

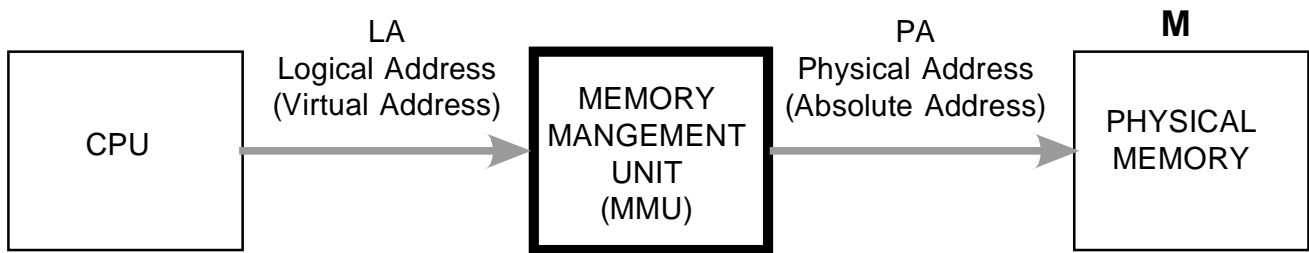
Chapter 4

MEMORY MANAGEMENT

Table of contents:

| | |
|------------------------------|------|
| 4.1 Fixed-Size Partitioning | 4-3 |
| 4.2 Dynamic Partitioning | 4-4 |
| 4.3 Paging | 4-5 |
| 4.4 Multi-Level Paging | 4-11 |
| 4.5 Inverted Page Tables | 4-16 |
| 4.6 Segmentation | 4-18 |
| 4.7 Segmentation with Paging | 4-21 |

The main purpose of memory management is to accommodate several programs (images) in primary memory and to provide adequate translation of logical addresses generated by the CPU into physical addresses of images loaded into the primary memory. This address translation is performed by hardware located between the CPU and the memory.



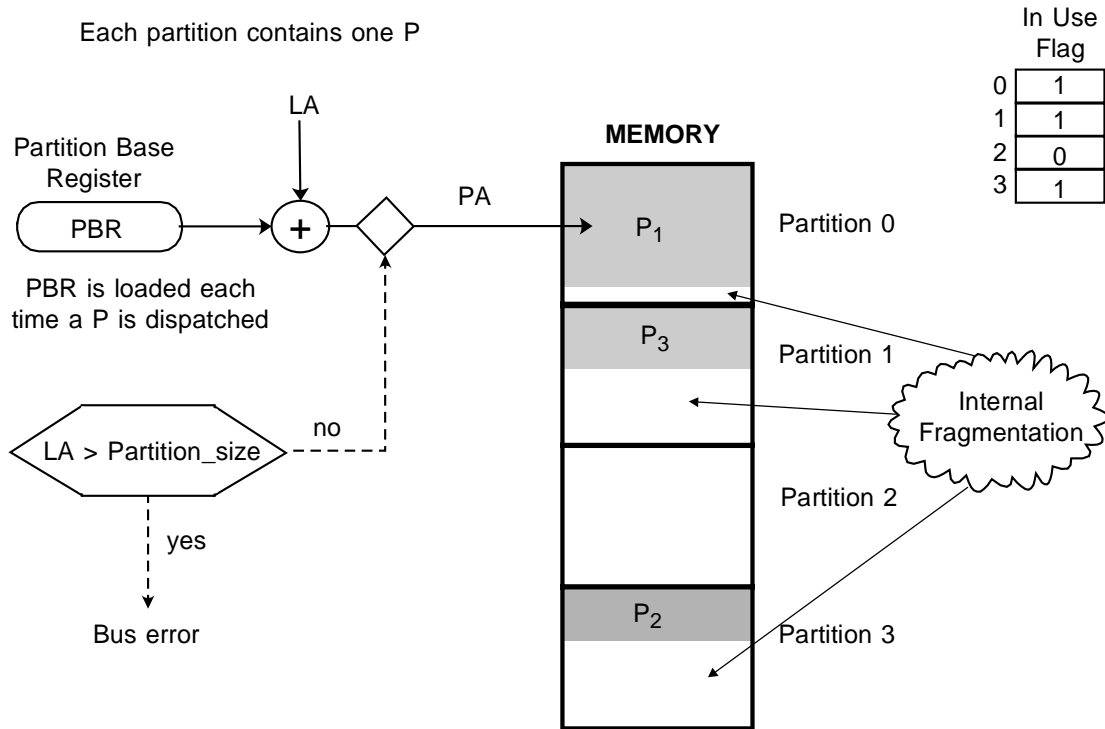
- Fixed-Size Partitioning
 - Dynamic Partitioning
 - Paging
 - Segmentation
 - Segmentation with Paging
- } Translation schemes

LA is generated by the CPU when executing the image
PA is generated by the MMU (user never sees the PA)

FIXED-SIZE PARTITIONING

Equal-Size Partitions
(IBM OS/360)

Each partition contains one P



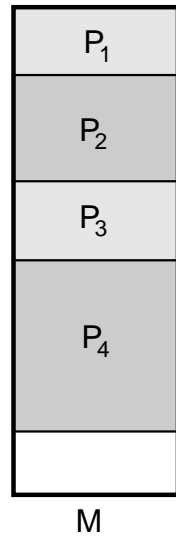
$$PA = (LA > partition_size) ? (Error) : (LA + PBR)$$

Unequal-Size Partitions

Wasted space due to the internal fragmentation can be lessened (but not solved) by using unequal-size partitions (for small and large processes).

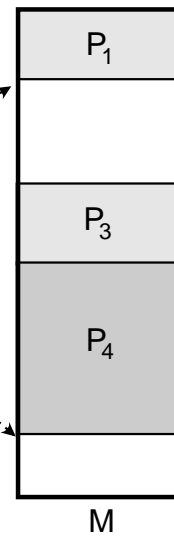
DYNAMIC PARTITIONING

Each P is allocated exactly the required space (no internal fragmentation)

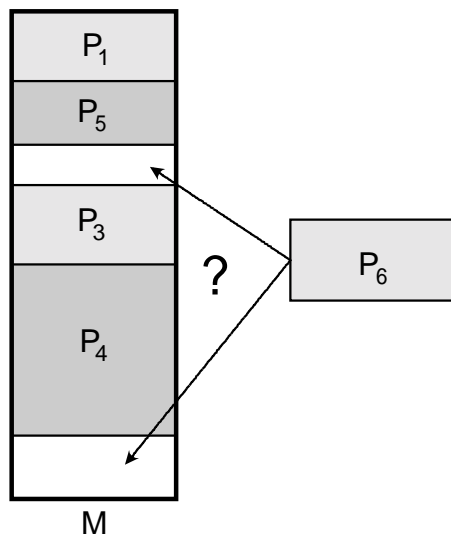


Out-swapping P₂ leaves a hole (External fragmentation)

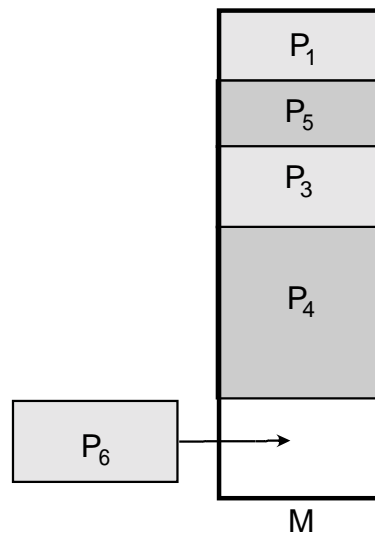
List of free space (holes) is maintained



P₅ is swapped-in. This leaves another hole. After some time there will be many small holes which can't accommodate any P. For example a new process P₆ needs to be allocated.



