

Locality Of Reference, so a few pages per Process need be Present, so Process can be > MM

## **2 PAGING AND VIRTUAL MEMORY**

Translation from VM address to MM address-page tables-MMU-Associative Memory

### **3 VIRTUAL MEMORY BASIC STRUCTURE**

**Indirection-VM advantage Trade-off Hardware**

Indexed Page Tables-1, 2, etc. Level

### **4 PAGE TABLES-INDEXED Indirection-Memory Trade-off Table Arrangement**

### **5 PAGE TABLES-INDEXED-RELATIVE SIZES**

### **6 PAGE TABLES-INVERTED ( HASHING), ASSOCIATIVE**

**Indirection-Memory Size Trade-off**

Hashed Page Tables-without and with Chaining

### **7 HASHING-LINEAR SECONDARY PROBE NON-CHAINING**

### **8 HASHING WITH CHAINING**

Overhead, Page Table and 1/2 page unused, vs Process access time

### **9 CHOOSING PAGE SIZE**

Grouping Pages into Meaningful Groups

### **10 SEGMENTATION-A MEANINGFUL UNIT AND PAGING**

### **11 SEGMENTATION TABLES SHARED SEGMENTS**

### **12 PAGED SEGMENT TABLES**

### **13 MEMORY: OTHER CONSIDERATIONS: WRITEBACK, PAGE FAULTS.**

High Hit Ratios are necessary for successful use of caching

### **14 FASTER SMALLER COMPONENT SUPPLEMENTS SLOWER LARGER**

**ONE TO IMPROVE AVERAGE SPEED PERFORMANCE Performance Analysis**

### **15 THE LONG-LONG TRAIL TRAVELED IN FINDING A PAGE**

Page Faults and Process state

### **6 PROCESS STATES WITH PAGING**

## **CONTENTS**

## Good Utilization = Many Processes

The more Processes that can be in MM at once the less likely ( $p^n$ ) all will be in the Waiting for IO state at once, and therefore the greater the probability ( $1 - p^n$ ) that the CPU will be Active with a User Process, and the greater the Utilization will be.

## Problems With Holes & Processes

### 1. Fragmentation-Compaction

In the **Holes and Processes** world, having **many Processes in memory will take large amounts of space, also it** can lead to (external) **fragmentation** (lots of little Holes, but none big enough for a Process). This may require moving entire Processes in and out of MM or even positioning Processes one after the other, **compaction**- very time consuming.

### 2. Overlays

A Process may be larger than MM. This was originally handled by "**Overlays**". To handle an **oversized program**, say  $P = P1 + P2$ , the programmer designed P so that a part of it, namely P1, runs initially. At the point in P1 at which more of P is needed a **read(..)** command in P1 reads in P2, to a position which overlaid part of P1. Programming like this is complex requiring the application programmer to consider positioning overlays on program already in MM. This should not be an application programmers concern.

---

## Paging Gives Solutions

### Basic Paging

**Paging involves:** partitioning MM into **fixed size pages** (*perhaps 2 different sizes*), of a size ranging from  $2^8$  to  $2^{12}$  and **addressed by page** with a pair of numbers, **<MM page number, offset>**. The **[MM page number] multiplied by the [page size]** locates the page address in the part of MM devoted to Processes. The offset part of the address locates the byte addressed within that page. When Process, P, is first moved into MM (In the Create state) an initial single page or small group of pages,  $G_1$  into are move into MM. from the Disk file where they resides. Only when P is Active and a page outside,  $G_1$ , is addressed from within  $G_1$ , is **a new page brought into MM**, perhaps replacing a page. On the other hand when a page of a process still in MM has not been used for a time **that page could be returned to a back-up Disk**. In general, space for **only a part of a Process, consisting of several pages, need be in MM** at any time, and these need not be contiguous.

### 1' Improved Utilization

Thus **more Processes can be candidates for CPU** use than if entire Processes had to be in MM in order to run. This improves Utilization. Furthermore the handling of a paged memory can be managed by the OS assisted by added hardware thus removing the need for the programmer to be involved. Since the biggest "Hole" or unused part at the end of a Process is  $< 1$  page, and pages need not be contiguous, the major allocation problems are solved. Thus many Processes can be represented in MM, each by only a few of its pages.

---

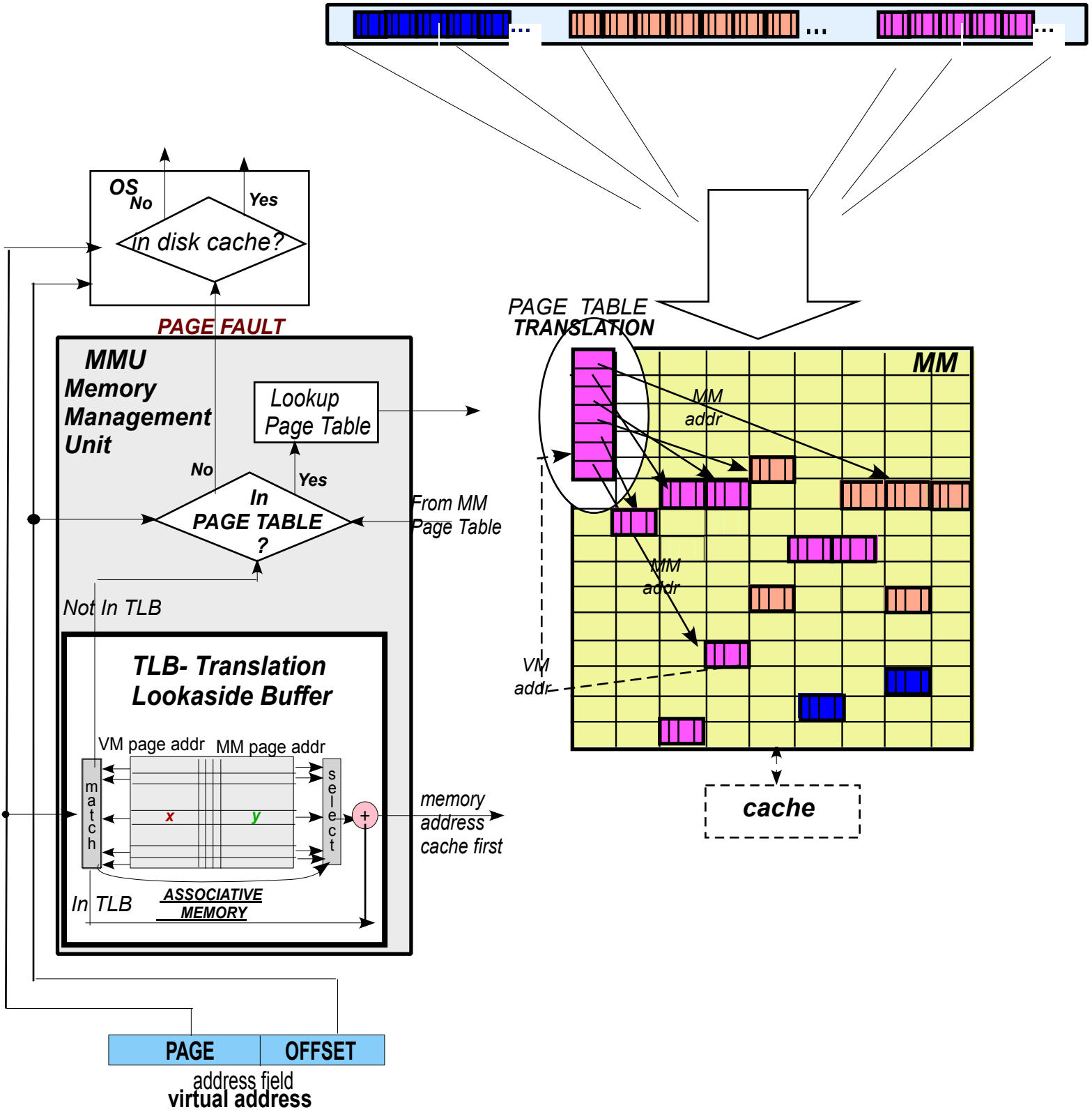
## Paging Gives Virtual Memory Bonus

### 2' Overlay Elimination

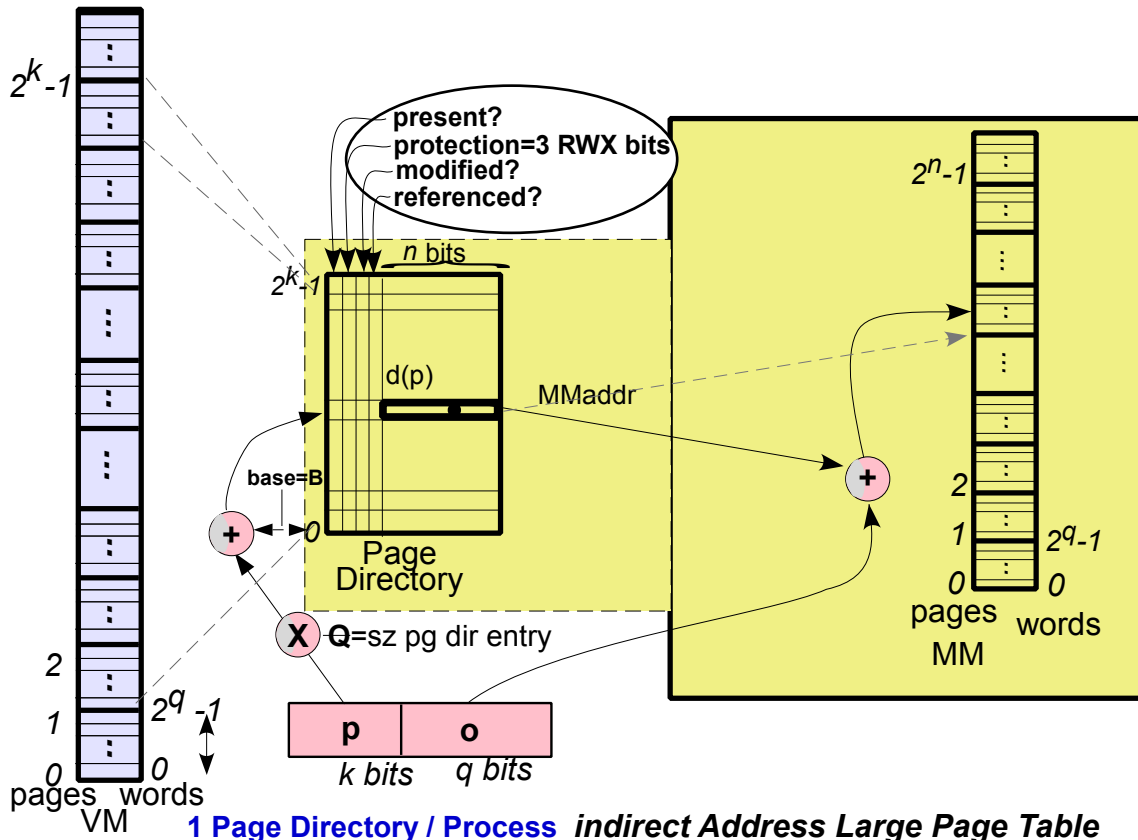
Now, since only part of any Process need ever be in MM a Process can be larger than all of MM, it can exist in a **virtual memory (VM)**. With VM. each instruction is also addressed with two numbers **<VM page number, offset>**. Now however, **the VM page number can be much larger than the largest MM pages**. The offsets, and page size in MM and VM are the same, only the number of pages present differ. ( However now it is necessary to convert VM page numbers to MM page numbers.)

