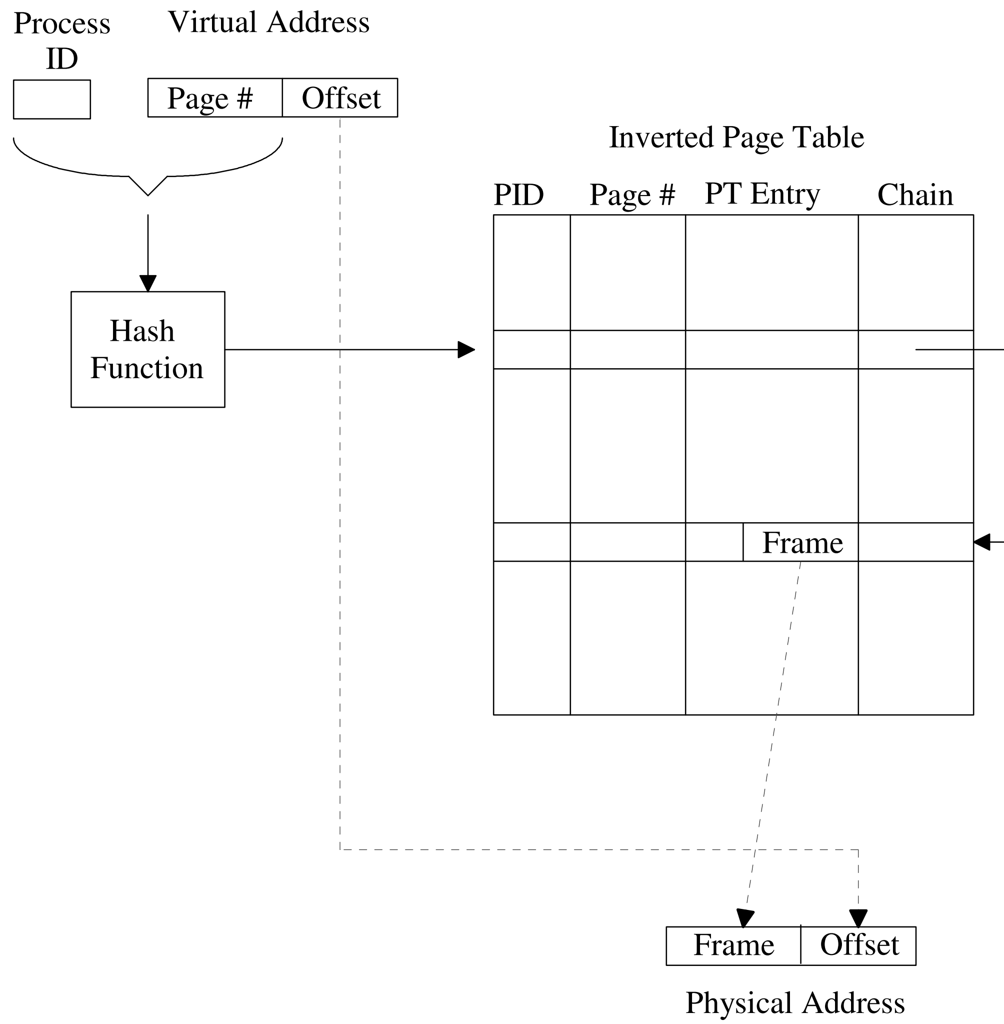
1 M (2^{20}) of page table entries, or 4MB for the whole page table with 4 byte page table entries

Solutions:

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.