In such situation the memory compaction is needed. (Compaction consolidates many small holes into a single big hole.)



PAGING

Concept

Each P is divided into equal-sized pages. Memory is also divided into equal-sized frames. (Process pages and memory frames are equal in size.)

This eliminates external fragmentation and reduces internal fragmentation (only the last page will have internal fragmentation, which is smaller than a page.)

Paging is similar to fixed partitioning, only the Ps extend across several frames.

Paging allows that only parts of P must be present in memory, while the other parts can be out-swapped, thus freeing the memory for other Ps.

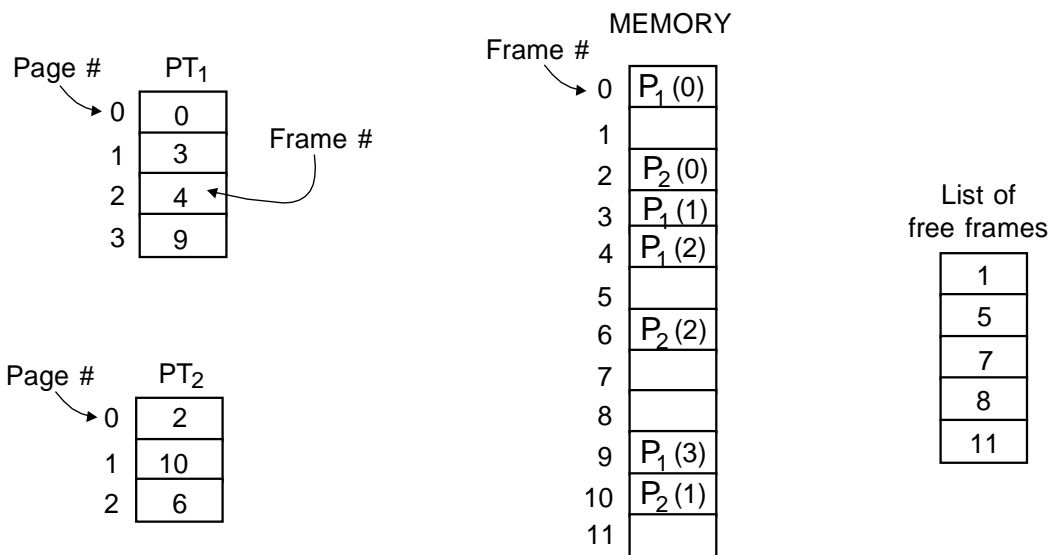
Pages of a P mapped in memory don't have to be consecutive.

Each P has a page table (PT) which maps pages onto memory frames.

A P can access only those memory frames which are listed in its page table.

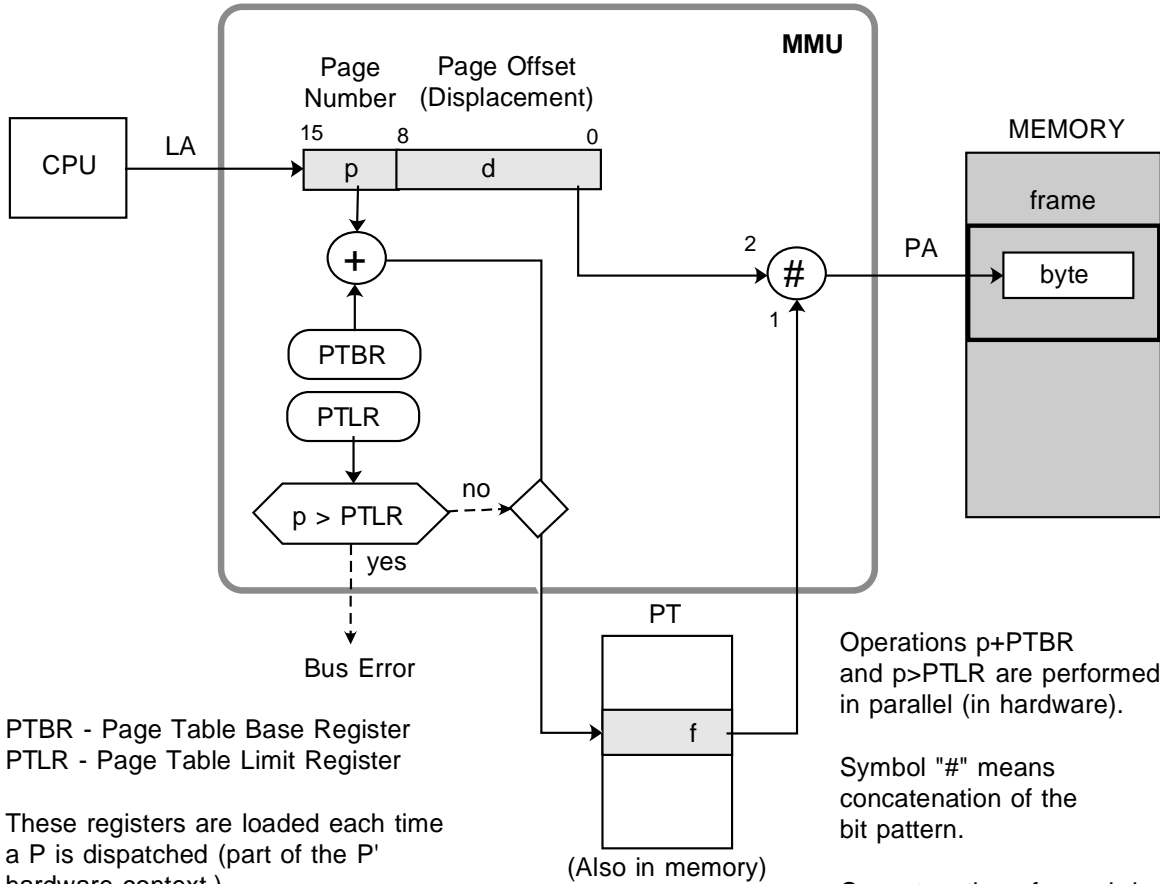
Typical size of frames and pages are $2^9 = 512$ bytes, $2^{10} = 1$ Kb, $2^{11} = 2$ Kb and $2^{12} = 4$ Kb (VAX - 512, NT - 4 Kb) Page size is always power of two!

Small pages decrease internal fragmentation, but introduce overhead in space: page tables become larger. Also, disk I/O is less efficient when transferring smaller blocks of data.



PAGING (Cont.)

Address Translation



PTBR - Page Table Base Register
 PTLR - Page Table Limit Register

These registers are loaded each time a P is dispatched (part of the P' hardware context.)

Operations p+PTBR and p>PTLR are performed in parallel (in hardware).

Symbol "#" means concatenation of the bit pattern.

Concatenation of p and d (or f and) is possible since p is power of two.