## PAGING AND VIRTUAL MEMORY

(VM) VIRTUAL MEMORY-In Files on Disk



**VIRTUAL MEMORY BASIC STRUCTURE**  
Indirection-VM advantage Trade-off Hardware



**1 Page Directory / Process indirect Address Large Page Table**

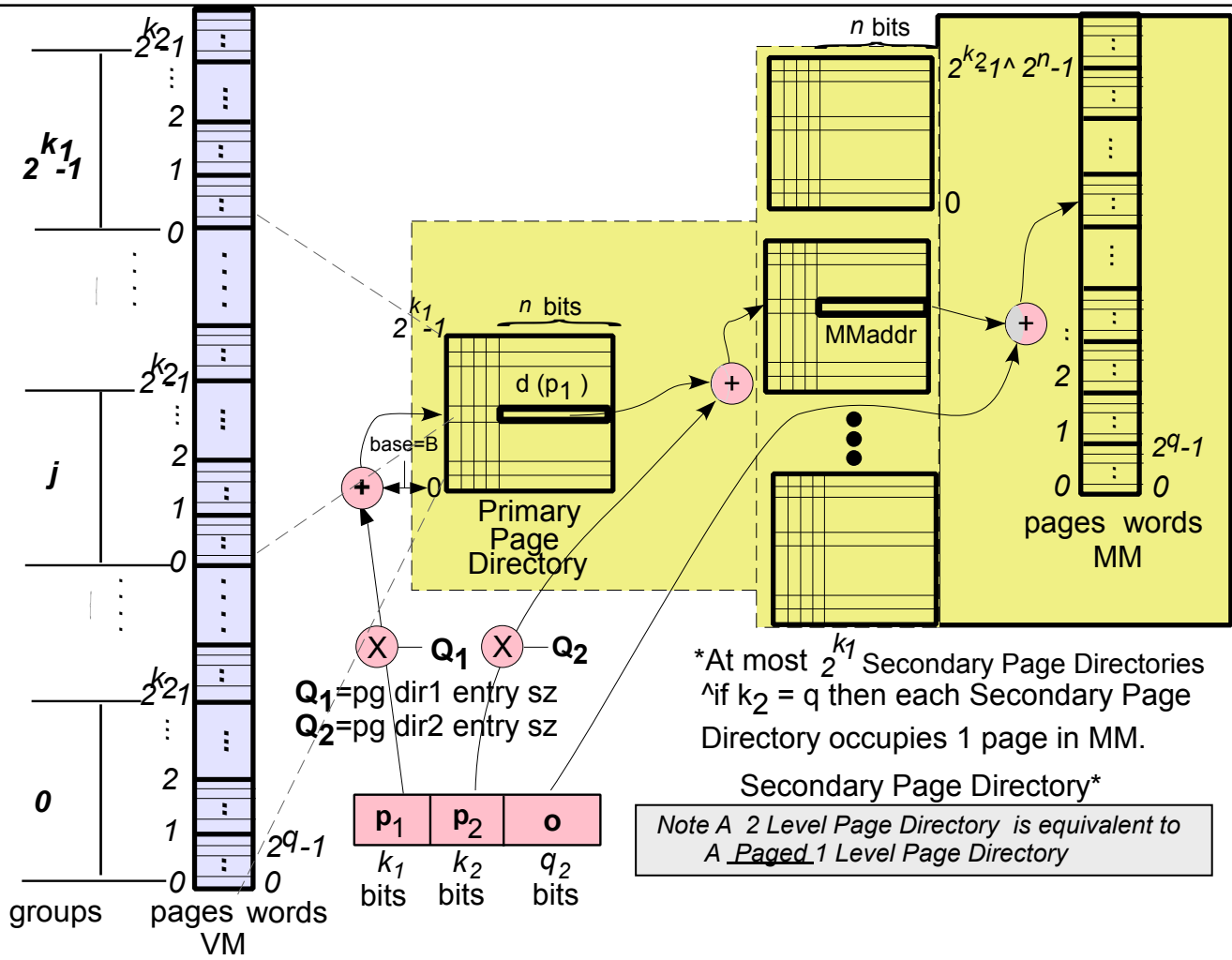
Virtual Memory

Permanent Tables Main Memory

Paged in Tables Main Memory

Hardware

Software

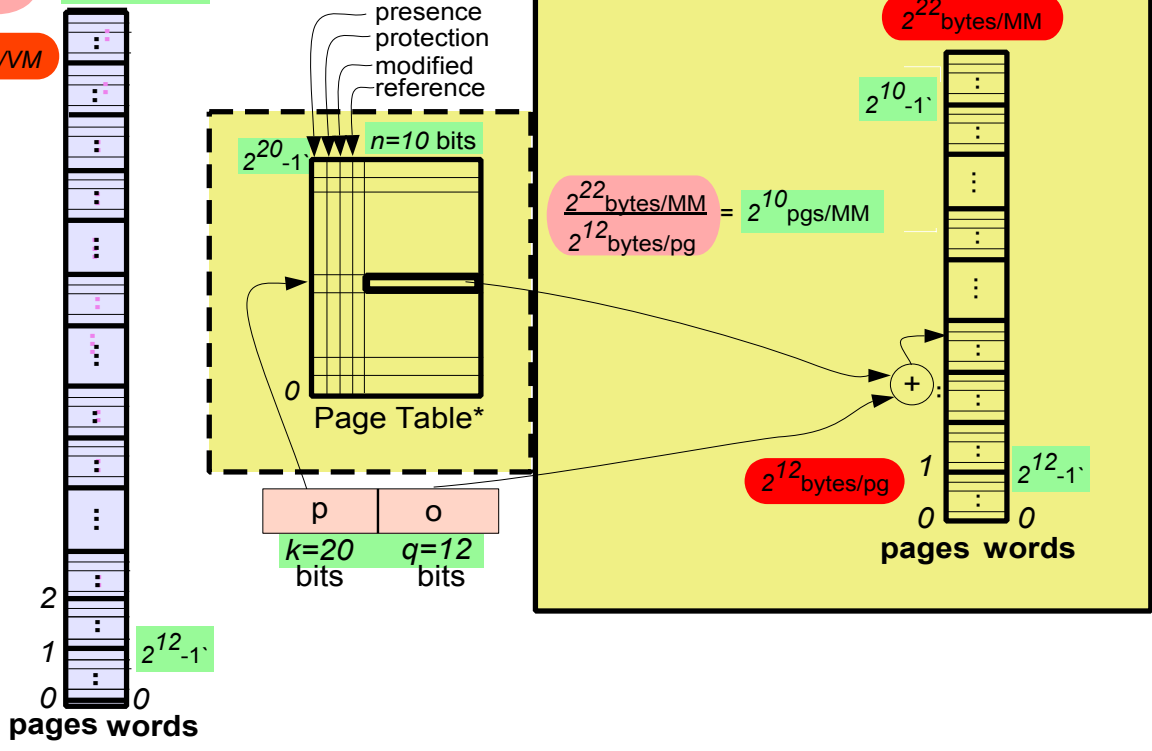


**1 Page Primary Directory / Process or All Processes Double Indirect Smaller Page Table**

**PAGE TABLES-INDEXED Indirect Access-Memory Size Trade-off Table Arrangement**

$$2^{20} \text{ pgs/VM} \times 2^{12} \text{ bytes/pg} = 2^{32} \text{ bytes/VM}$$

