

بسم الله الرحمن الرحيم

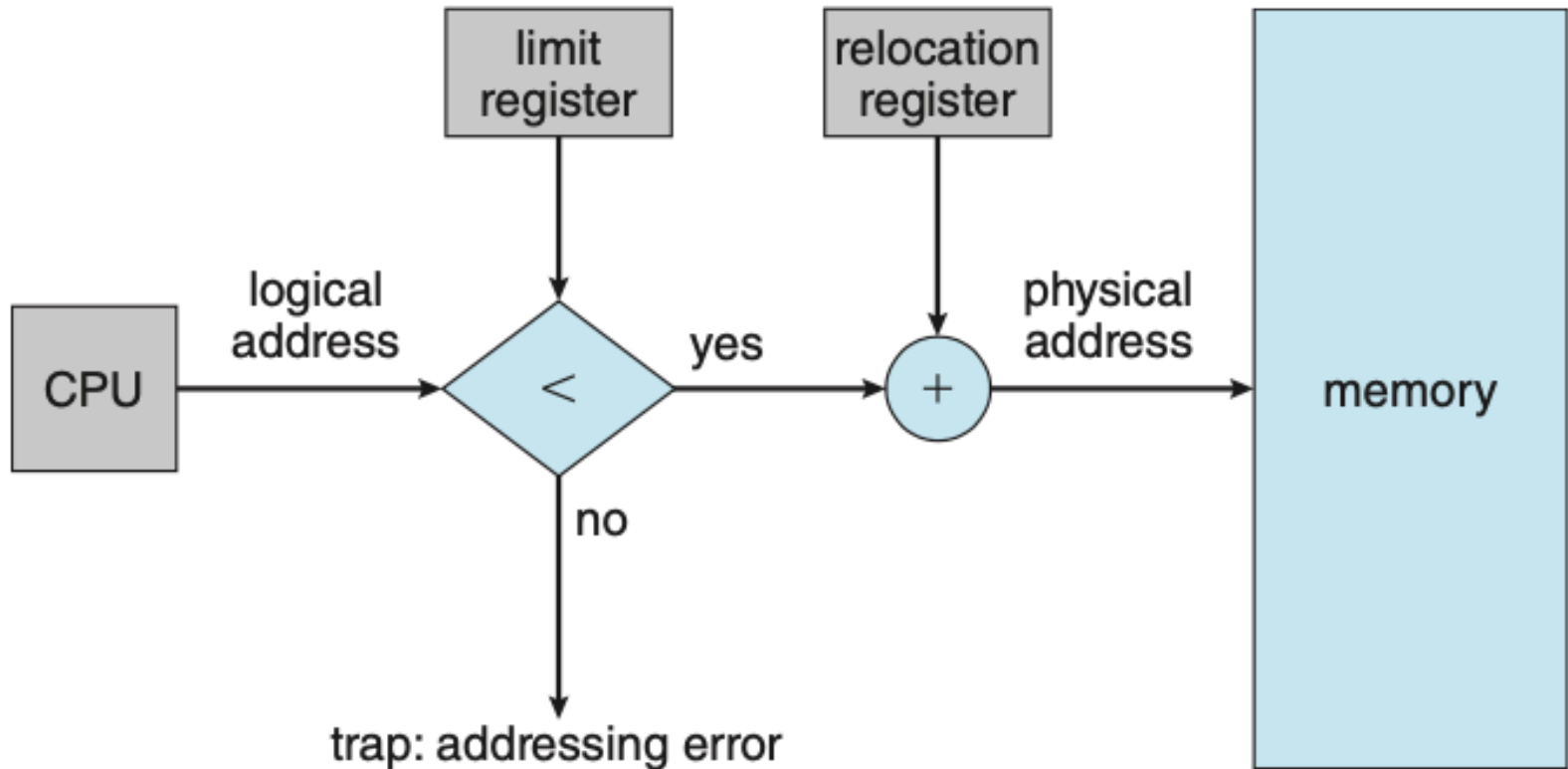
«سیستم عامل»