Logical Address: LA = <p # d>

Physical Address: PA = $\alpha(p,d) = (p > PTLR) ? (Error) : (<M[p+PTBR].f \# d >)$

P' address space: $\{ \alpha(p,d) \mid 0 \leq p < PTLR ; 0 \leq d < 2^n \}$ (n is number of bits for d)

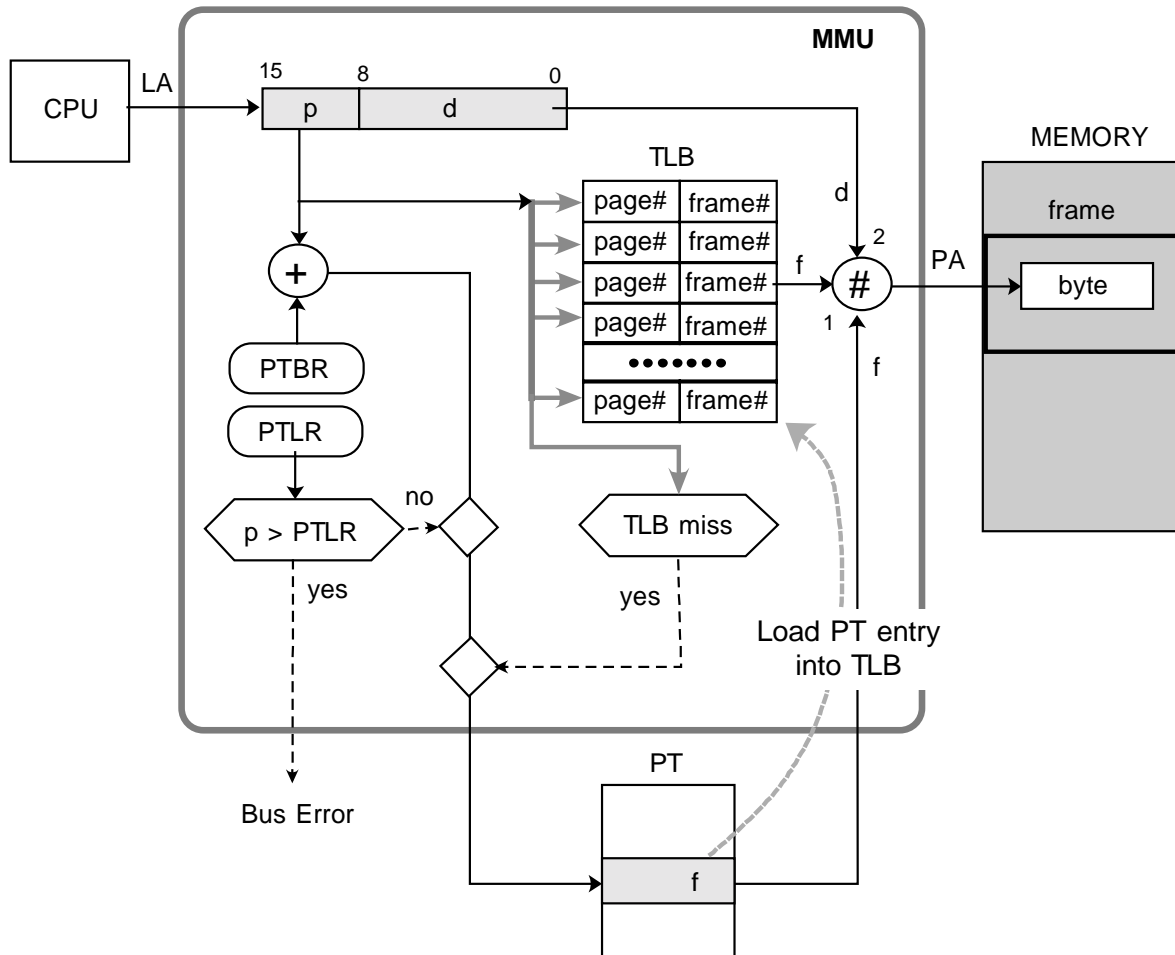
Examples: 16-bit logical address, 512 page size:
 There will be for each P total of $2^{16-9} = 2^7 = 128$ pages,
 (If a PT entry takes 2 bytes, then the PT can fit into a single page)
 Each P has maximal address space of $128 \times 512 = 2^{16} = 64$ K words

32-bit logical addresses, 4 K page size:
 There will be for each P total of $2^{32-12} = 2^{20} = 1$ M pages,
 (If a PT entry takes 4 bytes, then the PT would require $4 \times 2^{20}/2^{12} = 2^{10} = 1024$ pages, i.e. the PT has to be paged too.)
 Each P has maximal address space of $2^{20} \times 2^{12} = 2^{32} = 4$ G words

PAGING (Cont.)

Translation Look-aside Buffer

Address translation via page table (which is a data structure residing in memory) requires one additional memory access per memory reference. In order to speed up the translation the Translation Look-Aside Buffers (TLB) are used as a part of MMU.



TLB is an associative memory which performs simultaneous check of all keys (page numbers) and finds the corresponding frame number in one cycle. TLB contains the most recent entries of the page table (it typically has 8 to 2048 entries.)

TLB is part of the P' hardware context - it has to be flushed and reloaded after each P dispatch.

If the page is not found in TLB (TLB miss) then the memory resident PT is used and the entry from the PT is inserted into TLB.

PAGING (Cont.)

With a reasonable size of TLB a relatively high hit-ratio can be achieved (Intel 80486 has 32 associative registers giving an average hit-ratio of 98%)

Time overhead:

$$\begin{aligned} t_{PA} &= h t_{TLB} + (1-h) (t_M + t_{TLB}) \\ &= t_{TLB} + (1-h) t_M \\ &= t_M (\mu + (1-h)) \end{aligned}$$

where:

$$\begin{aligned} t_{PA} &= \text{time per address translation} \\ t_{TLB} &= \text{TLB access time} \\ t_M &= \text{memory access time} \\ h &= \text{TLB hit-ratio} \\ \mu &= \text{TLB efficiency } (\mu = t_{TLB}/t_M) \end{aligned}$$

Example:

Suppose that TLB takes 20% of a memory access ($\mu = 0.2$) then:

$$t_{PA} = 0.40 t_M \quad \text{for } h = 80\%$$

$$t_{PA} = 0.22 t_M \quad \text{for } h = 98\%$$

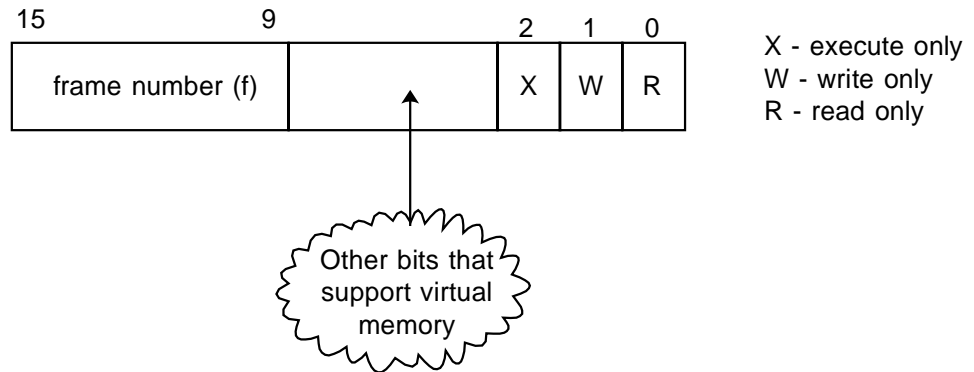
PAGING (Cont.)

Protection

The most important aspect of protection is that one P can't see the address space of another P. This is ensured by the fact that each P has its own page table. A P can access its PT only via PTBR which is a part of the P's hardware context. In addition, the register PTLR is also used to prevent the running P to access invalid PT entries.

In addition to this, different pages can be given different access rights. This information is added to PT entries.