$$2^{20} \text{ pgs/VM}$$

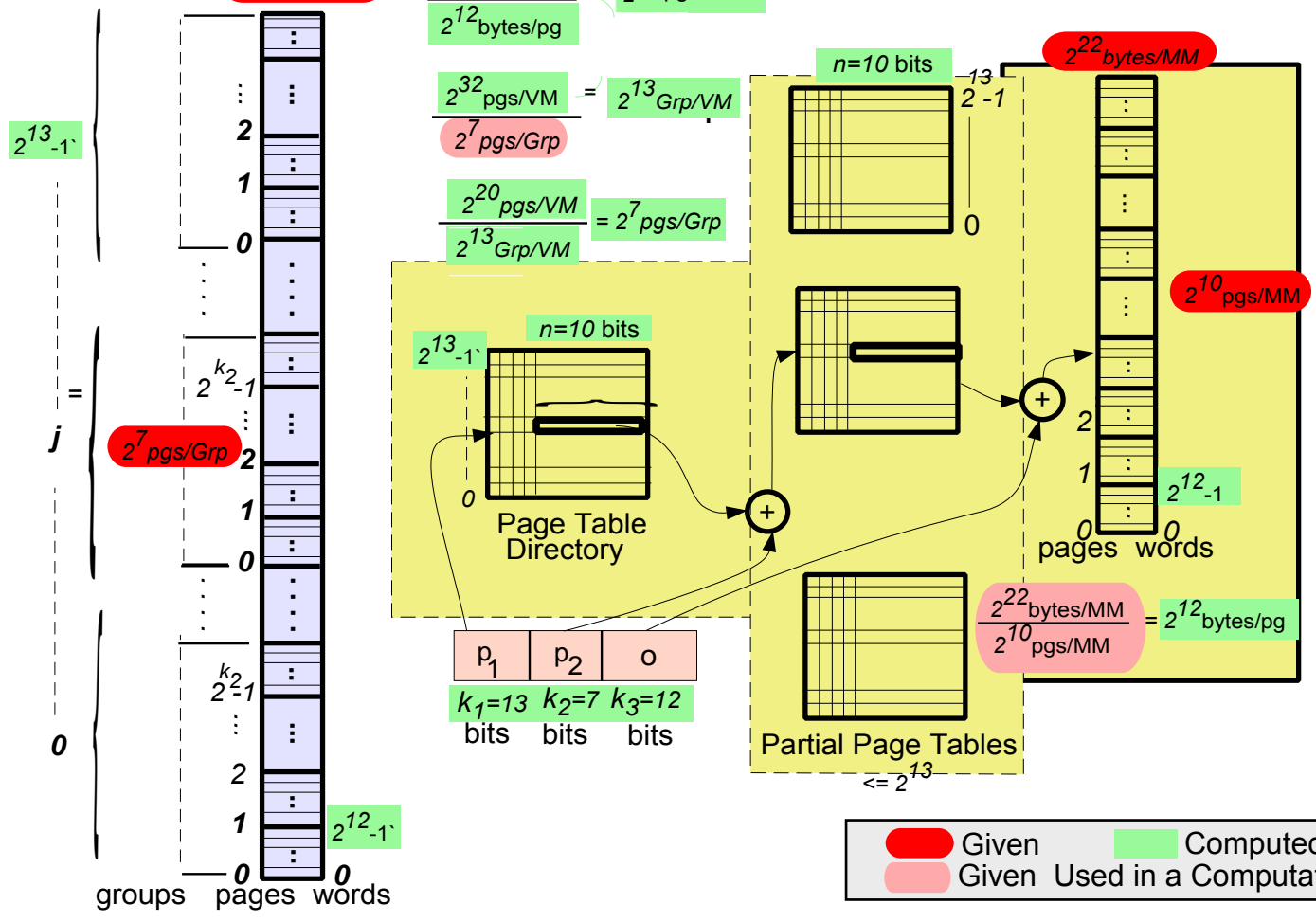


$$2^{32} \text{ bytes/VM}$$

$$\frac{2^{32} \text{ bytes/VM}}{2^{12} \text{ bytes/pg}} = 2^{20} \text{ pgs / VM}$$

$$\frac{2^{32} \text{ pgs/VM}}{2^7 \text{ pgs/Grp}} = 2^{25} \text{ Grp/VM}$$

$$\frac{2^{20} \text{ pgs/VM}}{2^{13} \text{ Grp/VM}} = 2^7 \text{ pgs/Grp}$$

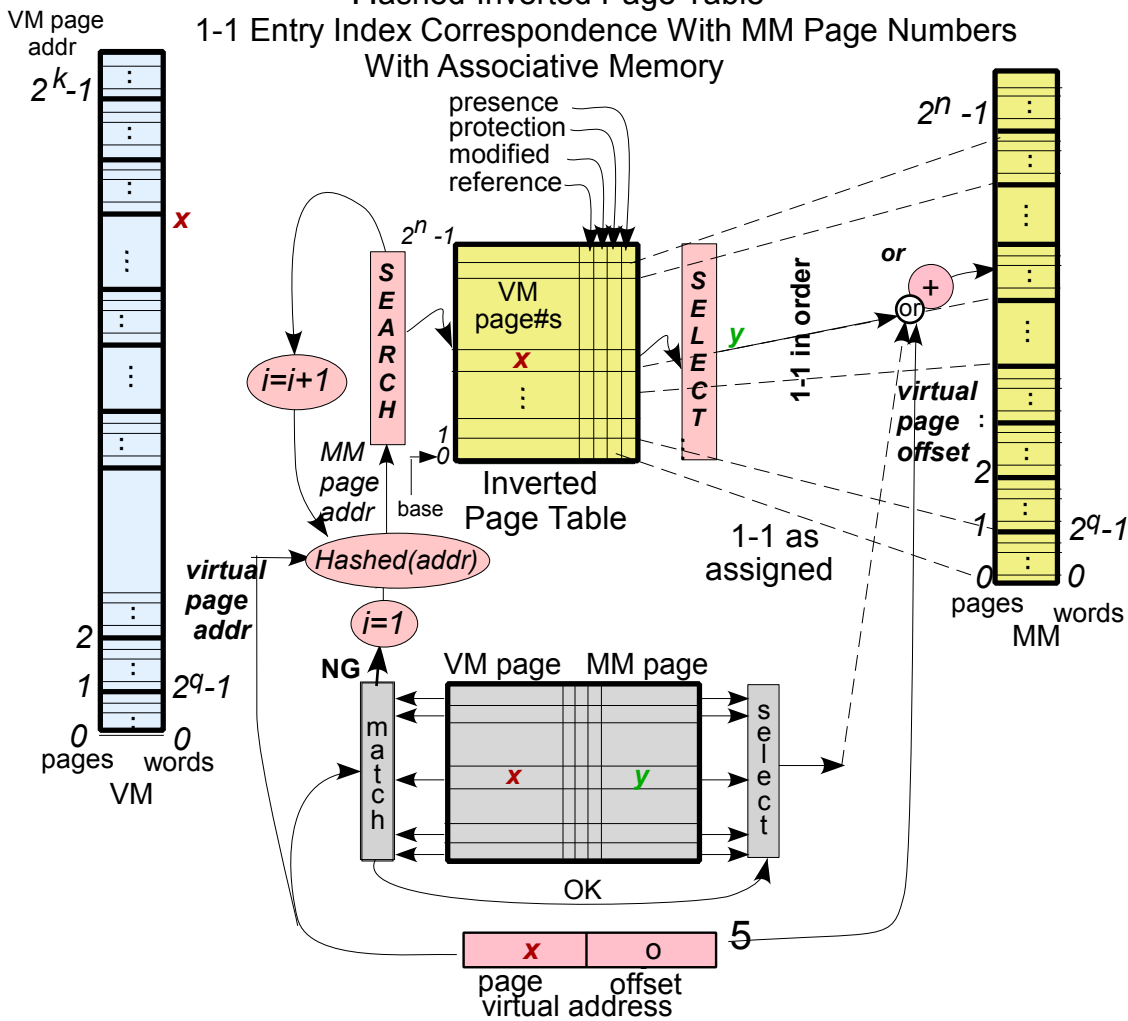


Given	Computed
Given	Used in a Computation

### PAGE TABLES-INDEXED-RELATIVE SIZES

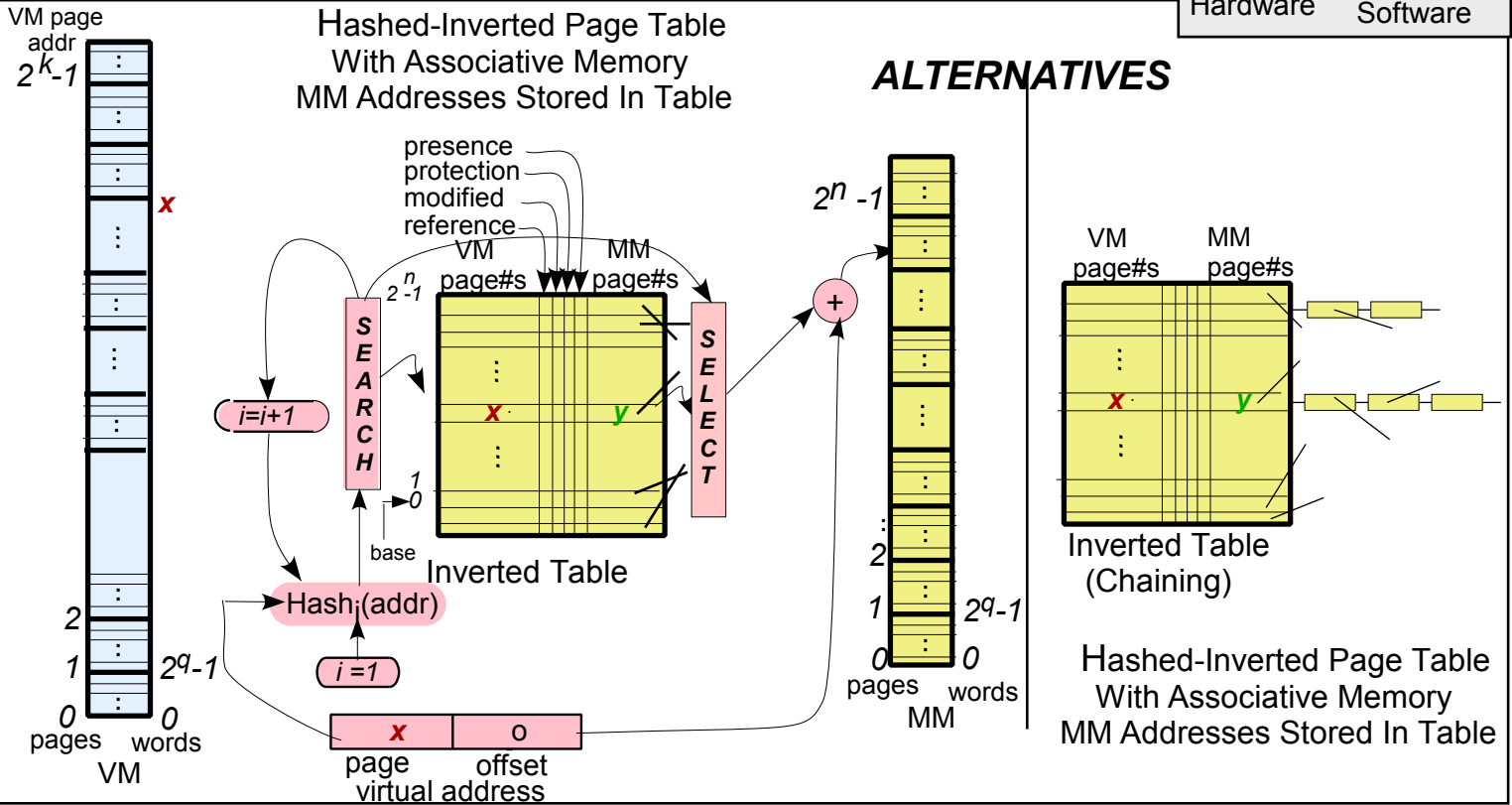
# Hashed-Inverted Page Table

1-1 Entry Index Correspondence With MM Page Numbers  
With Associative Memory



Hashed-Inverted Page Table  
With Associative Memory  
MM Addresses Stored In Table

## ALTERNATIVES

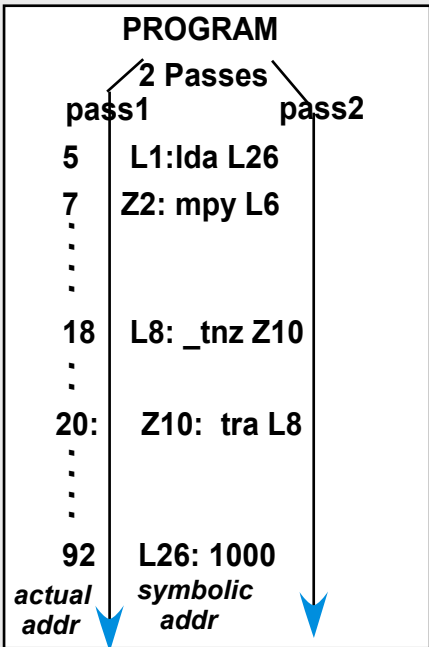


## PAGE TABLES-INVERTED ( HASHING), ASSOCIATIVE Indirection-Memory Size Trade-off

**HASH FUNCTION**  
*X = English Letter Cardinality*  
*Y = integer[1-26]     $\lfloor \quad \rfloor$  = floor*  
 Hash1(XY) =  $\lfloor X/2 \rfloor$   
 Hash2(XY) =  $\lfloor Y/2 \rfloor$   
 Hashj=3...(XY) =  $\lfloor [Y/2 + j - 2] \bmod 14 \rfloor$  :j > 2\*  
 \* Hashj(XY) =  $\lfloor [Y/2 + j] \rfloor$  :j-2= 1,2,...,13, 0,1...13,..