۹۹

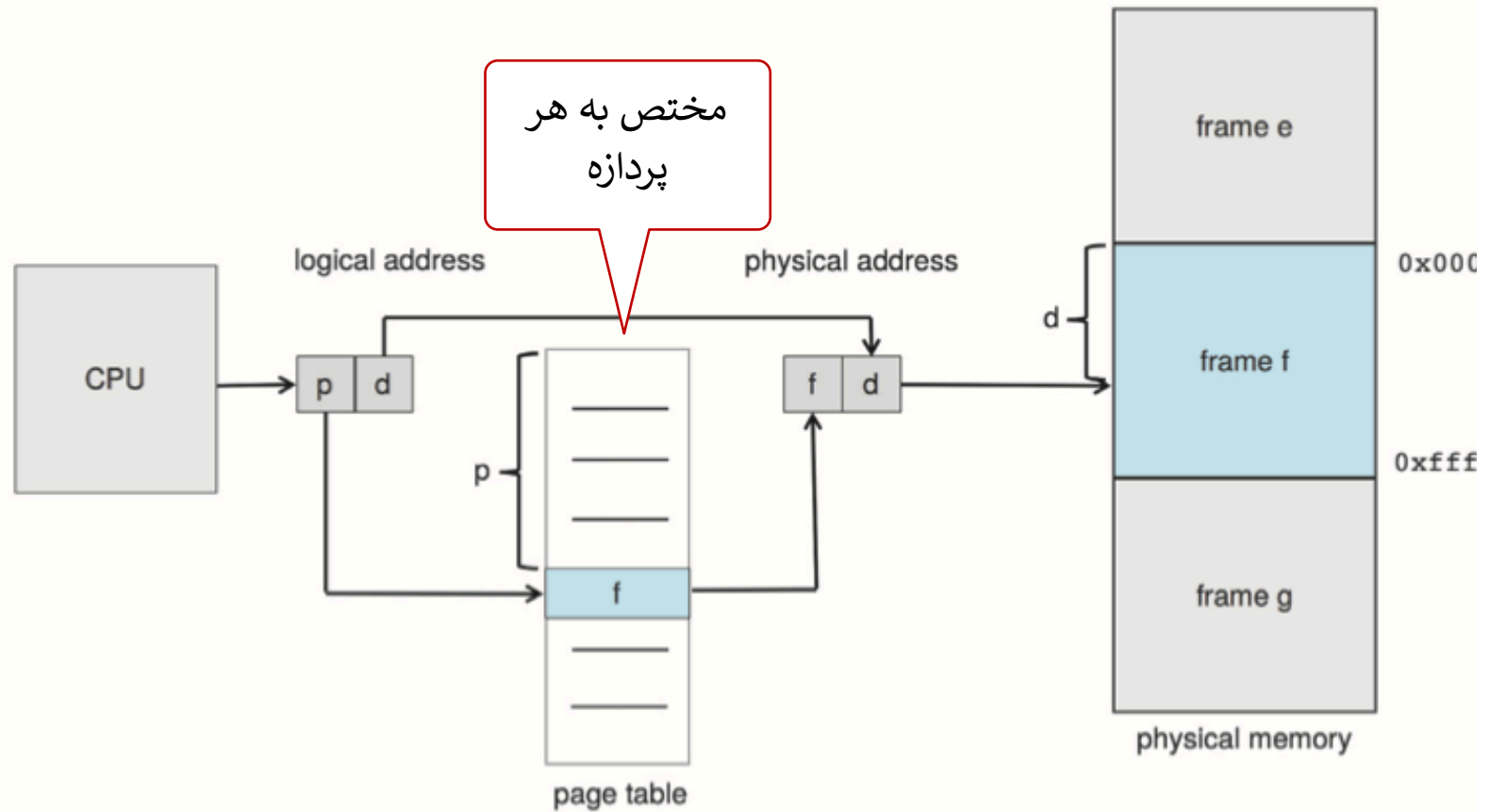
جلسه ۱۶: مدیریت حافظه (۴)

مرور - حافظه متوالی (Contiguous)