Page Table Entry (example for 16-bit addressing):



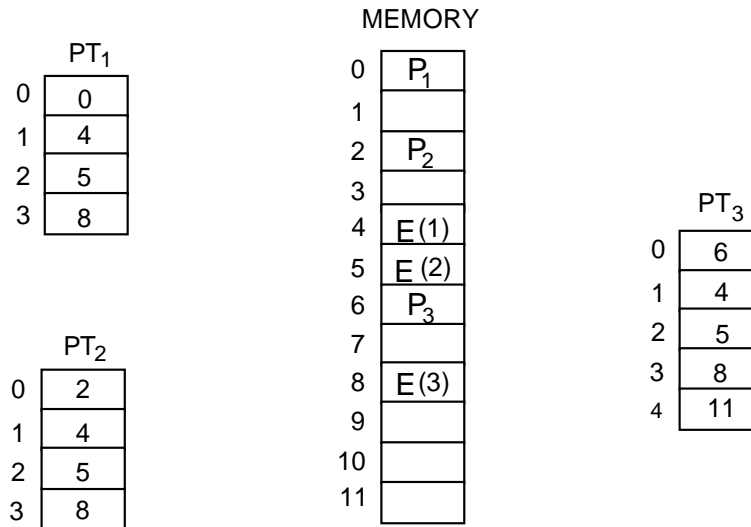
PAGING (Cont.)

Memory Sharing

Paging is convenient for memory sharing - several P's can be allowed to point to the same frames in memory.

Example:

Two processes are using the same text editor



E(1) E(2) E(3) - Editor pages

P₁ P₂ P₃ - Process code including private editor data instances

NOTICE: Shared programs must be reentrant, i.e. their code must not be modifiable by any P. Way to do it is to maintain separate instances of the shareable program's global data for each P that uses the shareable program.

Example:

Suppose there are 40 users who use the same text editor with the size 200 Kb. Private data take 50 Kb. The total memory used in the nonsharing case would be:

$$(200 + 50) \times 40 = 10,000 \text{ Kb} = 9.8 \text{ Mbytes.}$$

In sharing case, the used memory space is:

$$50 \times 40 + 200 = 2200 \text{ Kb} = 2.1 \text{ Mbytes.}$$

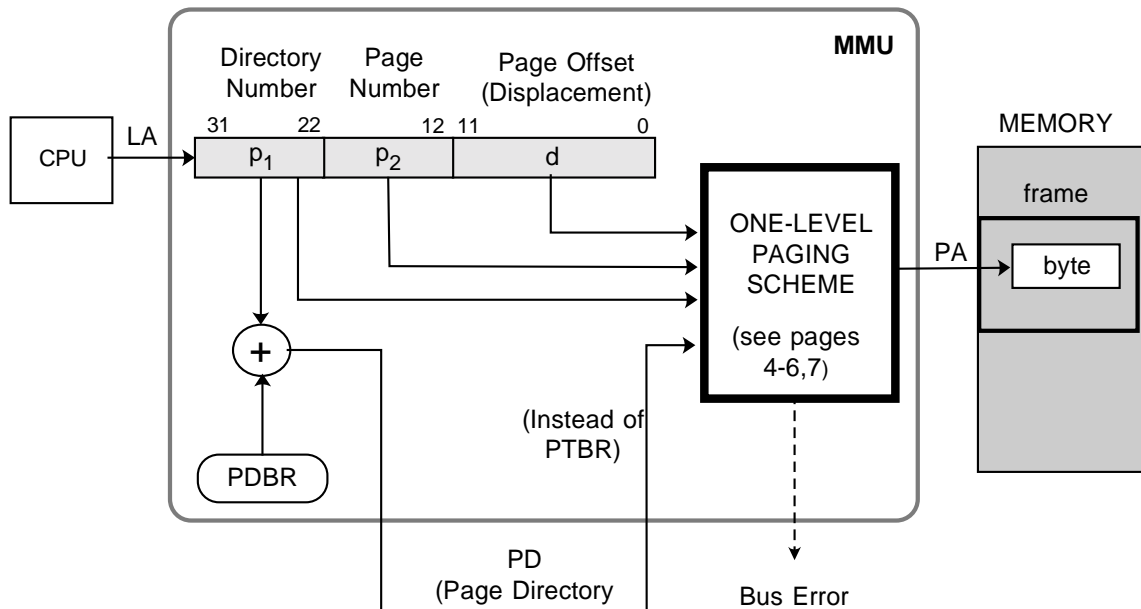
In real life (systems with GUI), the sharable libraries could take megabytes of memory.

MULTI-LEVEL PAGING

In systems with larger address space the one-level paging results in too long page tables which exceed the size of a page. Consequently they have to be paged as well as the program code and data. For example the 32-bit addressing system with 4 Kbyte pages would have page tables with max 1 M entries. If each entry takes 4 bytes, the PT becomes 4 Mbytes long, or $4 \times 2^{20} / (2^{12}) = 1024$ pages. Therefore a two-level paging scheme is needed.

The two-level paging is used in Windows NT on Intel 80386, 80486, Pentium and MIPS R4000.

Two-Level Paging



PDBR - Page Directory Base Register

This register is loaded each time a P is dispatched (part of the P's hardware context.)

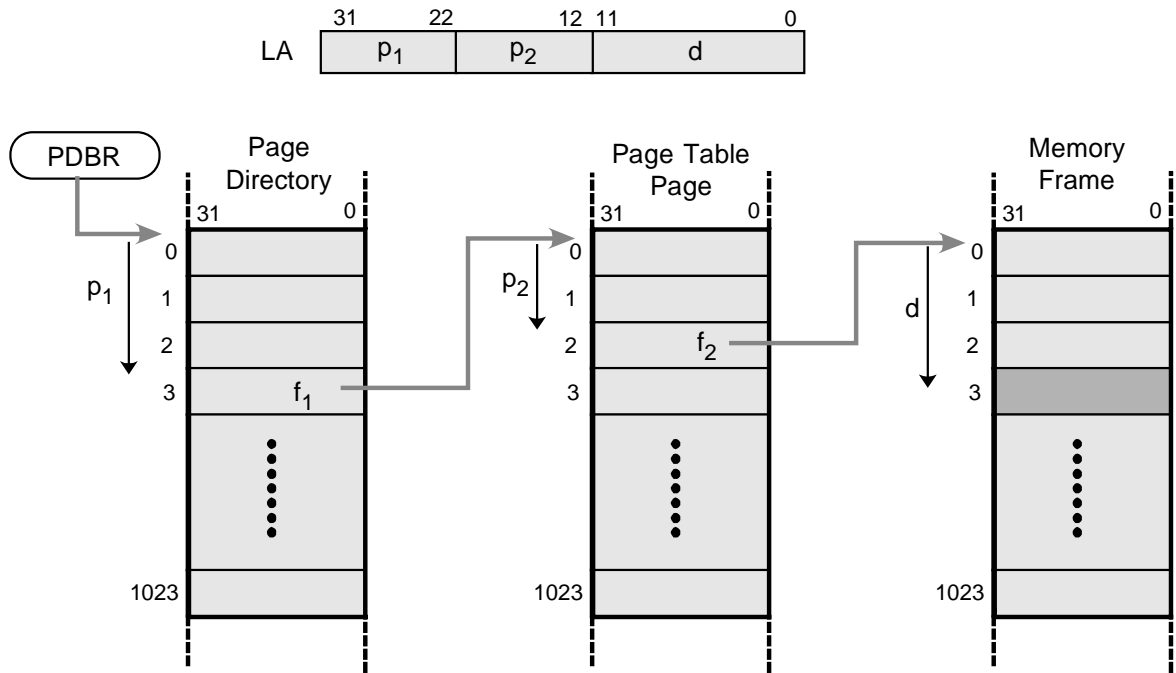
The most recent PT entries (second level) can be kept in TLB)

The page directory has 2^{10} entries = 4 KB and requires one page. Each page directory entry (size 4 bytes) has a frame number (f_1) whose maximal value is $2^{32-12} = 2^{20} = 1$ M (total number of frames in memory)..