Enter

Find



Hash1(L1) = 12/2 = 6 ok  
1 probes

Hash1(Z2) = 26/2 = 13 ok  
1 probe

\*Hash1(L8) = 12/2 = 6 full  
Hash2(L8) = 8/2 = 4 ok  
2 probes

\*Hash1(Z10) = 26/2 = 13 full  
Hash1(Z10) = 10/2 = 5 full  
2 probes

\*Hash1(L26) = 12/2 = 6 full  
Hash2(L26) = 26/2 = 13 full  
Hash3(L26) = 13 + 1 = 0 ok  
3 probes

Hash1(L1) = 12/2 = 6 OK=5  
1 probes

Hash1(Z2) = 26/2 = 13 OK=7  
1 probe

\*Hash1(L8) = 12/2 = 6 Collision  
Hash2(L8) = 8/2 = 4 OK=18  
2 probes

\*Hash1(Z10) = 26/2=13 Collision  
Hash1(Z10) = 10/2=5 OK=20  
2 probes

\*Hash1(L26) = 12/2 = 6 Collision  
Hash2(L26) = 26/2=13 Collision  
Hash3(L26) = 13 + 1 = 0 OK=26  
3 probes

Enter

Lookup

Find\*

HASH TABLE	
Key	Addr
0	L26
1	
2	
3	
4	L8
5	Z10
6	L1
.	.
.	.
13	Z2

Lookup of these 5 entries with equal likelihood requires  
 $(1 + 1 + 2 + 2 + 3) / 5 = 1.8$  average probes per lookup

**ILLUSTRATES: ENTER and LOOKUP OF HASH TABLE**

**OTHER FUNCTIONS CAN INCLUDE:**

**REMOVAL-Must leave place holder GROWTH-Rehash All (like Compaction) SPEEDUP-Enlarge Table**

**HASHING-LINEAR SECONDARY PROBE NON-CHAINING**

# HASH FUNCTION

Hash1[X|Y] = X/2  
with CHAINING

Enter

\*Find

\*Hash1(L1) = L/2 = 12/2 = 6  
1 probe

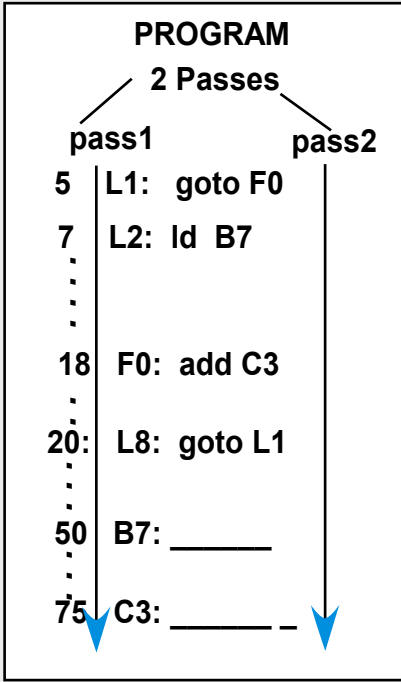
Hash1(L2) = L/2 = 6 ->  
2 probe

Hash1(F0) = 6/2 = 3  
1 probe

Hash1(L8) = L/2 = 6 -> +->  
3 probes

\*Hash1(B7) = B/2 = 2/2 = 1  
1 probe

\*Hash1(C3) = C/2 = 3/2 = +->  
2 probes



\*Hash1(L1) = L/2 = 12/2 = 6  
1 probe

Hash1(L2) = L/2 = 6 ->  
2 probe

Hash1(F0) = 6/2 = 3  
1 probe

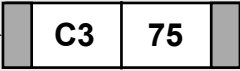
Hash1(L8) = L/2 = 6 -> ->  
3 probes

\*Hash1(B7) = B/2 = 2/2 = 1  
1 probe

\*Hash1(C3) = C/2 = 3/2 = +->  
2 probes

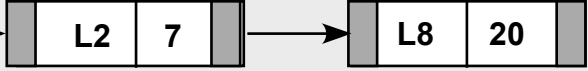
Enter

HASH TABLE	
0	
1	B7 60
2	
3	F0 18
4	
5	
6	L1 5
...	...
13	



\*Find

Ave size of non 0 length chains  
= (2+1+3)lengths / 3chains 2/chain  
if all 14 entries were made in the worst  
case the average non-0 chain length  
would be 14

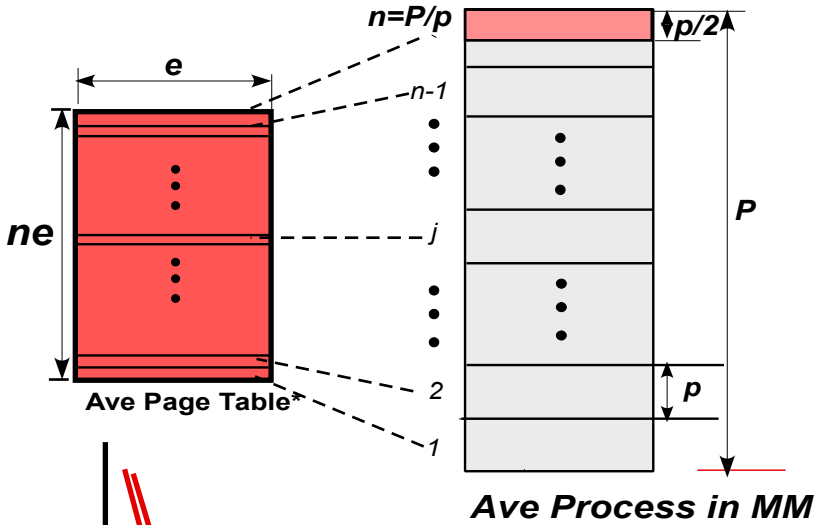


**ILLUSTRATES: ENTER and LOOKUP OF HASH TABLE**

**OTHER FUNCTIONS CAN INCLUDE:**  
REMOVAL Shorten Chain GROWTH Longer chains or Large Table SPEEDUP -Increase Table Size

## HASHING WITH CHAINING

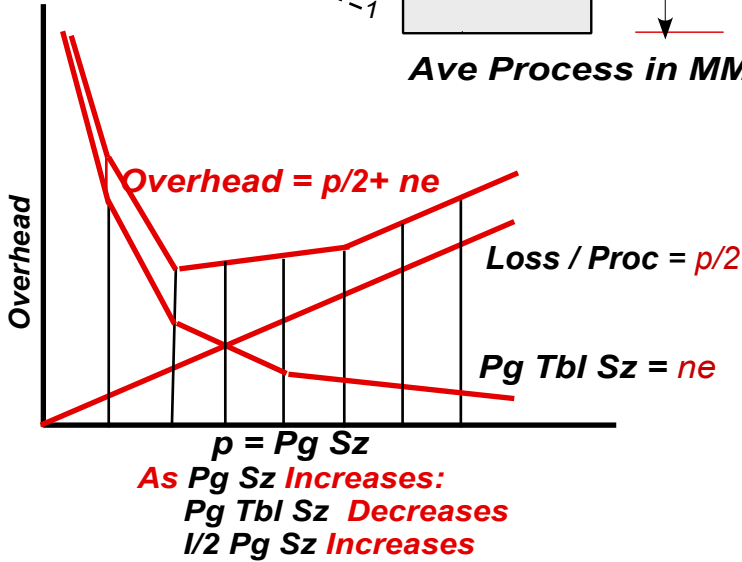
**MEMORY OVERHEAD: (WORST CASE?)  
DEPENDENCE ON PAGE AND PROCESS SIZE**



Definitions: (All are averages)

- $P$  = Process Size (bytes)
- $p$  = Page Size (bytes)
- $n$  = Ave Number of Pages in a Process =  $(P/p)$
- $n$  = Number of Page Table Entries per Process
- $e$  = Size in bytes of a Page Table Entries

$O$  = Overhead [per Process]  
 = Ave Page Table Size + Unused Page Part [per Process]  
 $O = ne + p/2$   
 $= (P/p)e + p/2$



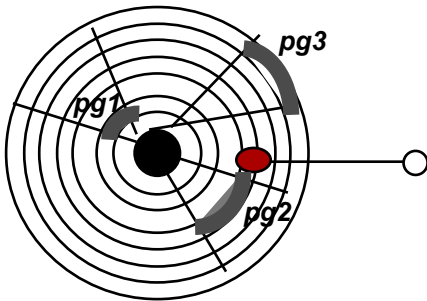
$dO/dp = e d((P/p) + p/2)/dp$   
 $dO/dp = -eP/p^2 + 1/2 = 0$  (Minimize)

$p_{optimal} = \sqrt{2Pe}$

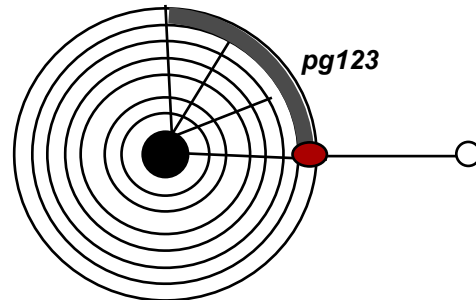
$\sqrt{2(2^{11}32)} = \sqrt{2^{11+3}} = 2^7$