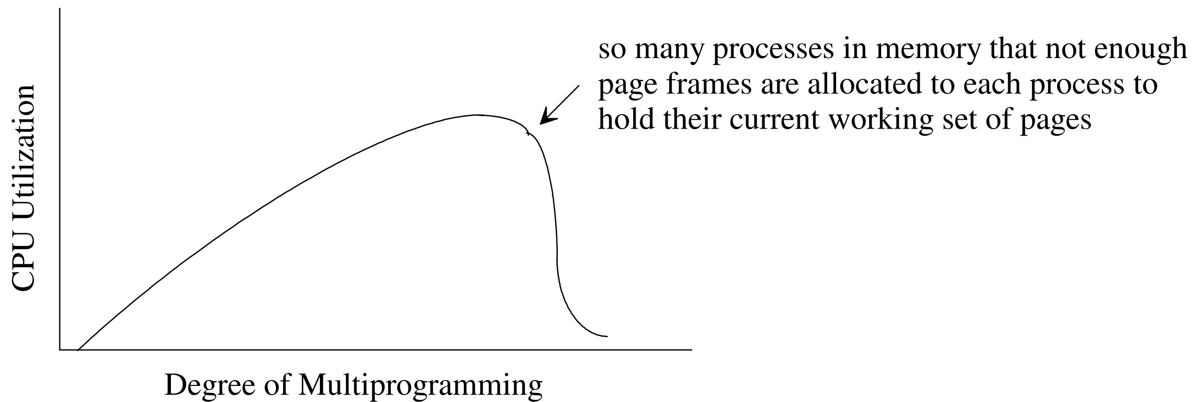


b) inverted page table - use a hash table of what's actually in the physical memory frames to reduce the size of the necessary data structures



Design issues for Paging Systems
Conflicting Goals:
1) Want as many (partial) processes in memory (high degree of multiprogramming) as possible so we have better CPU & I/O utilization ⇒ allocate as few page frames as possible to each process

2) Want as low of page-fault rate as possible ⇒ allocate enough page frames to hold all of a process' current working set (which is dynamic as a process changes locality)



Thrashing occurs when processes spend more time in page fault wait than doing useful work

Operating systems need to have
1) ***frame-allocation algorithm*** to decide how many frames to allocate to each process
2) ***page-replacement algorithm*** to select a page to be removed from memory in order to free up a page frame on a page fault

Page-Replacement Algorithm - selects page to replace when a page fault occurs
Reasonable page-replacement algorithms must
1) not be too expensive to implement w.r.t. time, hardware, or memory
2) have good phase transition - "forget"/replace pages of the old locality of reference when the program moves on to a new locality of reference.

Possible Page-Replacement Algorithms:
A. FIFO

References String:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Page Frames Allocated = 3	1*	2*	3*	4*	1*	2*	5*	5	5	3*	4*	4
		1	2	3	4	1	2	2	2	5	3	3
			1	2	3	4	1	1	1	2	5	5

9 page faults Page fault rate = 9/12

What if we allocate more page frames? What would we expect to happen to the page fault rate?

References String:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Page Frames Allocated = 4	1*	2*	3*	4*	4	4	5*	1*	2*	3*	4*	5*
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
				1	1	1	2	3	4	5	1	2

10 page faults Page fault rate = 10/12

We increased the number of page frames allocated and the page fault rate got **worse**!!!
Belady's anomally (IBM) - this does not usually happen with FIFO, but it can.

B. Optimal Page Replacement Algorithm - select page that's not needed for the longest time in the future (impossible to actually implement)

References String:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Page Frames Allocated = 3	1*	2*	3*	4*	4	4	5*	5	5	5	5	5
		1	2	2	2	2	2	2	2	3*	3	3
			1	1	1	1	1	1	1	1	4*	4

7 page faults Page fault rate = 7/12

This is the best we can do with 3 page frames allocated. Optimal algorithm does not suffer from Belady's anomally.

C. LRU (Least Recently Used) - uses principle of locality to approximate the optimal algorithm

References String:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Page Frames Allocated = 3	1*	2*	3*	4*	1*	2*	5*	1	2	3*	4*	5*
		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3

10 page faults Page fault rate = 10/12

LRU does not perform well for this string, but in general you can construct a reference string to make a page-replacement algorithm look bad (except for optimal alg.)
In general LRU is a good page-replacement algorithm.