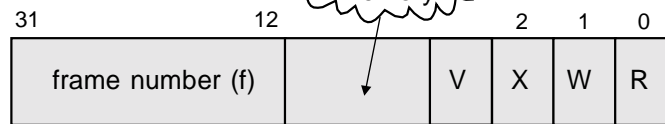
The page table (second level) has 1 M entries, i.e. $4 \times 2^{20} / 2^{12} = 1024$ pages - each page is pointed to by an entry from the PD.

MULTI-LEVEL PAGING (Cont.)

Address Space of Two-Level Paging Scheme



Page Table Entry:



X - execute only
 W - write only
 R - read only
 V - Valid/invalid bit

V=1, the page is valid, i.e. it is a part of the **P**'s address space; V=0 - the page is invalid, i.e. it is not a part of the **P**'s address space.

Logical Address: LA = <p₁ # p₂ # d>

Physical Address: PA = α(p₁,p₂,d) = <f₂ # d>

f₁ = (M[PDBR+p₁].V = 0) ? (Error) : (M[PDBR+p₁].f)

f₂ = (M[f₁+p₂].V = 0) ? (Error) : (M[f₁+p₂].f)

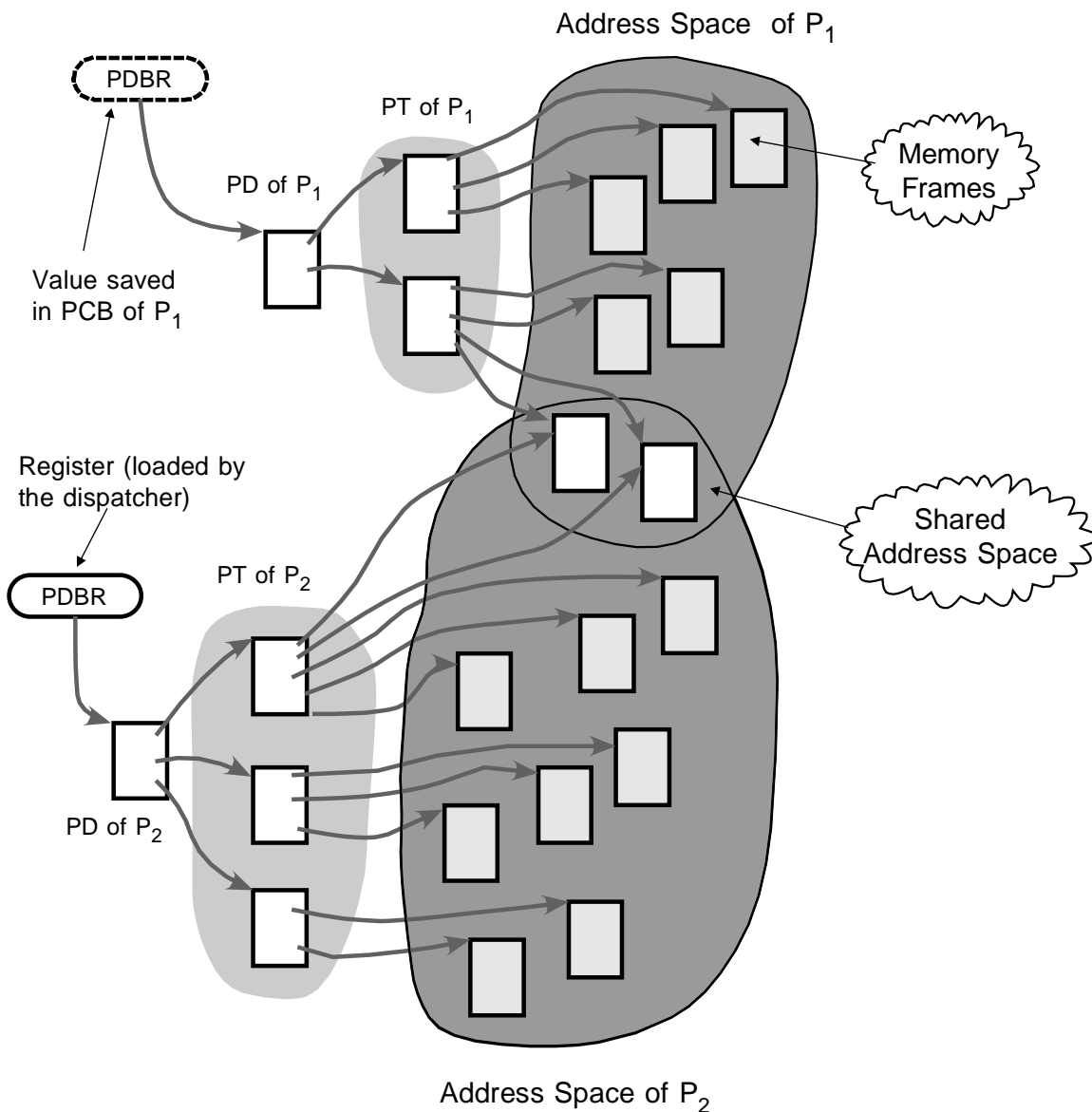
Address Space: { α(p₁,p₂,d) | M[PDBR+p₁].V = 1, M[f₁+p₂].V = 1 }

Time Overhead: t_{PA} = t_M (μ + 2(1-h)); μ = 20%, h=98%, t_{PA} = 0.25 t_M

MULTI-LEVEL PAGING (Cont.)

Example of Two Processes

Suppose there are two processes P_1 and P_2 , the latter has a running thread, while the threads of the former process are in ready state. Also suppose that both processes share two pages.