$\sqrt{2(2^932)} = \sqrt{2^{9+3}} = 2^6$

(Like Bit Map Overhead, But No Holes To Account For, but **How About the Free list**)



$3 \text{ seeks} - 3\text{pgs} = B$

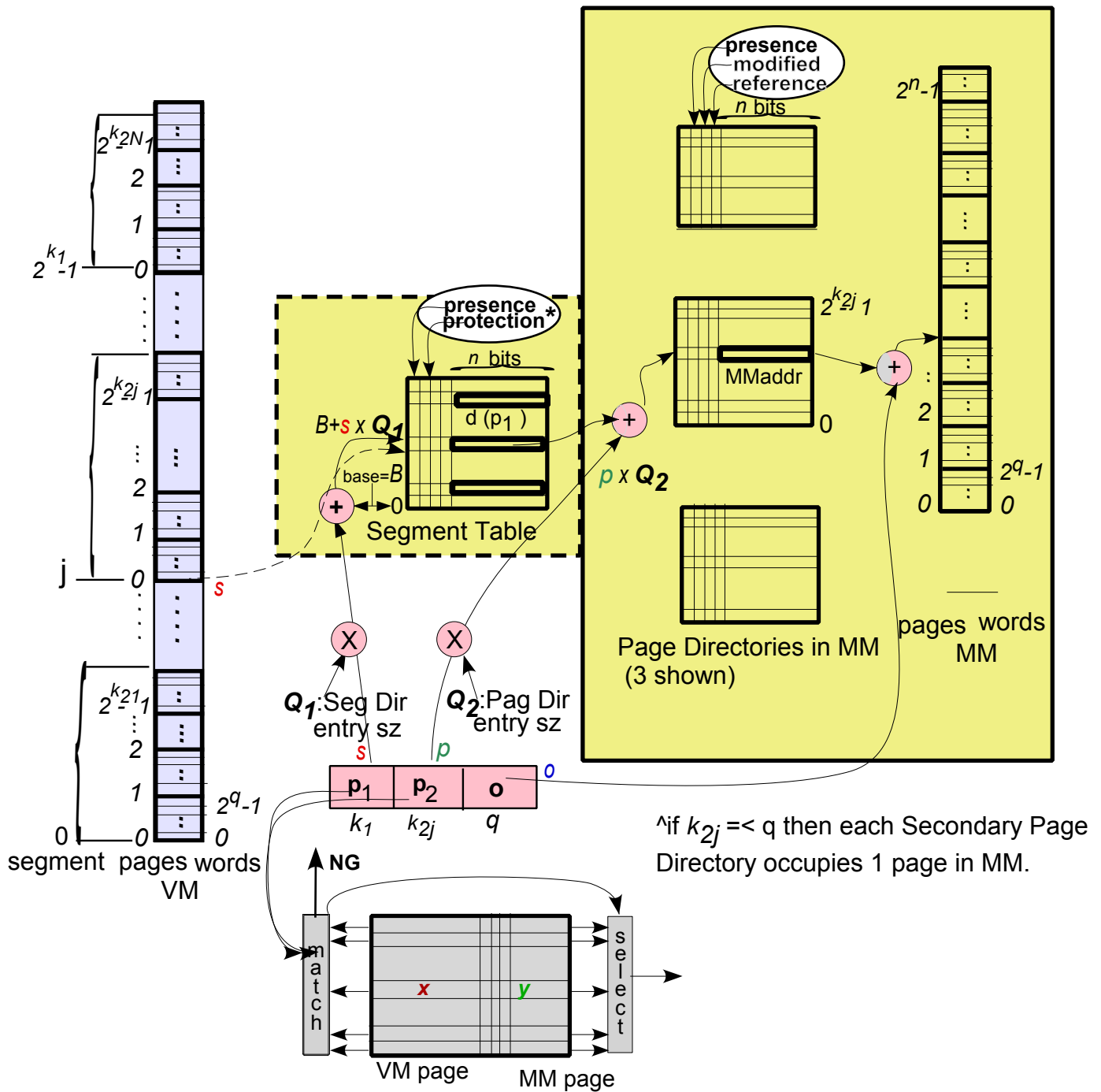


$1\text{seek} - 1\text{pg} \text{---size} = \text{size pg1} + \text{size pg2} + \text{size pg3} = B$

However page size also effects the time to move B bytes of a Process between MM and Disk. If page size is small then movement of B bytes will require the movement of more pages than if page size is large. To find a page on Disk requires the movement of the read-write head to its position on the Disk as well as the reading time of the page, which is proportional to the size of the page. The *positioning time is generally much larger than the read time once the page is located* so Disk access time for B bytes increases with decreasing page size.

**PAGE SIZE AND READ-WRITE TME ON DISK**

**CHOOSING PAGE SIZE**

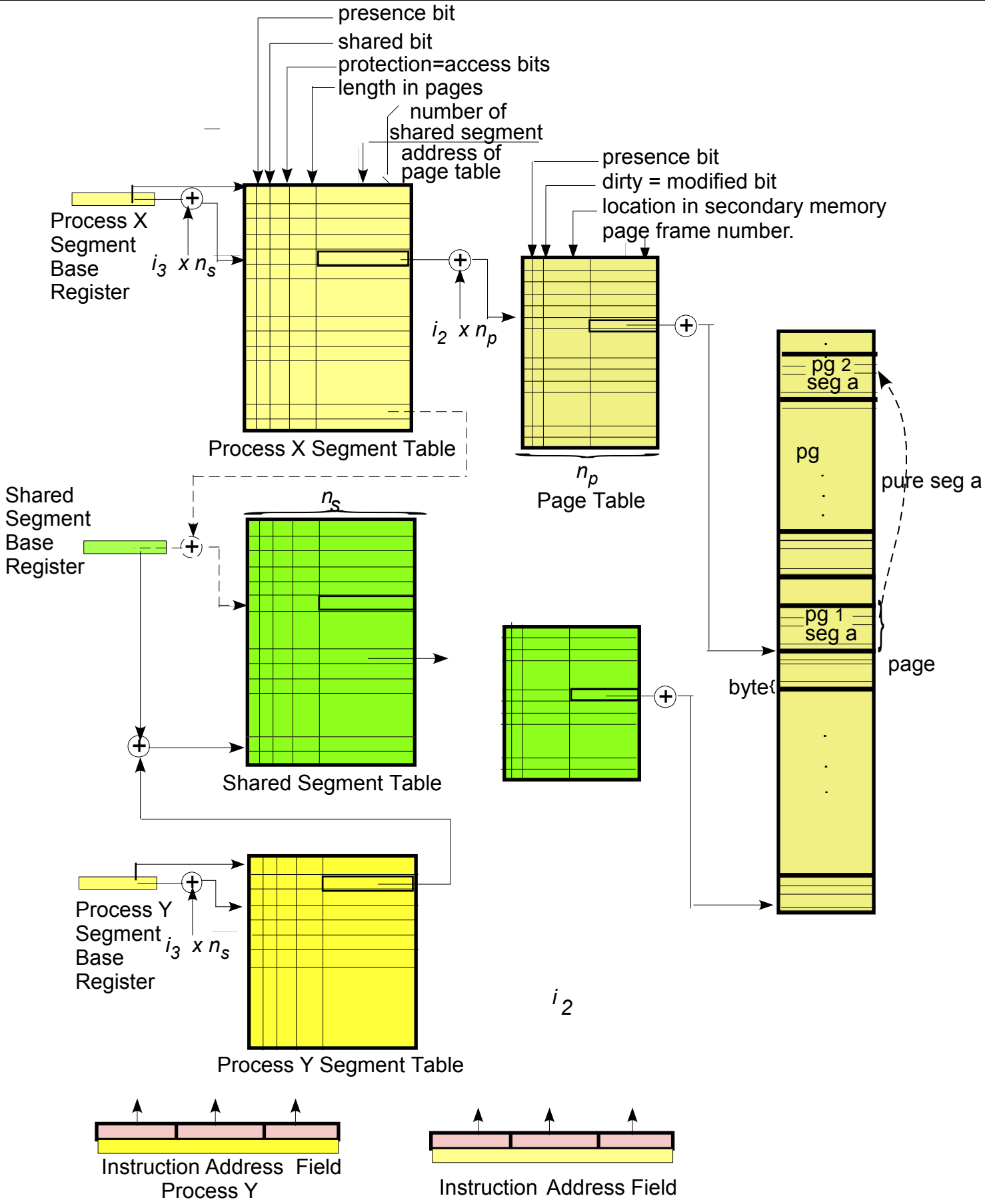


^if  $k_{2j} \leq q$  then each Secondary Page Directory occupies 1 page in MM.

\*if there is 1 segment table for all Processes protection = 9 bits 3 RWX bits for pid, for guid, others  
**Pages are not a good unit for protection but Segments are.**  
**Data - Stack - Text (Read-Only)**