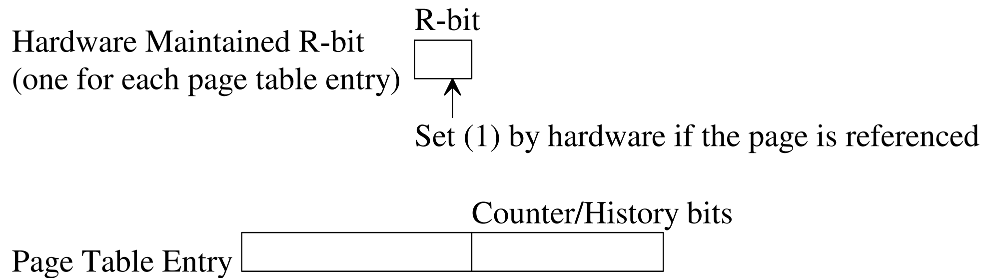
Implementation of LRU Algorithm
What information would we need to keep track of to implement LRU?

Either:
1) a counter in each page-table entry indicating the last time-of-use, or

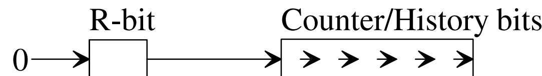
2) stack - (like above tables) indicating the order of usage.

Either of these faithful implementation of LRU would be expensive!!! Both would require a substantial amount of work with **each memory access**.

Approximations to LRU - that are less expensive to implement
Usually done in software with little hardware support, except for reference (R) bits that are maintained by hardware for each entry in the page tables. (The R-bits are maintained in the TLB)
A. Counter/History Bits - approximate time stamp



Periodically, say every 20 milliseconds, interrupt and have the OS shift the R-bit into the counter/history bits. Such as

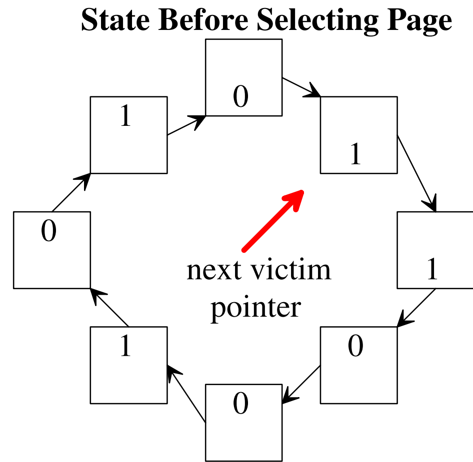


On a page fault, select the page with the smallest counter/history bits to replace. For example, consider the following collection of couter/history bits.

R	Counter/History bits	
0	0 1 0 1 0 1 0	
1	0 0 1 0 0 0 0	
0	0 0 0 1 1 1 1	← select for replacement since it was not access in last 4 intervals
0	1 1 0 1 0 1 0	← we don't know which of these was actually accessed last
0	1 1 0 1 0 1 0	←

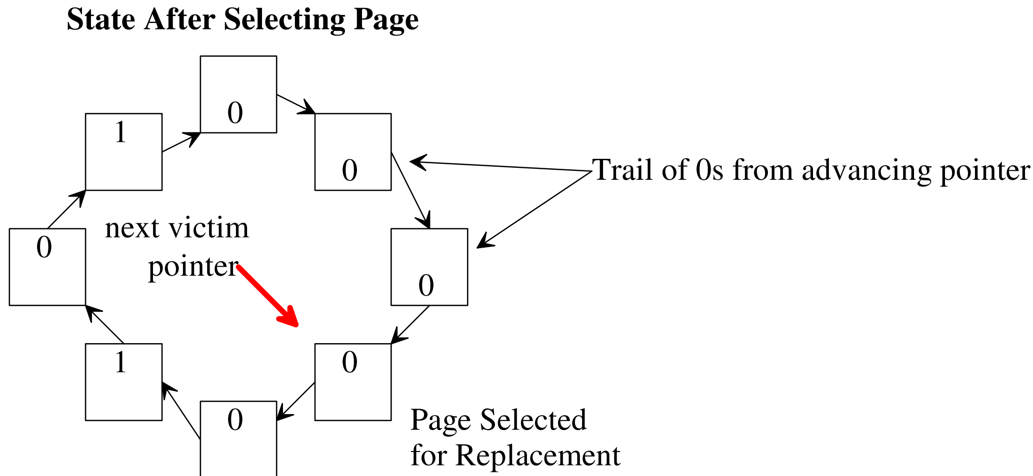
B. Second-Chance/Clock Replacement - only store one counter/history bit per page-table entry
For the pages in memory, maintain a circular FIFO queue of pages