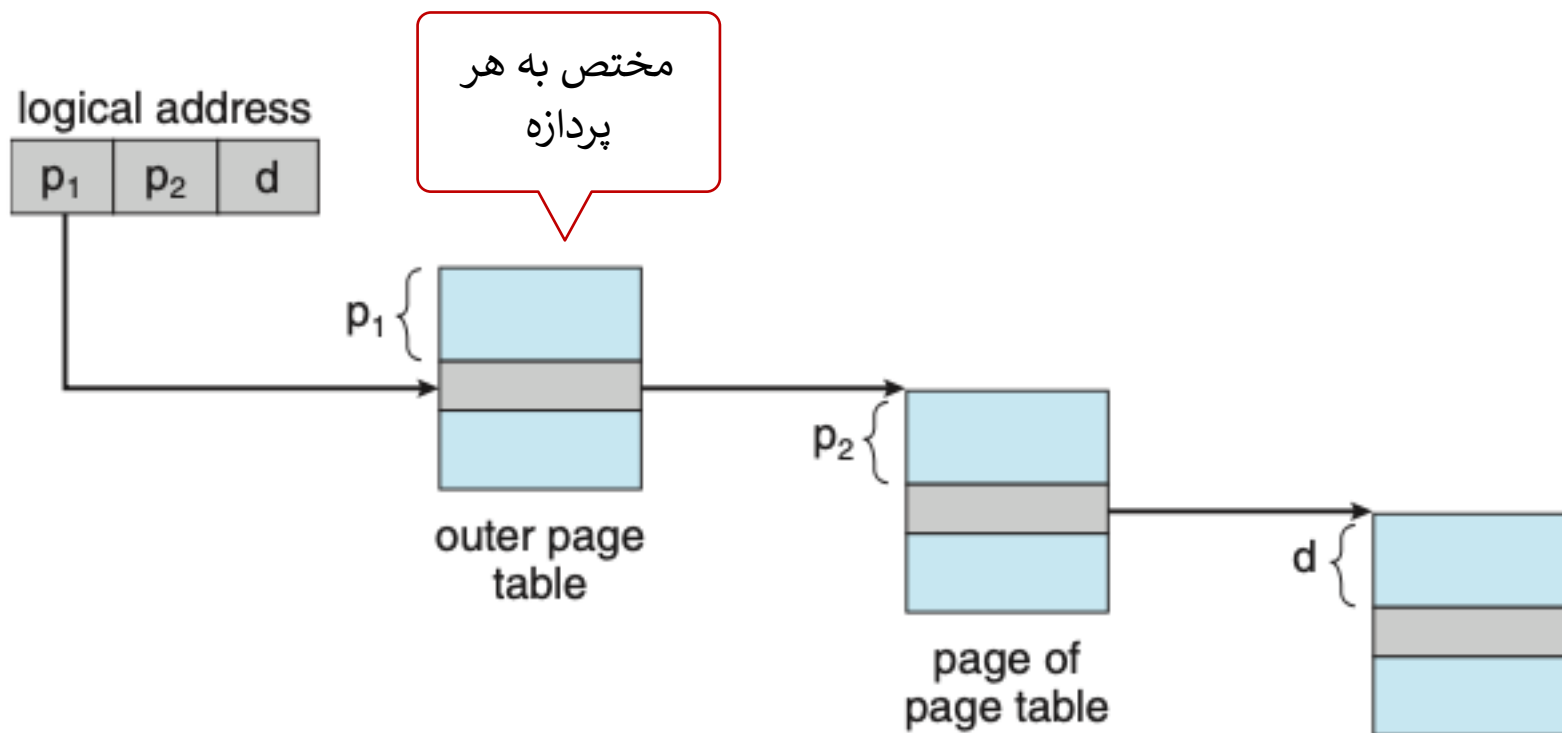
مختص به هر
پردازه



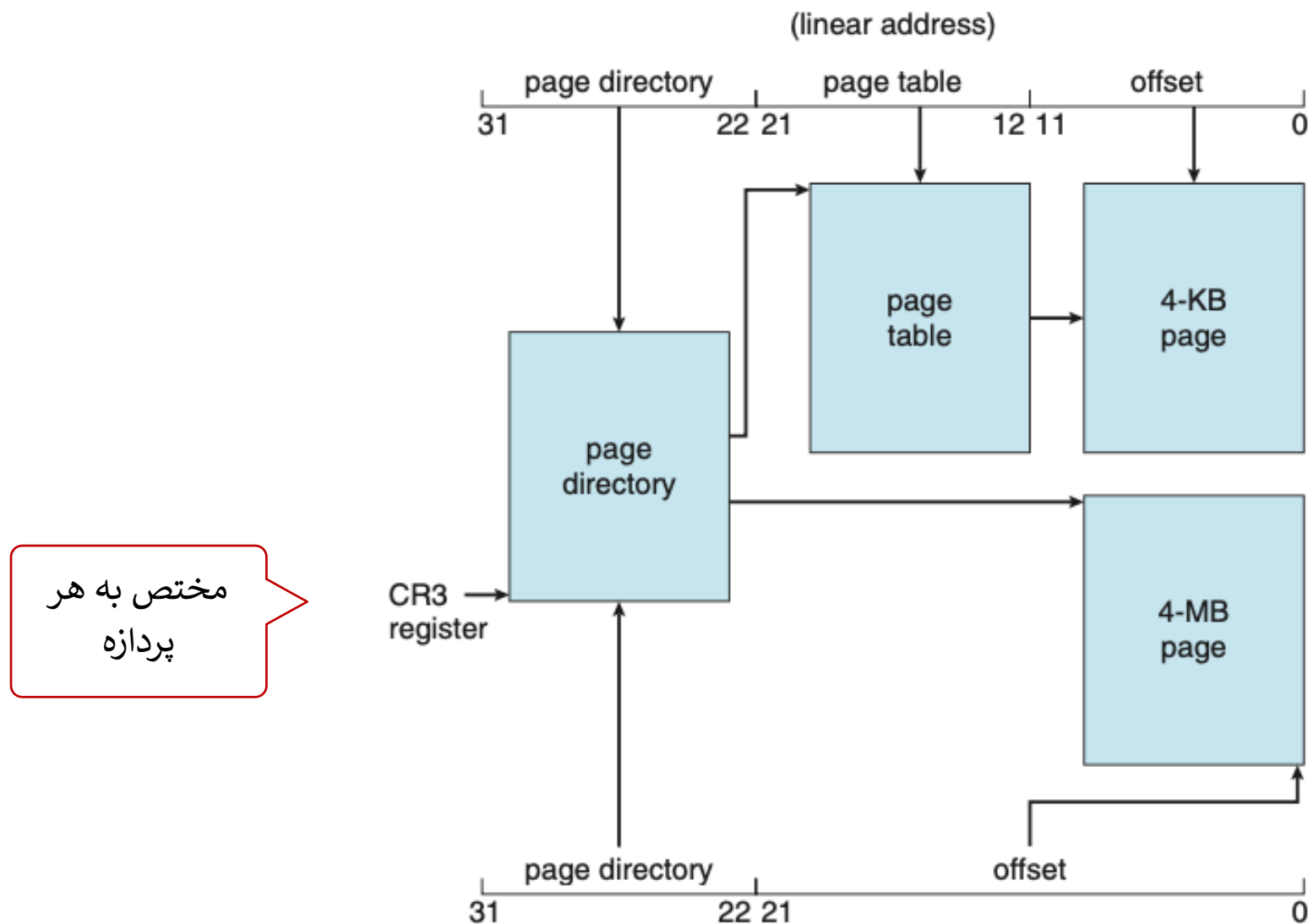
مرور - حافظه صفحه بندی شده



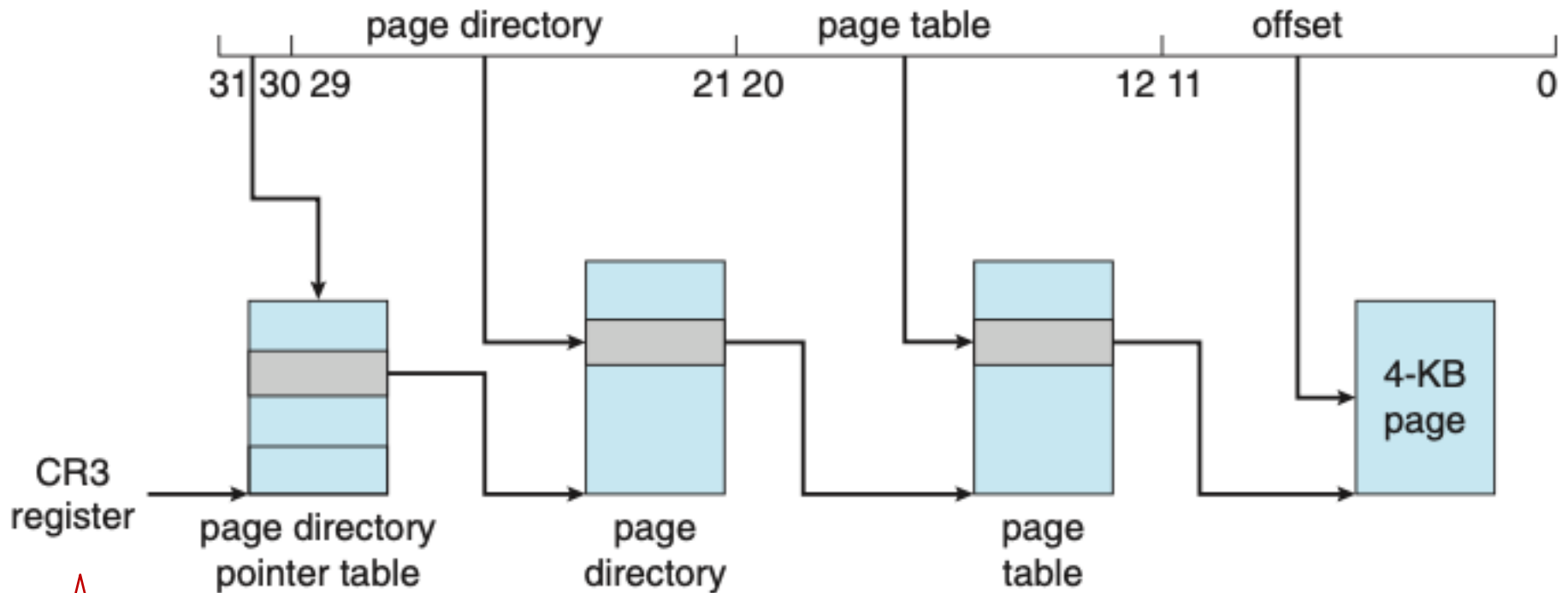
مرور - حافظه صفحه بندی شده - دو سطحی



مرور - حافظه صفحه بندی شده - دو سطحی (مثال IA-32)

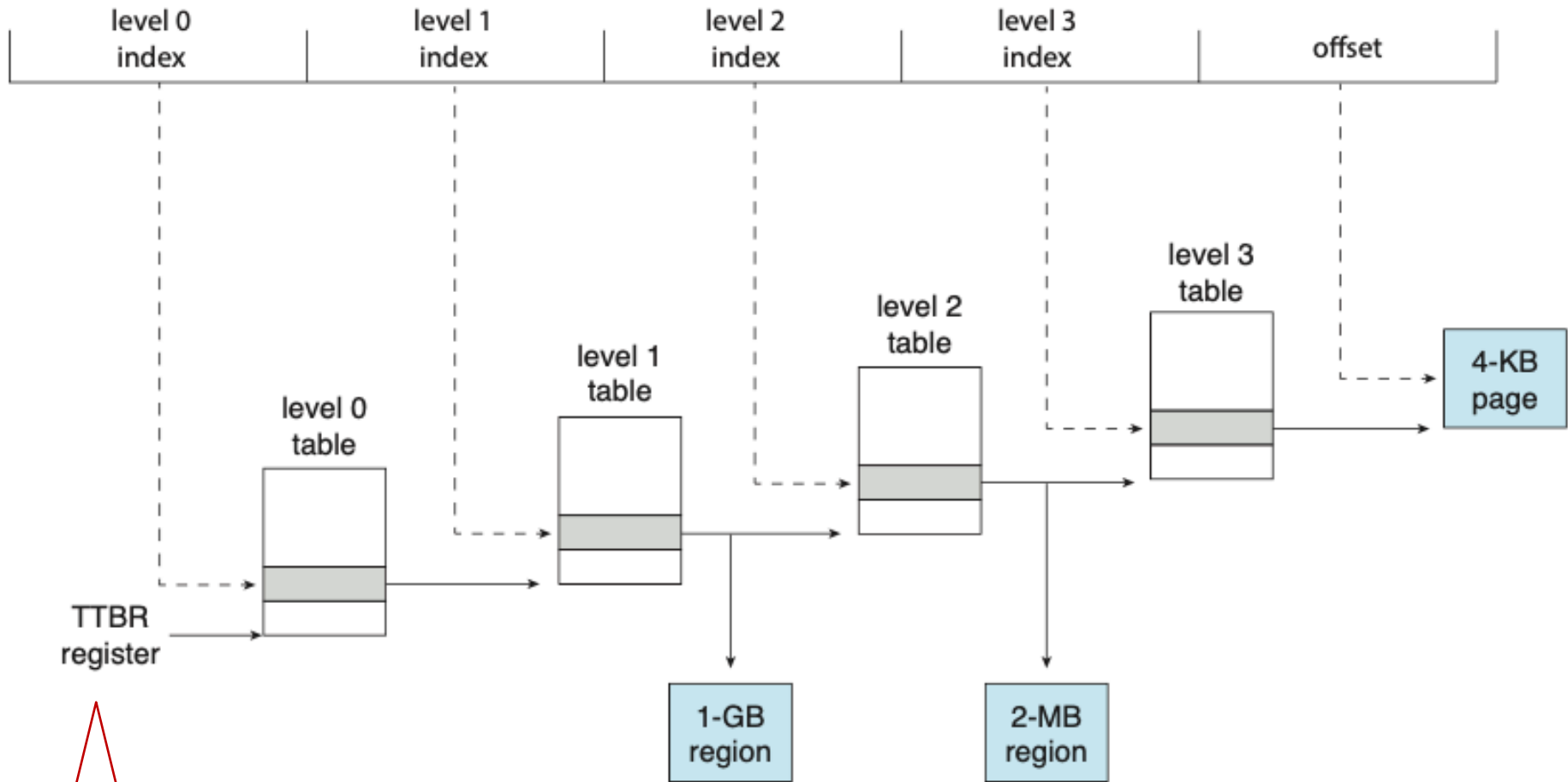


مرور - حافظه صفحه بندی شده - دو سطحی (مثال PAE)



مختص به هر
پردازه

مرور - حافظه صفحه بندی شده - دو سطحی (مثال PAE)



مختص به هر
پردازه

Figure 9.27 ARM four-level hierarchical paging.

قطعه بندی ...



Inverted page tables

- ❑ **Problem:**
 - ❖ Page table overhead increases with address space size
 - ❖ Page tables get too big to fit in memory!