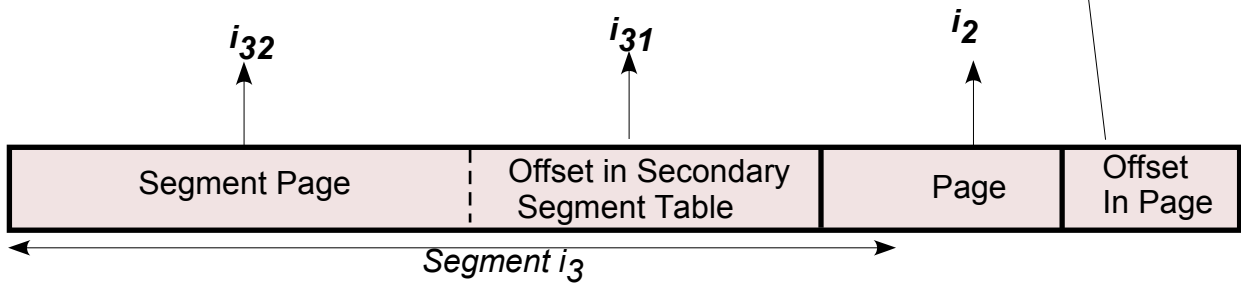
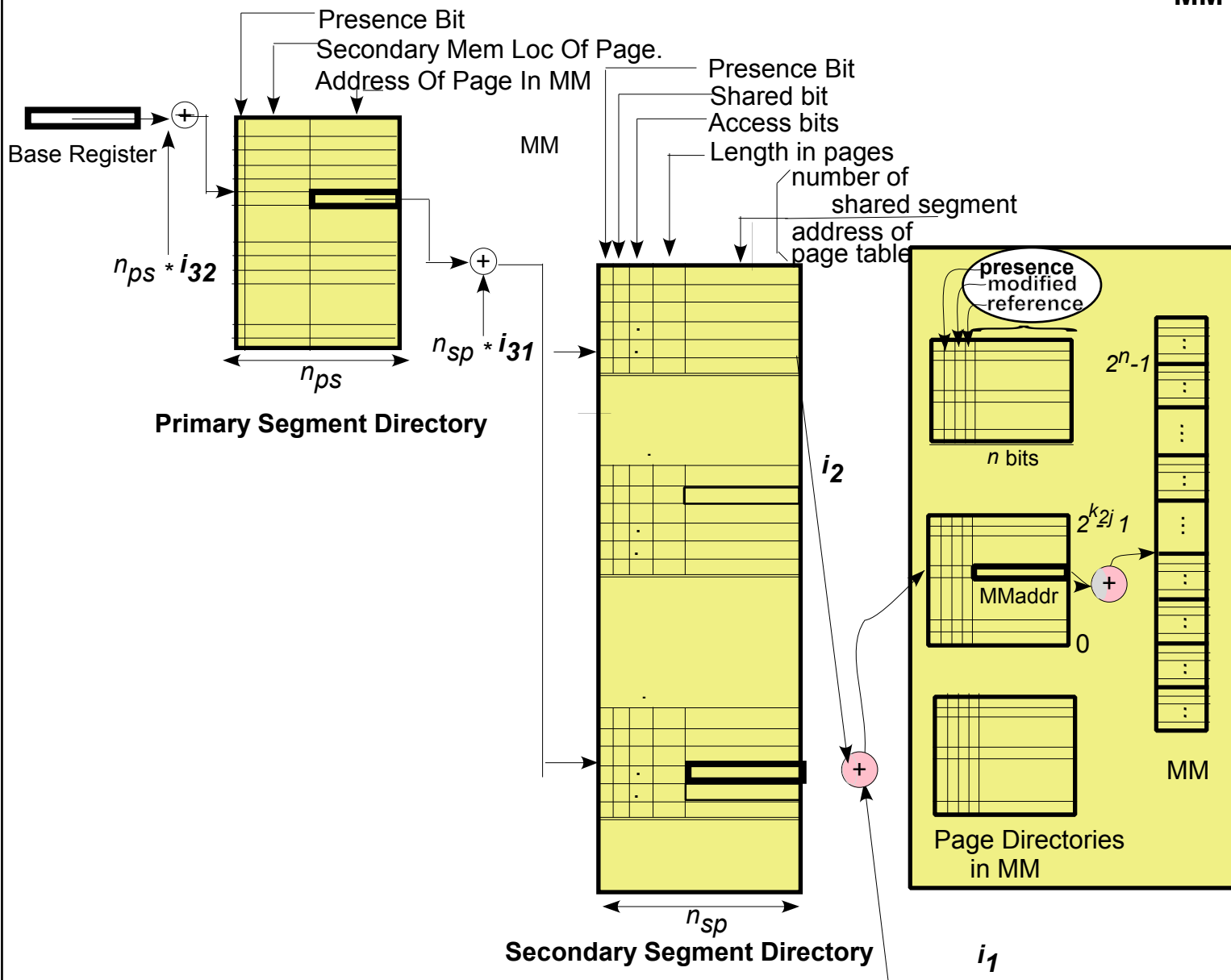
Max No.Of Segments =  $2^{k_1}$   
 Max No.Of pages in a Segment =  $\text{Max}_j(2^{k_{2j}})$

**SEGMENTATION-A MEANINGFUL UNIT AND PAGING**  
**1 Segment Table For all Prozesse Possible**



***A Procedure May be Accessed By More Than 1 Process. if Its Re-entrant Code it can be shared***

**SEGMENTATION TABLES SHARED SEGMENTS**



This is Equivalent To Partitioning All entries in the Original Single Segment Table into groups of equal size (= to the size of 1 page) Then each entry in the new Primary Segment Directory corresponds to a group of entries in the Original Segment Table. Now each entry in the Primary Segment Table points to a page containing the group of entries to which it corresponds.

### PAGED SEGMENT TABLES

## Assignment Of Pages

Processes in MM at the same time use disjoint sets of Pages . These are assigned either:

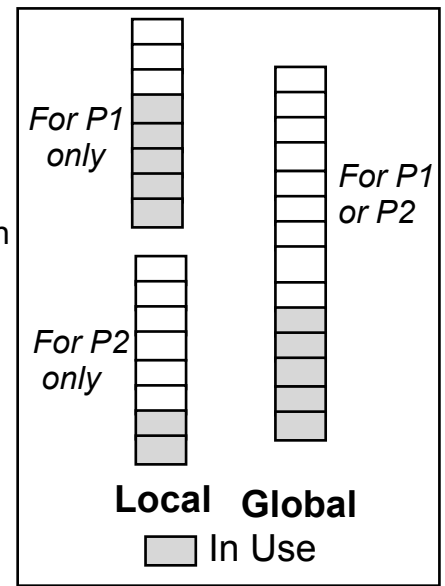
**Locally** (Each Process assigned a Number of Pages): Can have thrashing\* if one is at its limit even while other use only a fraction of their allotment.

**Globally** All pages are accessible to all Processs (A Common Pool) To each according to its need-so if one is using very little another can use more rather than thrashing. To measure a Processes' need for pages use:

**Page Fault Frequency(PFF)** = # of page faults/unit time. If large it needs more pages, if small needs fewer.

\*Keep having page faults from all then: needing to remove a page in order to bring in a page

**Pooling Resources**  
Making Space Available To More than 1 variable size resource



## Sharing Page-Segment Tables

Separating RO (Read Only) or I (Instructions)-Space, from RW(Read-Write) or D (Data)- Space:

Separate Page (Segment) Tables is useful for memory shared between Processes.

I-Space needs no writeback, no ambiguity about sharing.

D-Space does require writeable data UNIX does shared data between Parent and Child until one writes data "copy-on-write".]

In all cases: Must take care not to swap out shared memory when one of a number of users is swapped out- (like symbolic linking).

## Instruction Backup:

Many types of Interrupts are processed **between** instructions, exs. I/O Complete, Quanta Complete, but a Page Fault interrupt can occur in the **middle** of an instruction. When returning from an interrupt OS must restart the instruction. So OS needs a way of undoing anything already done by instruction before restart. OS may need hardware assist.

Problem gets worse with pipelining and superscalars.

## Backing Store:

### (1) Swap Area 1-1 with Process

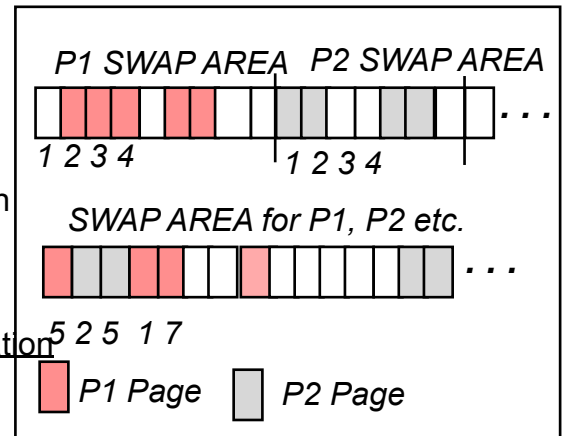
Put Process in when it first enters so each Page has a fixed Disk Address established and

Put it in page by page at the established fixed page location when a page is swapped out.

*Requires knowledge of start location of swap area start only, anti-Pooling*

### (2) Allocate a page at a time as page is brought in, must find a location for a page when it is swapped out

Requires keeping track of location of each page on Backing Store. Kept in Page Map in UNIX. It is possible to handle Text, Stack and Data separately. *Requires knowledge of location of each page as well as available page frames on Disk. Pooling.*

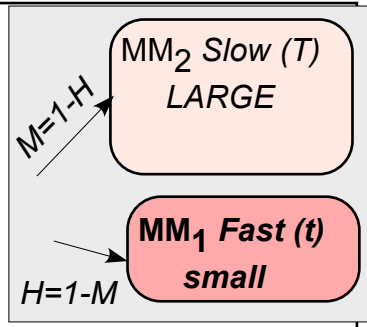


## Locking Pages In Memory

Necessary with DMA- Pages used by DMA might be swapped out or replace before DMA is finished. Information in Page Table. [Like locking parts of files-in in core I-node.]