State Before Selecting Page

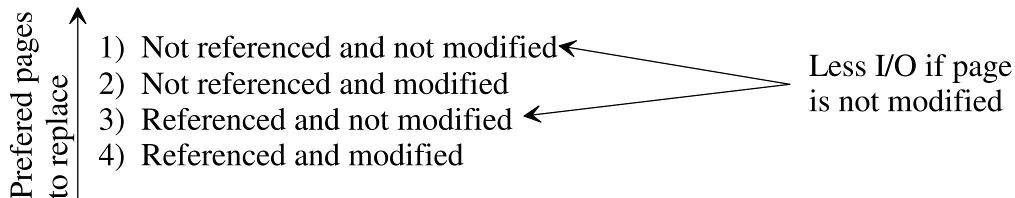


On a page fault, the pointer is moved until a page with a reference bit of 0 is found. This page is selected to be replaced. While the pointer is being moved, reference bits are cleared. As the pointer moves it leaves a trail of 0s in its wake. If one of these pages is referenced be for the pointer returns, it gets a "second change" and can remain in memory.

State After Selecting Page



C. Not Recently Used (NRU) / Enchanced Second-Chance Algorithm - in addition to the a single reference bit, use the modified/dirty bit to decide which page to replace.
On a page fault, the OS splits pages in memory fall into the following four categories based on their reference and modify bits.

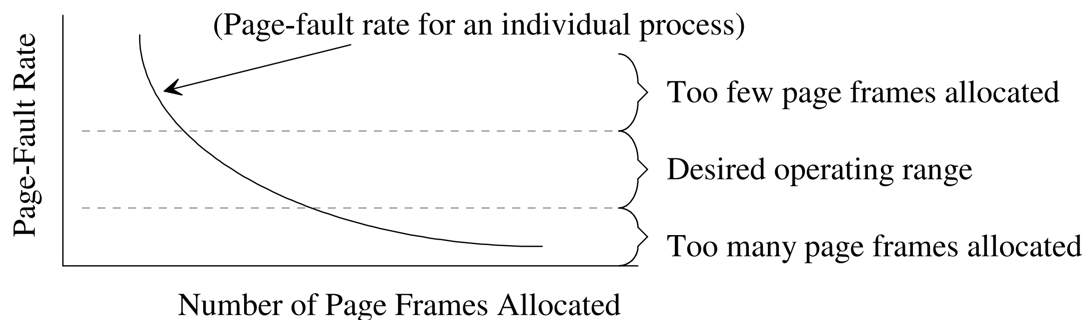


Frame-Allocation Algorithms - to decide how many frames to allocate to each process
Goal: Allocate enough frames to keep all pages used in the current locality of references without allocating too many page frames.
The OS might maintain a pool of free frames in a ***free-frame list***.

Allocation Policies:
1) ***local page replacement*** - When process A page faults, consider only replacing pages allocated to process A. Processes are allocated a fixed fraction of the page frames.

2) ***global-page replacement*** - When a page fault occurs, consider replacing pages from any process in memory. Page frames are dynamically allocated among processes, i.e., # of page frames of a process varies.
Advantage: as a process' working set grows and shrinks it can be allocated more or less page frames accordingly.
Disadvantage: a process might be replacing a page in the locality of reference for another process. Thrashing is more of a problem in systems that allow global-page replacement.

Implementation of global-page replacement
a) Page-Fault Frequency - OS monitors page-fault rate of each process to decide if it has too many or not enough page frames allocated