- ❑ **Consider a computer with 64 bit addresses**
 - ❖ Assume 4 Kbyte pages (12 bits for the offset)
 - ❖ Virtual address space = 2^{52} pages!
 - ❖ Page table needs 2^{52} entries!
 - ❖ This page table is much too large for memory!
 - **Many peta-bytes per process page table**

Inverted page tables

How many mappings do we need (maximum) at any time?

Inverted page tables

How many mappings do we need (maximum) at any time?

We only need mappings for pages that are in memory!

Inverted page tables

- An **inverted page table**
 - ❖ Has one entry for every **frame** of memory
 - ❖ Records which page is in that frame
 - ❖ Is indexed by **frame number** not page number!
- So how can we search an inverted page table on a TLB miss fault?

Inverted page tables

- If we have a page number (from a faulting address) and want to find its page table entry, do we
 - ❖ Do an exhaustive search of all entries?

Inverted page tables

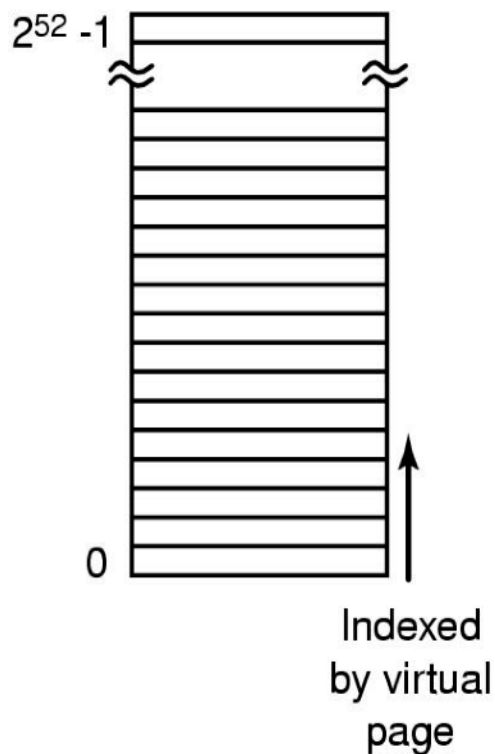
- If we have a page number (from a faulting address) and want to find its page table entry, do we
 - ❖ Do an exhaustive search of all entries?
 - ❖ No, that's too slow!
 - ❖ Why not maintain a **hash table** to allow fast access given a page number?
 - $O(1)$ lookup time with a good hash function

Hash Tables

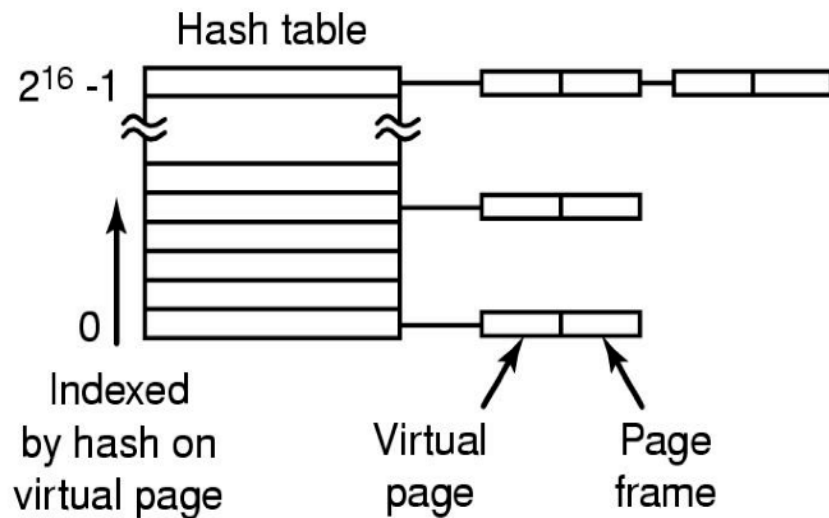
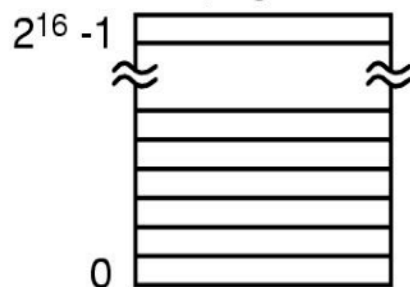
- ❑ **Data structure for associating a key with a value**
 - ❖ Perform hash function on key to produce a hash
 - ❖ Hash is a number that is used as an array index
 - ❖ Each element of the array can be a linked list of entries (to handle collisions)
 - ❖ The list must be searched to find the required entry for the key (entry's key matches the search key)
 - ❖ With a good hash function the list length will be very short

Inverted page table

Traditional page table with an entry for each of the 2^{52} pages



256-MB physical memory has 2^{16} 4-KB page frames



Which page table design is best?

- ❑ The best choice depends on CPU architecture
- ❑ 64 bit systems need inverted page tables
- ❑ Some systems use a combination of regular page tables together with segmentation (later)

Memory protection

- ❑ Protection through addressability
 - ❖ If address translation only allows a process to access its own pages, it is implementing memory protection
- ❑ But what if you want a process to be able to read and execute some pages but not write them?
 - ❖ text segment
- ❑ Or read and write them but not execute them?
 - ❖ stack
- ❑ Need some way of implementing protection based on access type

Memory protection

- ❑ **How is protection checking implemented?**
 - ❖ compare page protection bits with process capabilities and operation types **on every load/store**
 - ❖ sounds expensive!
 - ❖ Requires hardware support!
- ❑ **How can protection checking be done efficiently?**
 - ❖ Use the TLB as a “protection” look-aside buffer as well as a translation lookaside buffer
 - ❖ Use special segment registers