## MEMORY: OTHER CONSIDERATIONS: PAGE ASSIGNMENTS, PAGE FAULTS.

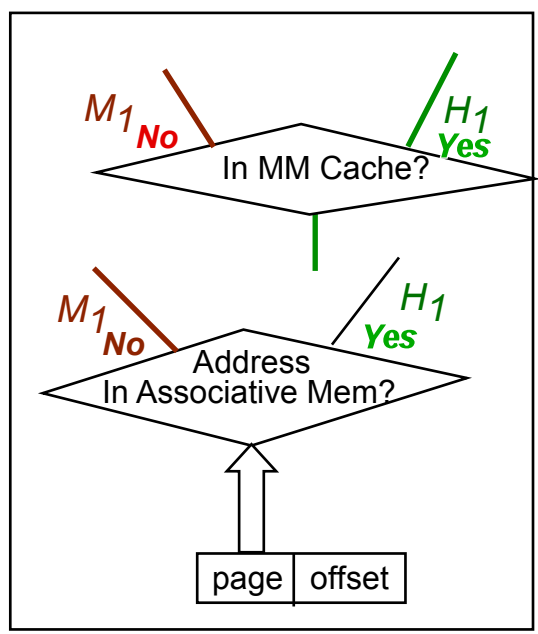
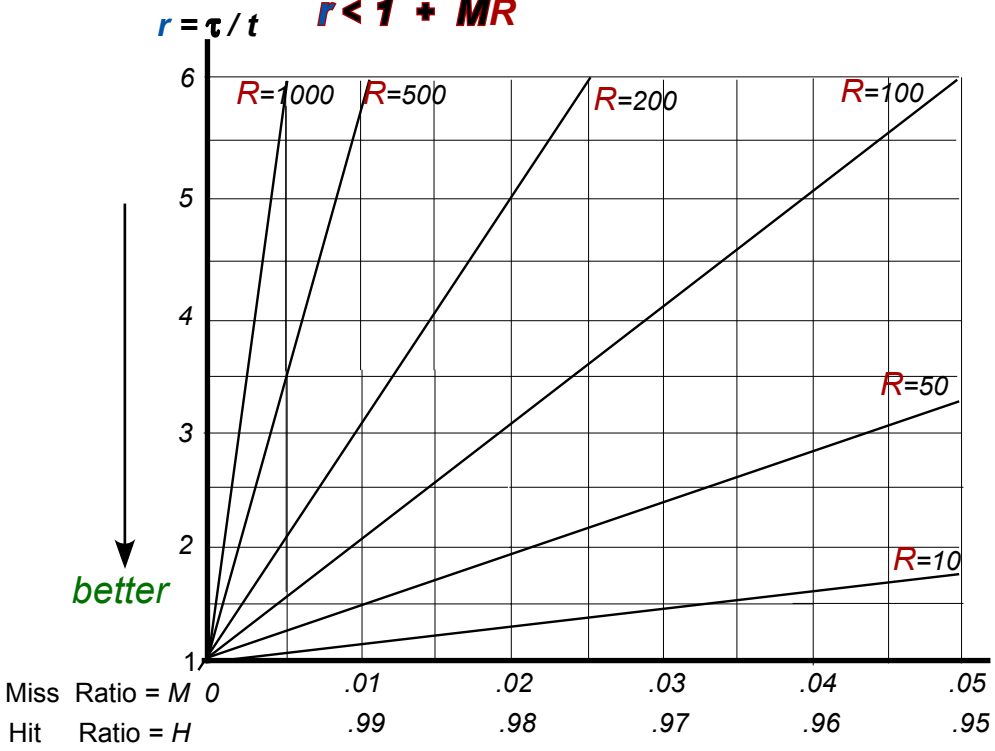
$t$  = Ave time of succesfull MM<sub>1</sub> memory access  
 $T$  = Ave time of access to (MM<sub>1</sub> + that to MM<sub>2</sub>) when the access to M<sub>1</sub> fails and it is necessary to go to MM<sub>2</sub>  
 $\tau$  = Overall average access time ( $> t$ )



$H$  = Hit ratio = fraction of MM<sub>1</sub> memory access which succeed,  
 $M$  = Miss ratio = fraction of MM<sub>1</sub> memory accesses which fail =  $1-H$

$R = T/t$  ratio of the slower to faster access time  
 $r = \tau/t$  ratio of the overall average time to faster access time ( $T$ ) (Closer to 1 the better)  
 Probability of successful MM<sub>1</sub> memory access =  $1-M$

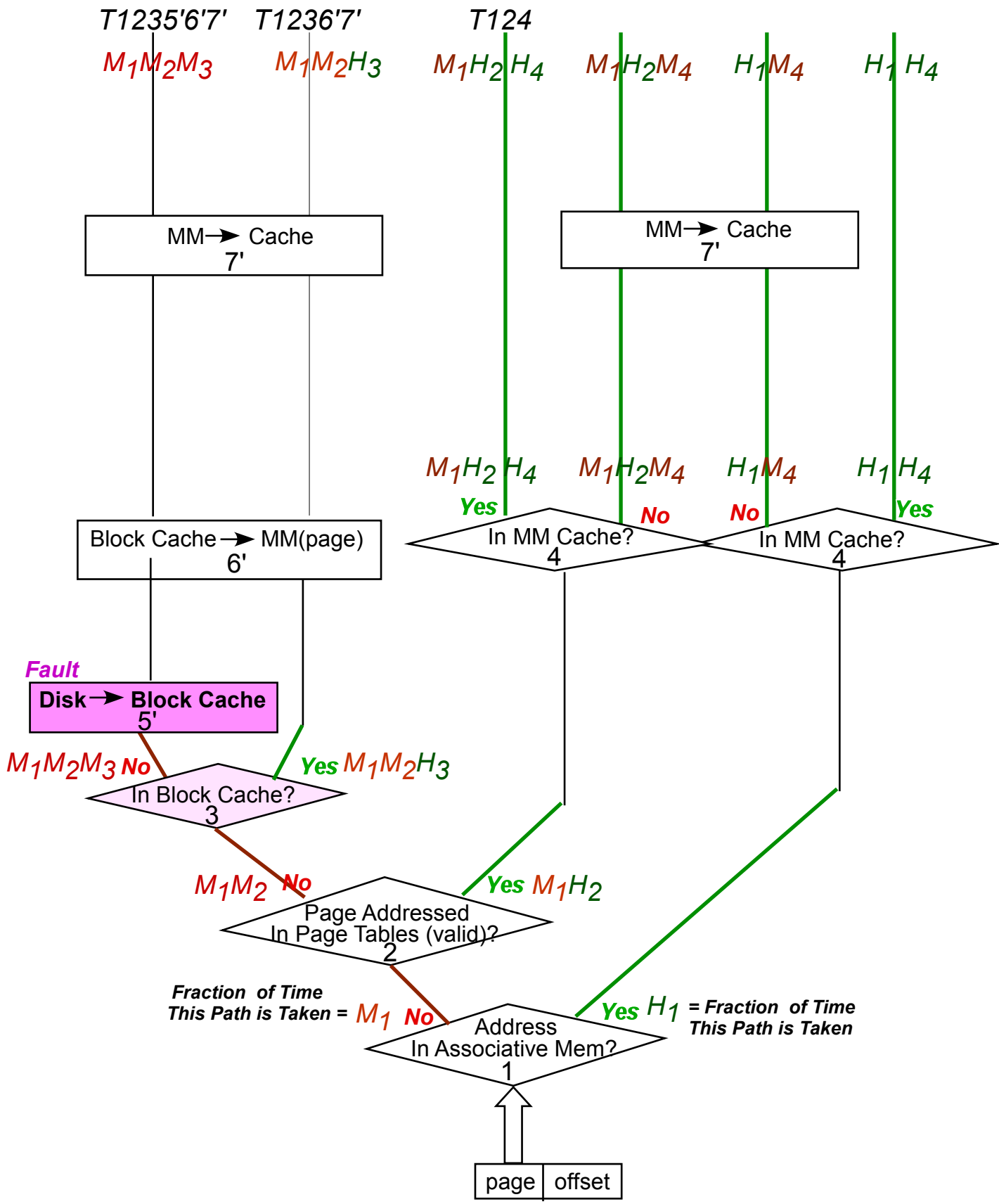
$\tau = (1 - M)t + MT$  -- divide by  $t$ :  $\tau/t = (1 - M) + MT/t$   
 $r = \tau/t$  (closer to 1 the better)  
 $r = (1 - M) + MR$  where  $R = T/t$   
 $r < 1 + MR$



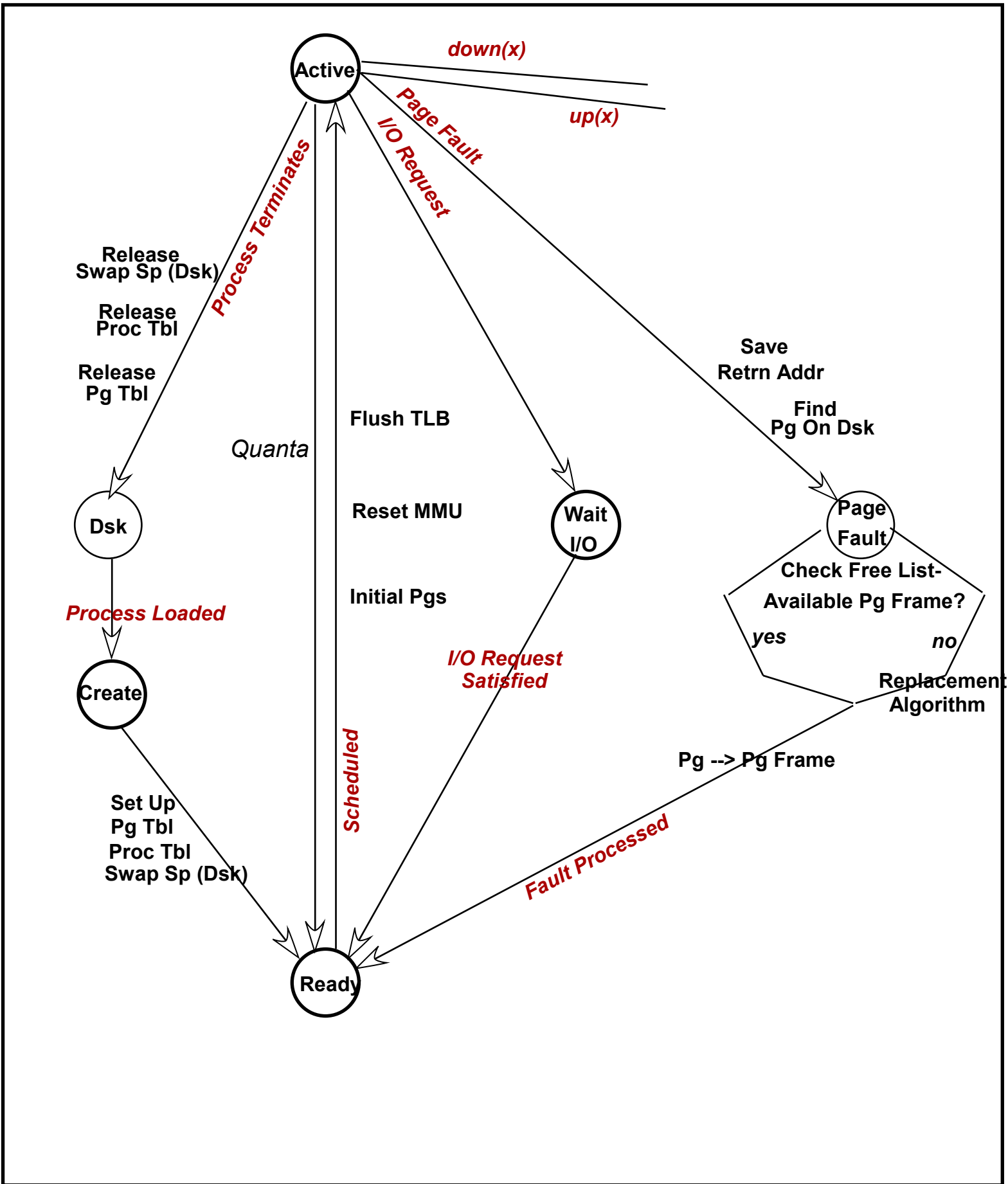
MM <sub>1</sub> ( $t$ )	MM <sub>2</sub> ( $T$ )	Looking For
Associative Memory	Page Table	MM Page Address
Cache2	MM Process Page	Page Addressed
Cache1	Cache2	
MM Process Page	Disk Cache	Page Addressed
Disk Cache	Disk	

(The Larger MM<sub>1</sub> / MM<sub>2</sub> is the higher the Hit Ratio)

**FASTER SMALLER MEMORY SUPPLEMENT SLOWER LARGER ONES TO IMPROVE AVERAGE SPEED PERFORMANCE Performance Analysis**



**THE LONG-LONG TRAILS TRAVELED IN FINDING A PAGE**



**PROCESS STATES WITH PAGING**

This analysis for the average number of probes per entry in T assumes each hash probe is independent and equally likely to find any location in the hash table. The probes are not guaranteed to be to different locations.