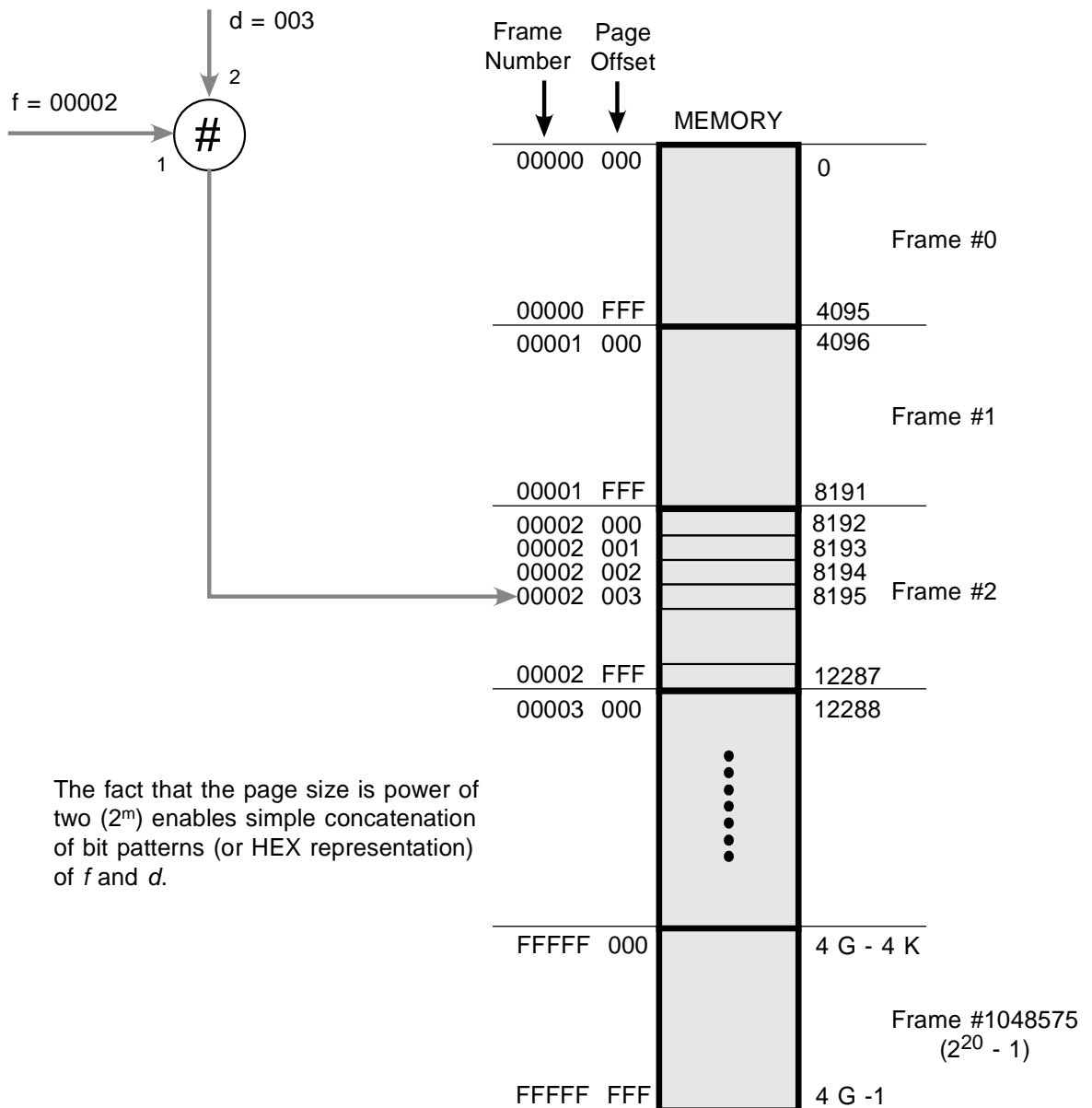


MULTI-LEVEL PAGING (Cont.)

How Physical Address Works

Suppose we know the frame number (e.g. $f = 2$) and the page offset (e.g. $d = 3$)
 The corresponding physical address is:

$$PA = \langle f \# d \rangle = (00002 \# 003)_{16} = (00002003)_{16} = (8195)_{10}$$

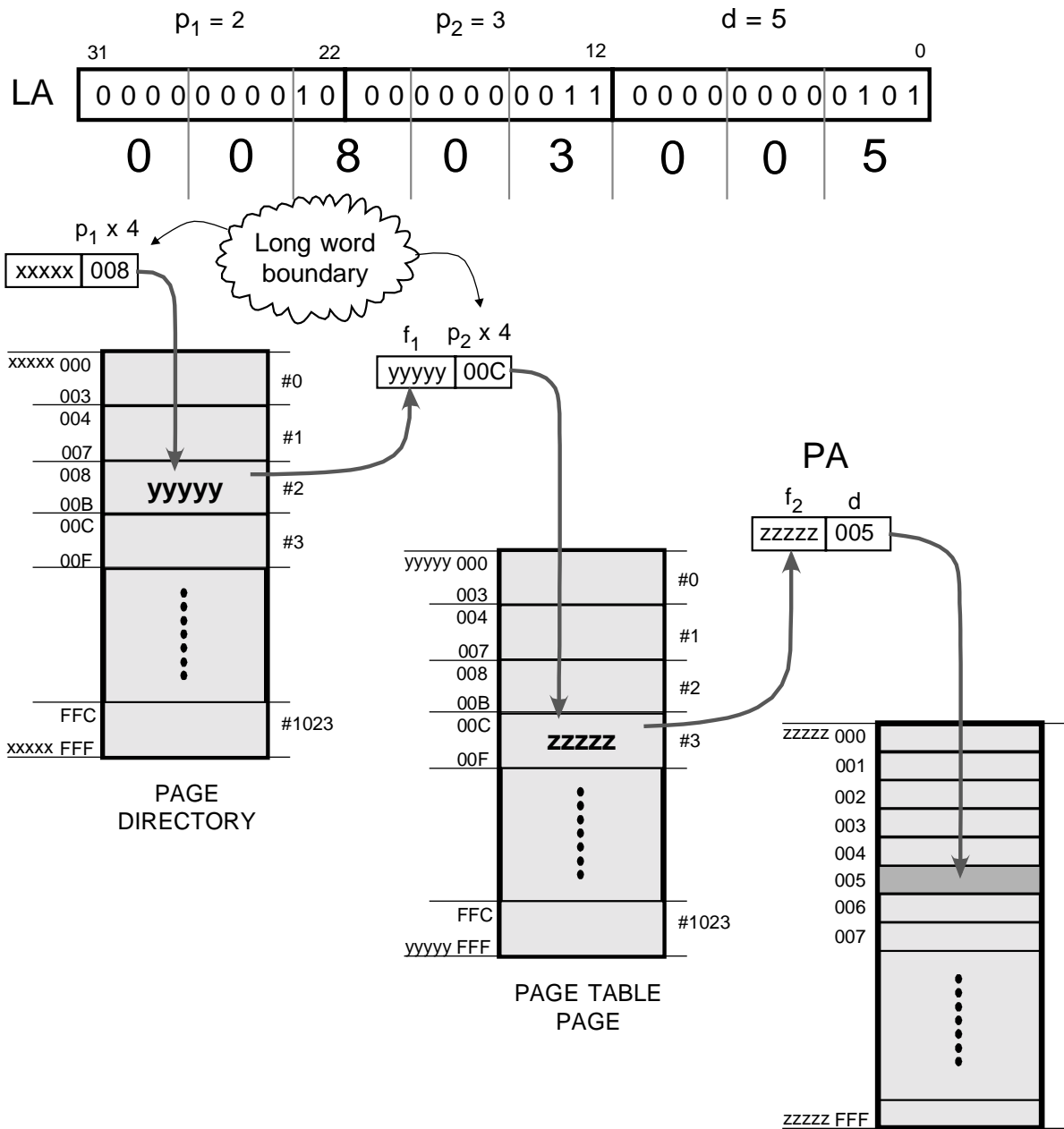


The fact that the page size is power of two (2^m) enables simple concatenation of bit patterns (or HEX representation) of f and d .

MULTI-LEVEL PAGING (Cont.)