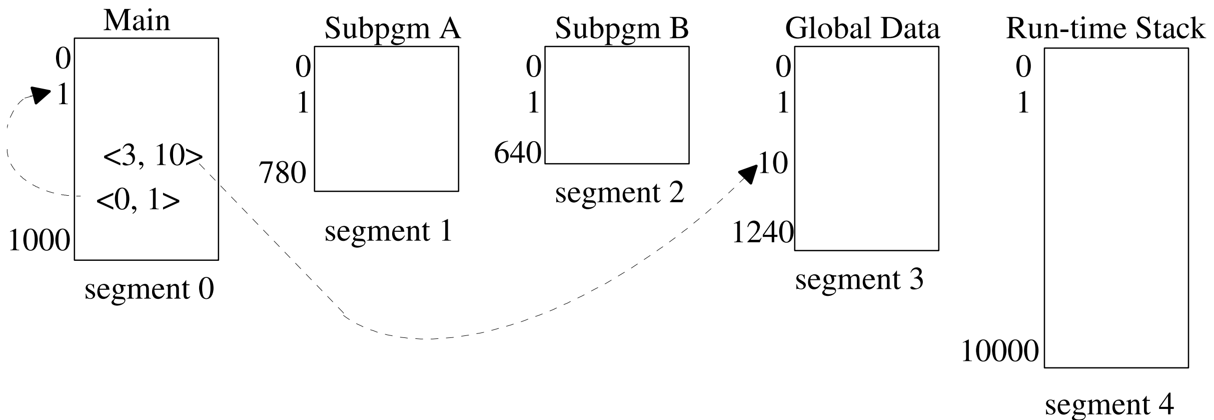
Additionally, free frames could be allocated from the free-frame list or removed from processes that have too many pages. If the free-frame list is empty and no process has any free frames, then a process might be swapped out.

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.

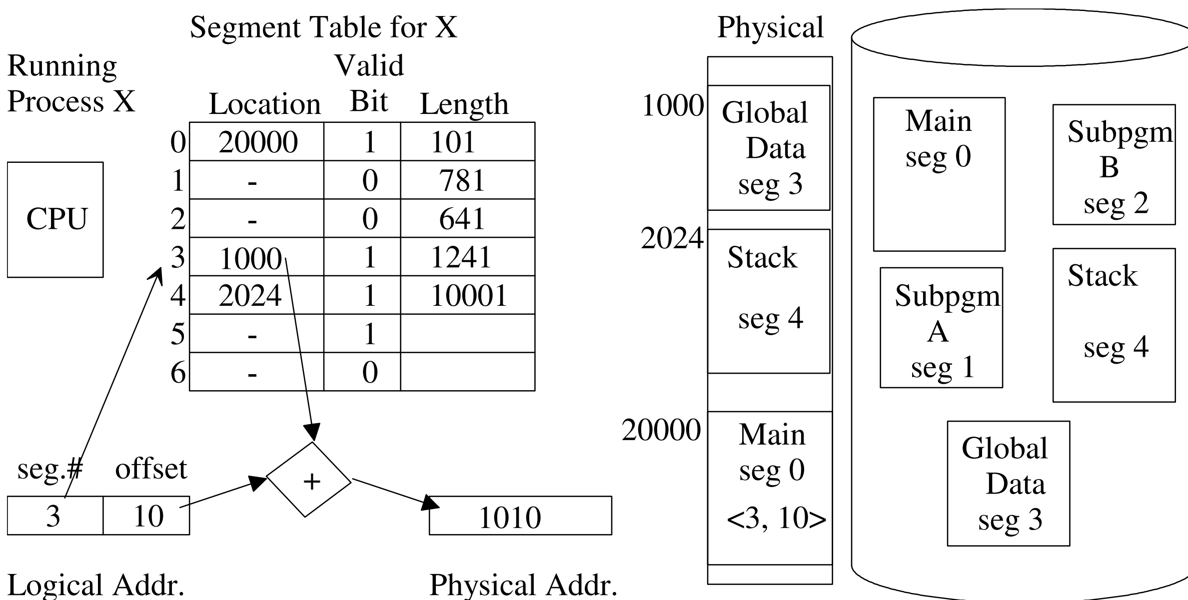
Segmentation - divides virtual address space in terms of meaningful program modules which allows each to be associated with different protection. For example, a segment containing a matrix multiplication subprogram could be shared by several programs.

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.



	Location	Valid Bit	Length
0	20000	1	101
1	-	0	781
2	-	0	641
3	1000	1	1241
4	2024	1	10001
5	-	1	
6	-	0	

Problems with Segmentation:

1) hard to manage memory efficiently due to internal 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.