Let the **fraction of the table occupied be  $f$**

$S(f,n)$  is meaningful for either the hashing within one table or hashing with chaining.

**Hashing within one table, T:** The average number of probes to find an unoccupied location in T (including the probe that finds it) in T when it is filled to the fraction  $f$  and the table has  $n$  locations.

**Hashing with chaining:** The number of probes that hit existing chains before a new chain is started including the probe that starts the new chain when the the head table, HT, is filled to a fraction  $f$ .

In either case:  $f$  is the probability of failure,  $1-f$  of success of a probe finding an empty location

$$S(f,n) = 1(1-f) + 2f(1-f) + 3f^2(1-f) + \dots + (n-1)f^{n-2}(1-f) + n f^{n-1}(1-f)$$

separating the + and the - terms

$$S(f,n) = 1 - f - 2f^2 - 3f^3 - \dots - (n-1)f^{n-1} - n f^n$$

Therefore adding the sets of terms:

$$S(f,n) = 1 + f + f^2 + f^3 - \dots + f^{n-1} - n f^n \text{ and now adding,}$$

$$-fS(f,n) = -f - f^2 - f^3 - \dots - f^{n-1} - n f^n \text{ gives}$$

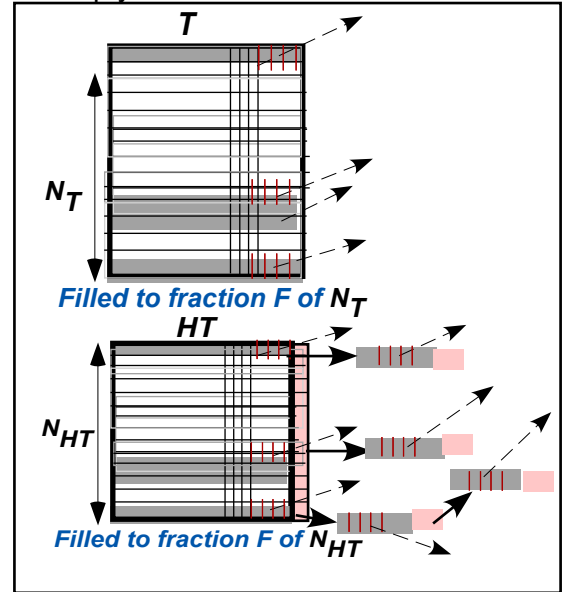
$$(1-f)S(f,n) = \frac{1 - (n+1)f^n + n f^{n+1}}{1-f} = 1 - n f^n - f^n + n f^n f = 1 - f^n - n f^n (1-f)$$

$$S(f,n) = \frac{1 - f^n}{1-f} - n f^n$$

$$S(f,n) = \frac{1 - f^n}{1-f} - n f^n$$

$$S(f,n) = \frac{1 - f^n}{1-f} - n f^n$$

$$S(f) < \frac{1}{1-f} \text{ for all } n, \text{ Then}$$



$S(f)$  = an upper bound on  $S(f,n)$  as  $n$  gets very large (The bound is closer to the actual case for larger size tables and for smaller  $f$ )

Now let  $F$  = fraction of the table T for a) and the fraction of the **head table, HT**, actually occupied in Hashing (respectively with or without Chaining)

$P(F)$  = **Average** number of probes to find an unoccupied location while making entries at all lower fractions in filling the H, and HT to a fraction  $F$ .

$$P(F) < (1/F) \int_0^F [S(f) df]$$

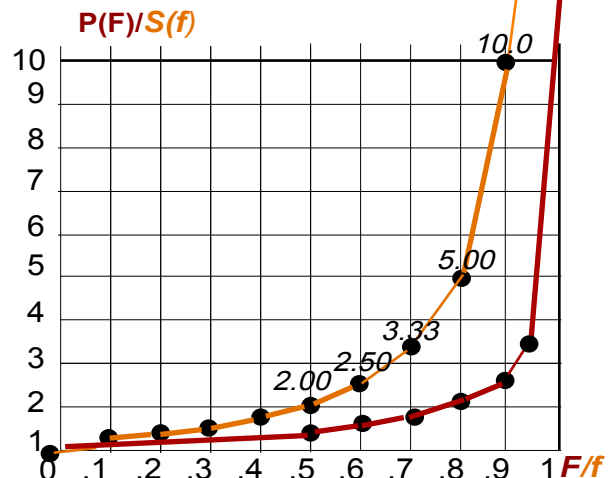
$$P(F) < (1/F) \int_0^F [df/1-f]$$

$$P(F) < -\ln(1-F) / F$$

$F/f$	$P(F)$	$S(f)$
.5	1.39	2.00
.6	1.53	2.50
.7	1.72	3.33
.8	2.01	5.00
.9	2.55	10.0

$$P(F) = -\ln(1-F) / F$$

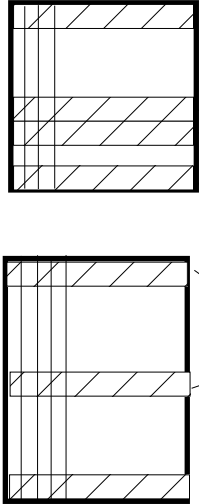
$$S(f) < 1/1-f$$



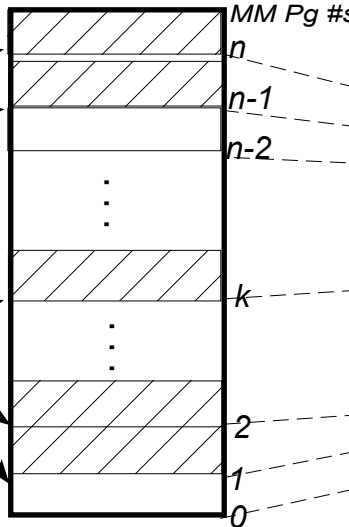
## Analysis Of Hashing-Non-Chaining & Chaining

1. **Daemons:** Associated With Paging-Updating Page Free List

**Page Tables**

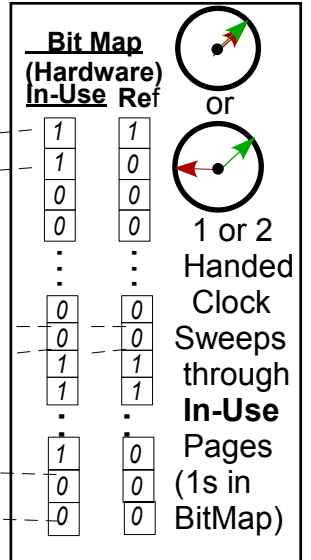


**MM Process MM Pages**

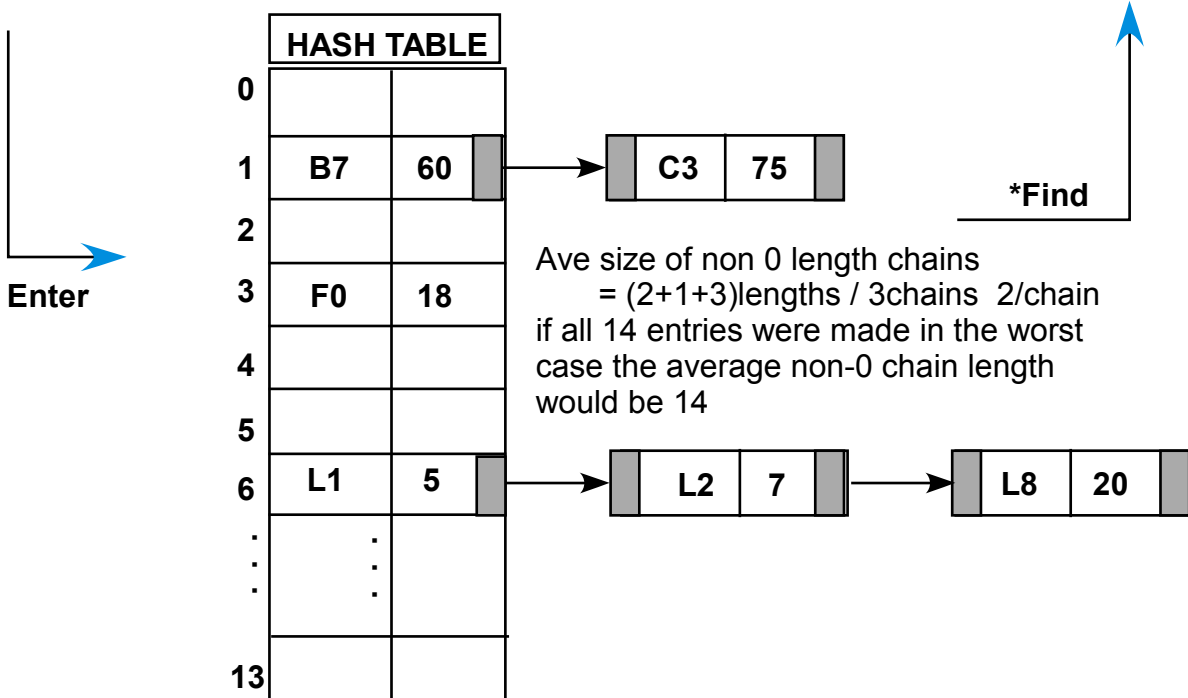


**Page Map**

Swap Lst	In File Dsk (Pg #)	Free
0	sd1 fa	
1	sd2 fb	
0		1
...	...	...
0	fc	1
...	...	...
1	sd3 fy	1
...	...	...
		1



Free  
Free Linked List



Lookup of these 6 entries with equal likelihood requires  $(1 + 2 + 1 + 1 + 2 + 3) / 6 = 1.67$  probes per lookup