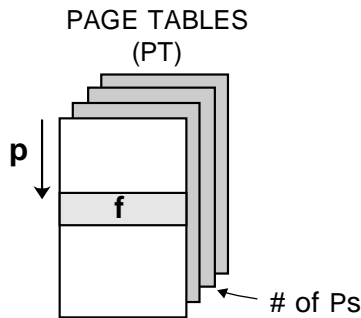
How Address Translation Works

Suppose the logical address is: $LA = (8400901)_{10} = (803005)_{16}$
 and suppose that the page directory for the running P resides at $PDBR = \text{xxxxx000}$

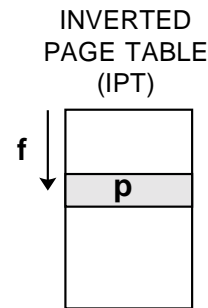


INVERTED PAGE TABLE

Basic Idea

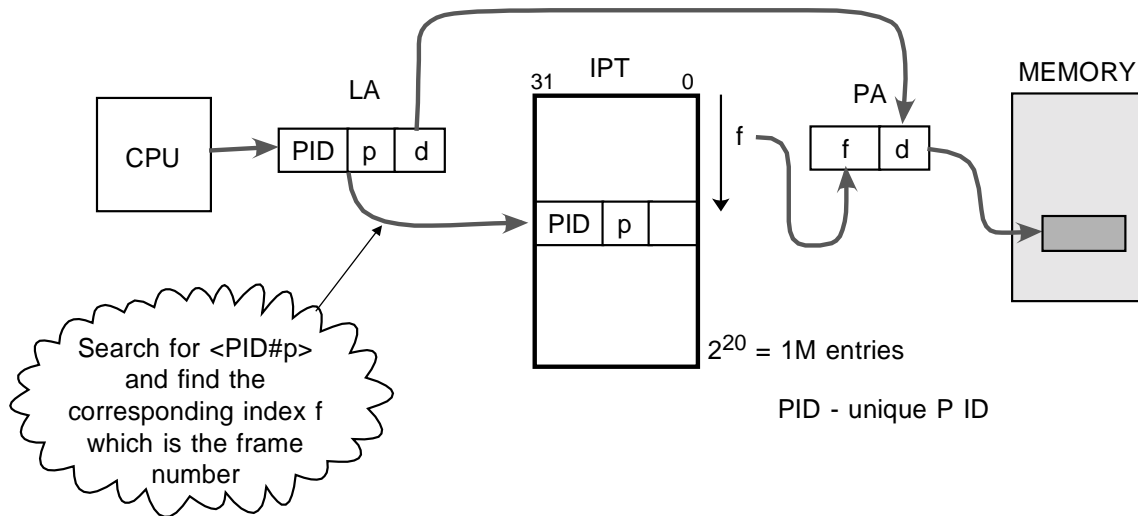


Each P has a PT
(There are as many address spaces as Ps - consumes a large amount of memory)



32-bit address system has 1 M frames, 26-bit address system has 16 K frames

There is only one IPT per system
(The number of entries equal to the number of frames)



Used on IBM AS/400, IBM RISC System6000, IBM RT, PowerPC, HP Spectrum w/s.

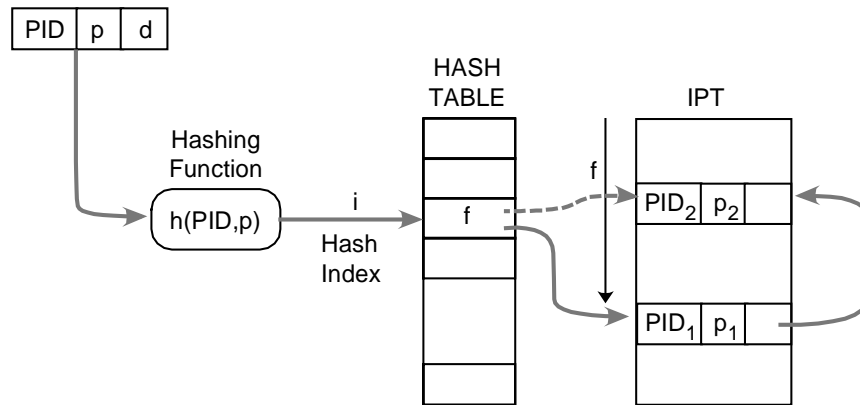
Space efficient but there is an additional time overhead for key-search (this problem can be lessened by using hashing - see next page.)

Memory sharing is more complicated than with PT: sharing means that several Ps point to the same frame, which would require doubling, tripling, ... the IPT entries.

INVERTED PAGE TABLE (Cont.)

Hashing

Problem of mapping of a larger range of numbers $\langle \text{PID}, p \rangle$ into a smaller range $\langle f \rangle$.
 In such situation is used hashing approach, based on hashing function and hash table.



Placing a new page (given $\langle \text{PID}, p \rangle$ and f , find links):

- OS assigns a frame number (f) to a page $\langle \text{PID}_2, p_2 \rangle$
- The hash index is computed: $i = h(\text{PID}_2, p_2)$
- If entry i in hash table is unused, f is copied into it (dashed arrow) - done.
- If entry is already used (by PID_1, p_1), then a link is created to $\langle \text{PID}_2, p_2 \rangle$.

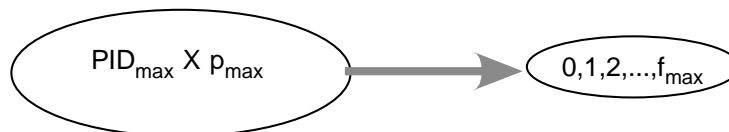
Searching for f (given $\langle \text{PID}, p \rangle$, find f):

- The hash index is computed: $i = h(\text{PID}_2, p_2)$
- Entry is checked in IPT at f : if the entry has (PID_2, p_2) - done,
- If not, the search continues following the link.

In most cases mapping requires only two memory references (when hashing function gives unique value.)

Popular hashing function: divide $\langle \text{PID}, p \rangle$ by $f_{\max} + 1 (= 2^{20})$ and take the remainder as the value of the hash index (pseudo-randomization).

What is hashing? Mapping of a larger set of integers (like $\langle \text{PID}, p \rangle$) into a smaller set (like $\langle f \rangle$)



SEGMENTATION

In paging approach the memory is viewed somehow as a linear structure. This is not how the programmer sees his/her program, which rather consists of several unordered and unequal sized modules, which could be residing in different segments of memory.

Division of a program into collection of segments is normally done automatically by the compiler. In assembly language user however can specify the segments explicitly.

Each segment has a name and the length and the memory appears to be two-dimensional:

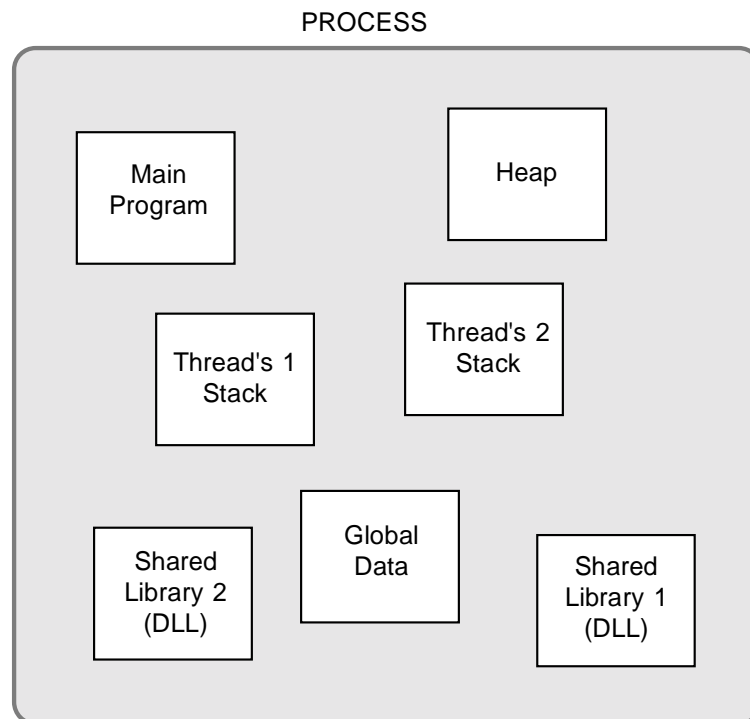
$$LA = \langle \text{segment_name}, \text{offset} \rangle$$

In order to simplify algorithms, the segment names are replaced by segment numbers:

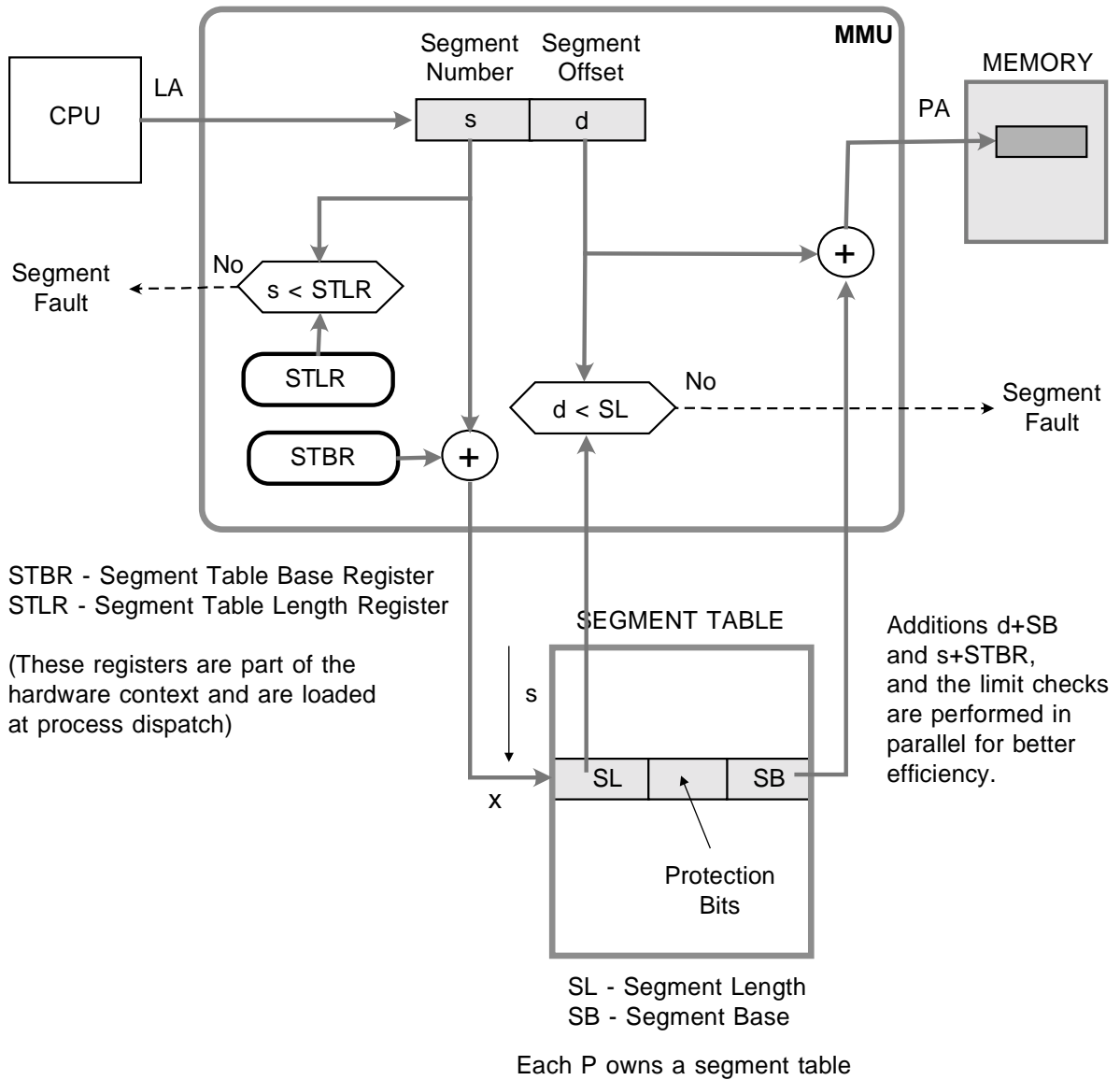
$$LA = \langle \text{segment_number}, \text{segment_offset} \rangle$$

NOTICE: The segment number and the offset are no longer concatenated, as in case of the page number and page offset (therefore the operator "," is used instead of "#")

Segmentation is similar to dynamic partitioning, only this time process can have several segments, while in dynamic partitioning a partition is accommodating the entire process.



SEGMENTATION (Cont.)



Logical Address: $LA = \langle s,d \rangle$

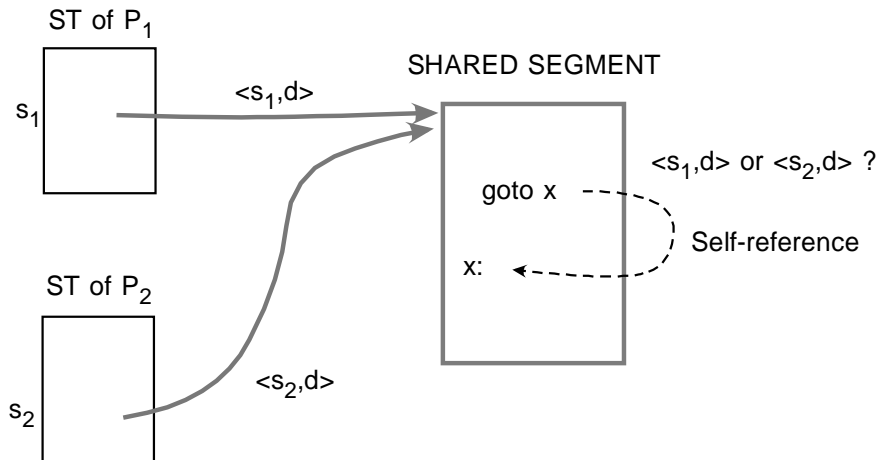
Physical Address: $PA = \alpha(s,d) = (d < M[x].SL) ? (M[x].SB+d) : (Error)$
 $x = (s < STLR) ? (STBR + s) : (Error)$

SEGMENTATION (Cont.)

Protection and Sharing

Protection and sharing is more appropriate with segmentation than with paging, because it is more natural to define shared parts and the access rights uniformly at the segment level (pages have no logical meaning from the programmer's point of view.)

There is however a little problem with sharing in segmented environment. Suppose a shared code which makes a self-reference. The segmentation algorithms must ensure that such shared code must have the same segment number for all processes that participate in the sharing.



Segments that make self-references have to have a unique segment number for each process.

If the segment doesn't use self-references, the addresses can be resolved by incrementing the PC by some offset (next instruction, short jump, PC-relative, etc.). Therefore the segment numbers can be arbitrary.

A drawback of segmentation is the storage allocation (this is always a problem whenever the chunks of memory to be allocated are of different size.) Consequently, there is an external fragmentation, which can be solved by compaction. Compaction is easy to do, due to the run-time loading.

Another approach to solve the allocation problem is to combine segmentation with paging.

SEGMENTATION WITH PAGING

Some modern processors allow usage of both, segmentation and paging alone or in a combination (Motorola 8030 and later, Intel 80386, 80486, Pentium) - the OS designers have a choice.

| Segmentation | Paging | |
|--------------|--------|--|
| No | No | Small (embedded) systems low overhead, high performance |
| No | Yes | Linear address space BSD UNIX, Windows NT |
| Yes | No | Better controlled protection and sharing. ST can be kept on chip - predictable access times (Intel 8086) |
| Yes | Yes | Controlled protection/sharing better memory management. UNIX Sys. V, OS/2. |

SEGMENTATION WITH PAGING (Cont.)

Intel 80386, 486 and Pentium support the following MM scheme which is used in IBM OS/2