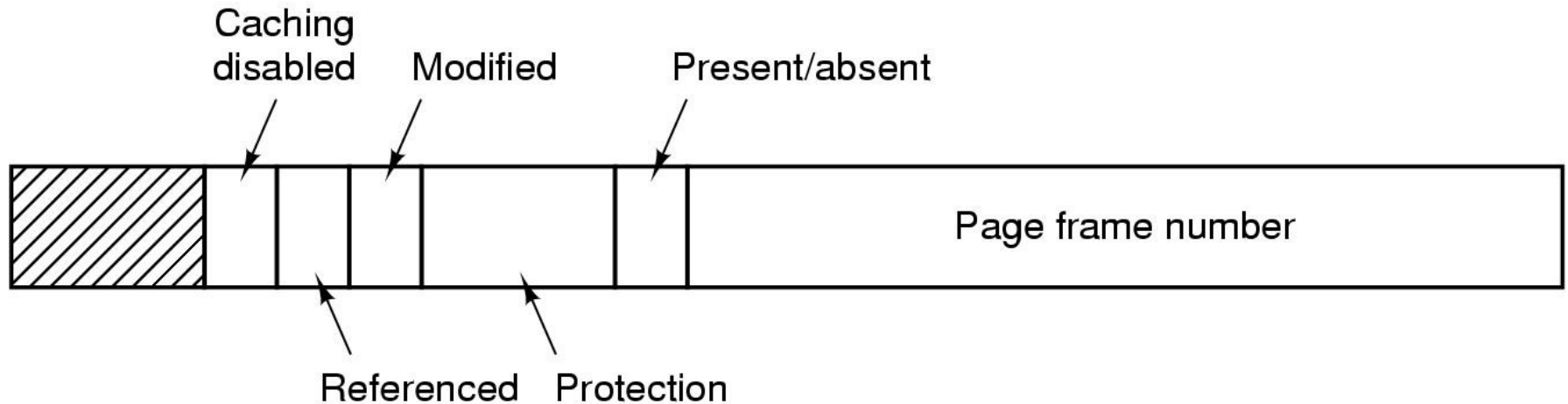
Protection lookaside buffer

- ❑ A TLB is often used for more than just “translation”
- ❑ Memory accesses need to be checked for validity
 - ❖ Does the address refer to an allocated segment of the address space?
 - If not: **segmentation fault!**
 - ❖ Is this process allowed to access this memory segment?
 - If not: **segmentation/protection fault!**
 - ❖ Is the type of access valid for this segment?
 - Read, write, execute ...?
 - If not: **protection fault!**

Page-grain protection checking with a TLB

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Page grain protection in a page table



**A typical page table entry with support for
page grain protection**

Memory protection granularity

- ❑ At what granularity should protection be implemented?
- ❑ **page-level?**
 - ❖ A lot of overhead for storing protection information for non-resident pages
- ❑ **segment level?**
 - ❖ Coarser grain than pages
 - ❖ Makes sense if contiguous groups of pages share the same protection status

Segment-grain protection

- All pages within a segment usually share the same protection status
 - ❖ So we should be able to batch the protection information
- Then why not just use segment-size pages?

Segment-grain protection

- All pages within a segment usually share the same protection status
 - ❖ So we should be able to batch the protection information
- Then why not just use segment-size pages?
 - ❖ Segments vary in size from process to process
 - ❖ Segments change size dynamically (stack, heap etc)
- Need to manage addressability, access-based protection and memory allocation separately
 - ❖ Coarse-grain protection can be implemented simply through addressability

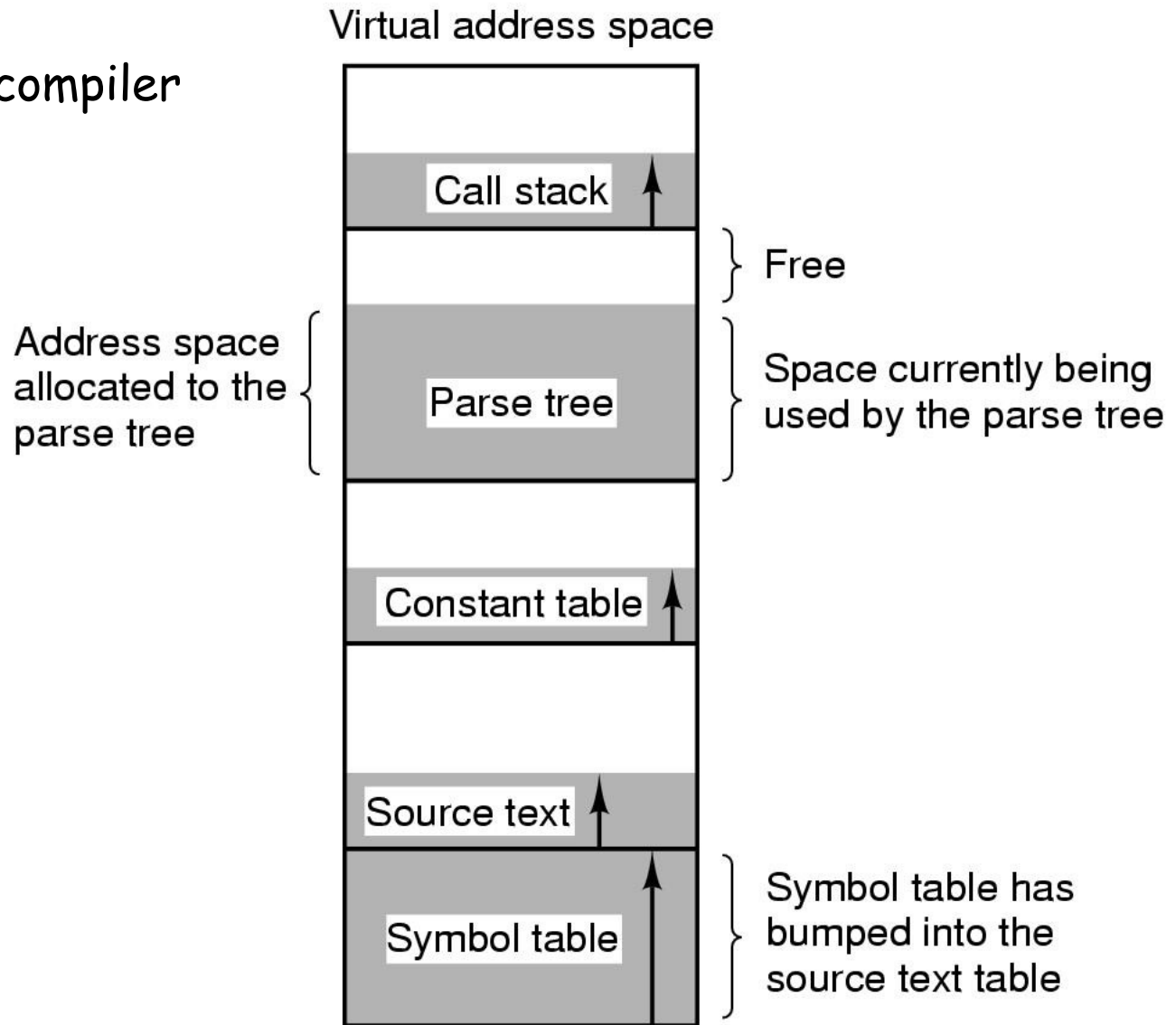
Segmented address spaces

- ❑ **Traditional Virtual Address Space**
 - ❖ “flat” address space (1 dimensional)

- ❑ **Segmented Address Space**
 - ❖ Program made of several pieces or “segments”
 - ❖ Each segment is its own mini-address space
 - ❖ Addresses within a segment start at zero
 - ❖ The program must always say which segment it means
 - either embed a segment id in an address
 - or load a value into a segment register and refer to the register
 - ❖ Addresses:
Segment + Offset
 - ❖ Each segment can grow independently of others

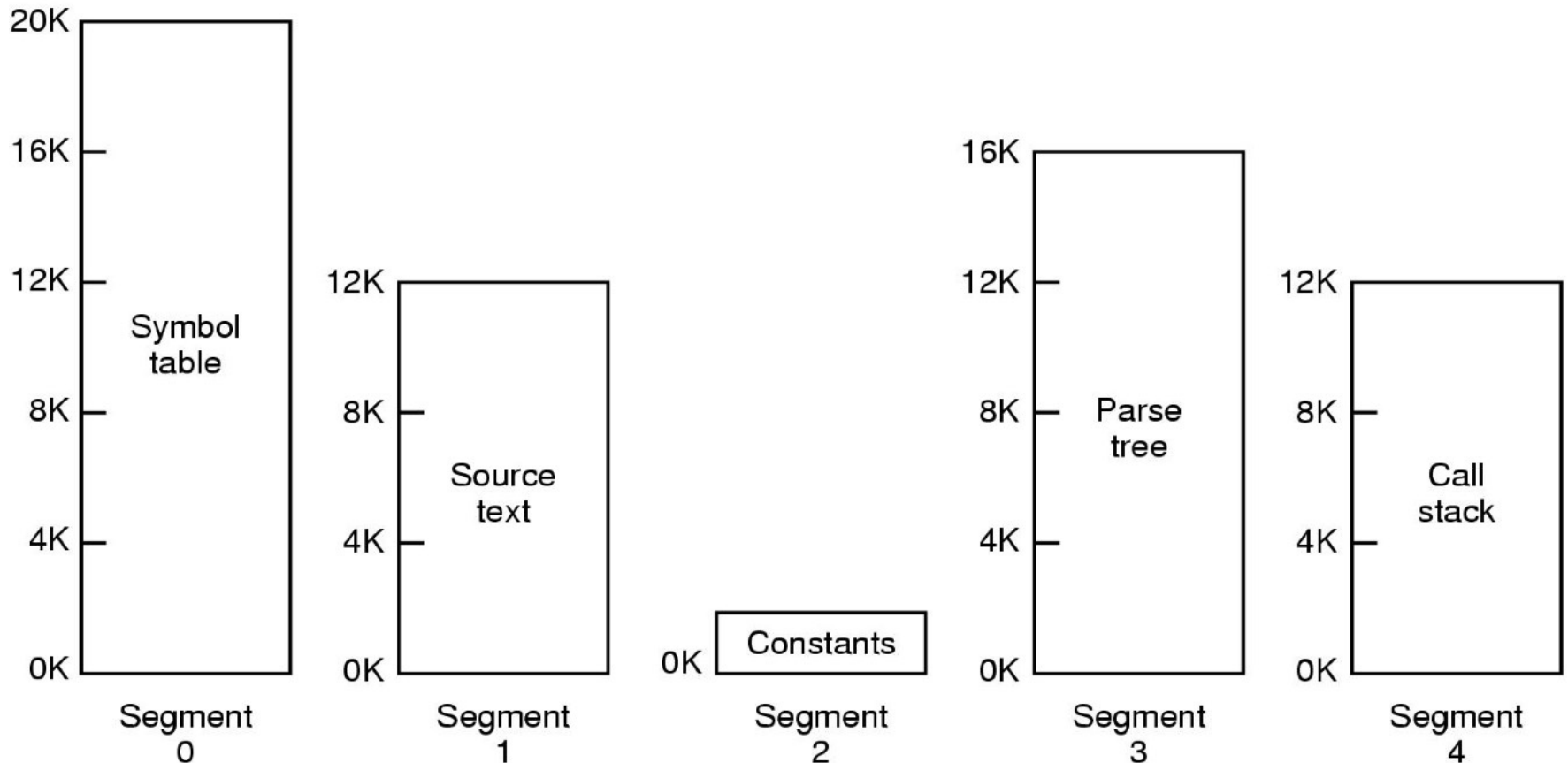
Segmentation in a single address space

Example: A compiler

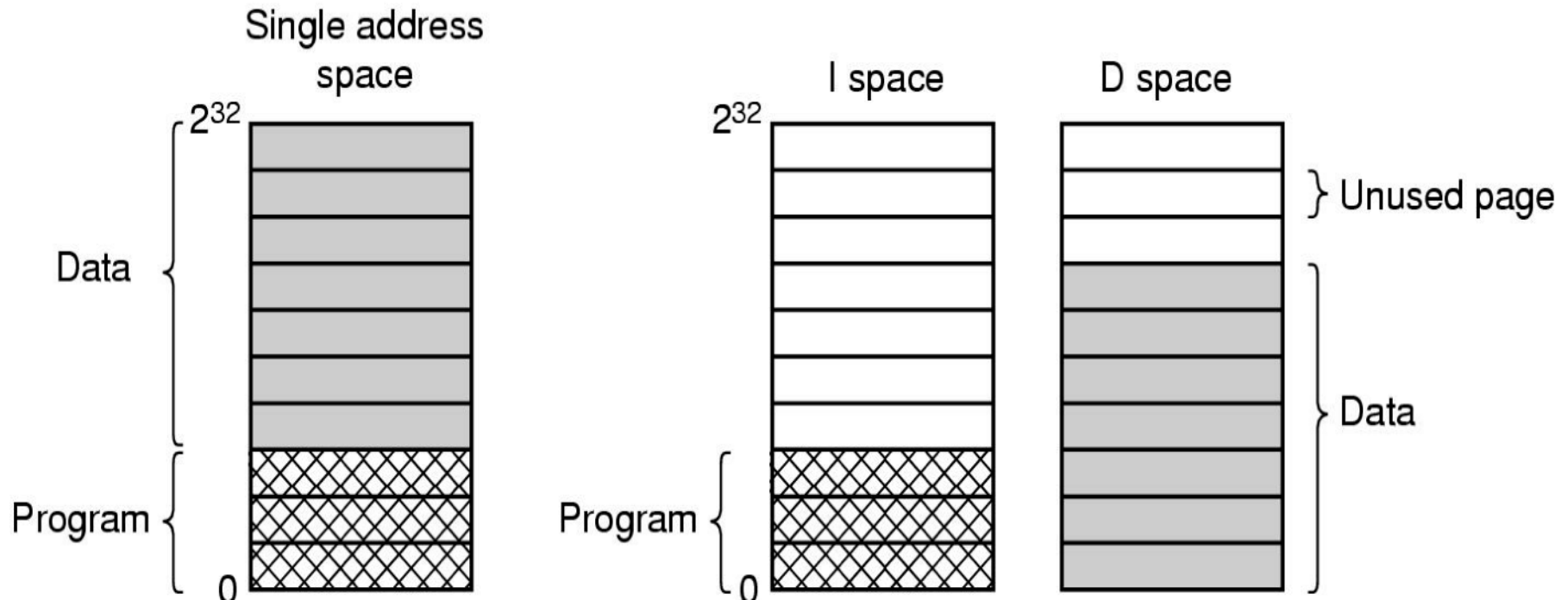


Segmented memory

- Each space grows, shrinks independently!



Separate instruction and data spaces



* One address space

* Separate I and D spaces

Comparison of paging vs. segmentation

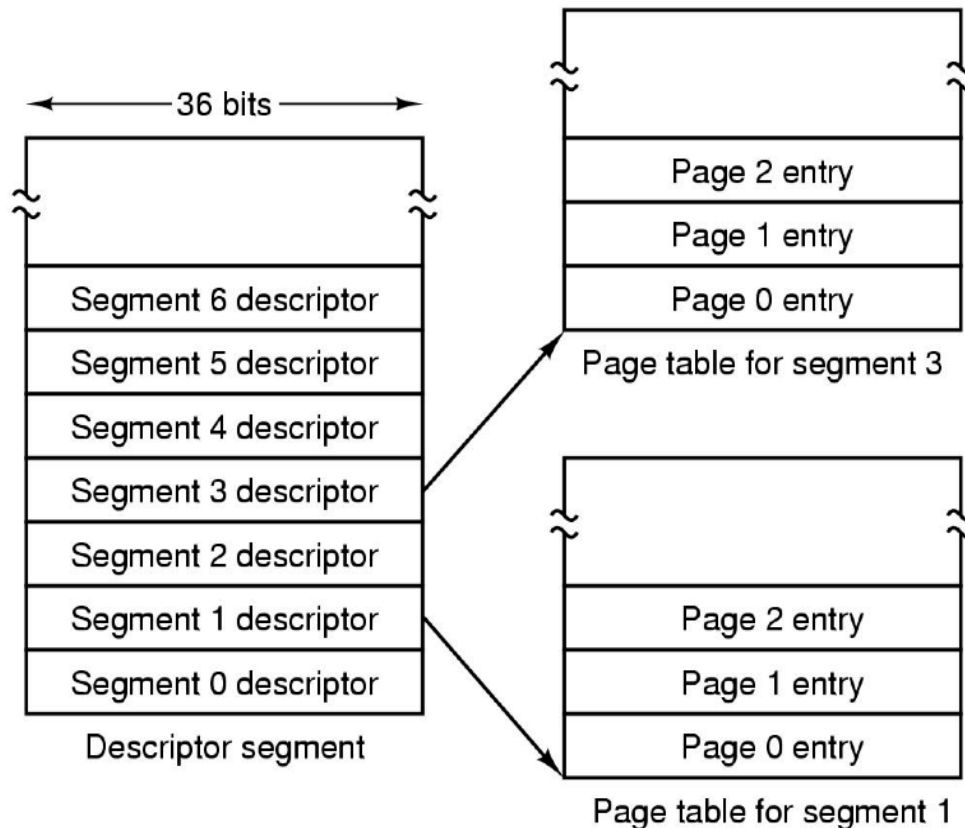
Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Segmentation vs. Paging

- ❑ Do we need to choose one or the other?
- ❑ Why not use both together
 - ❖ Paged segments
 - ❖ Paging for memory allocation
 - ❖ Segmentation for maintaining protection information at a coarse granularity
 - ❖ Segmentation and paging in combination for translation

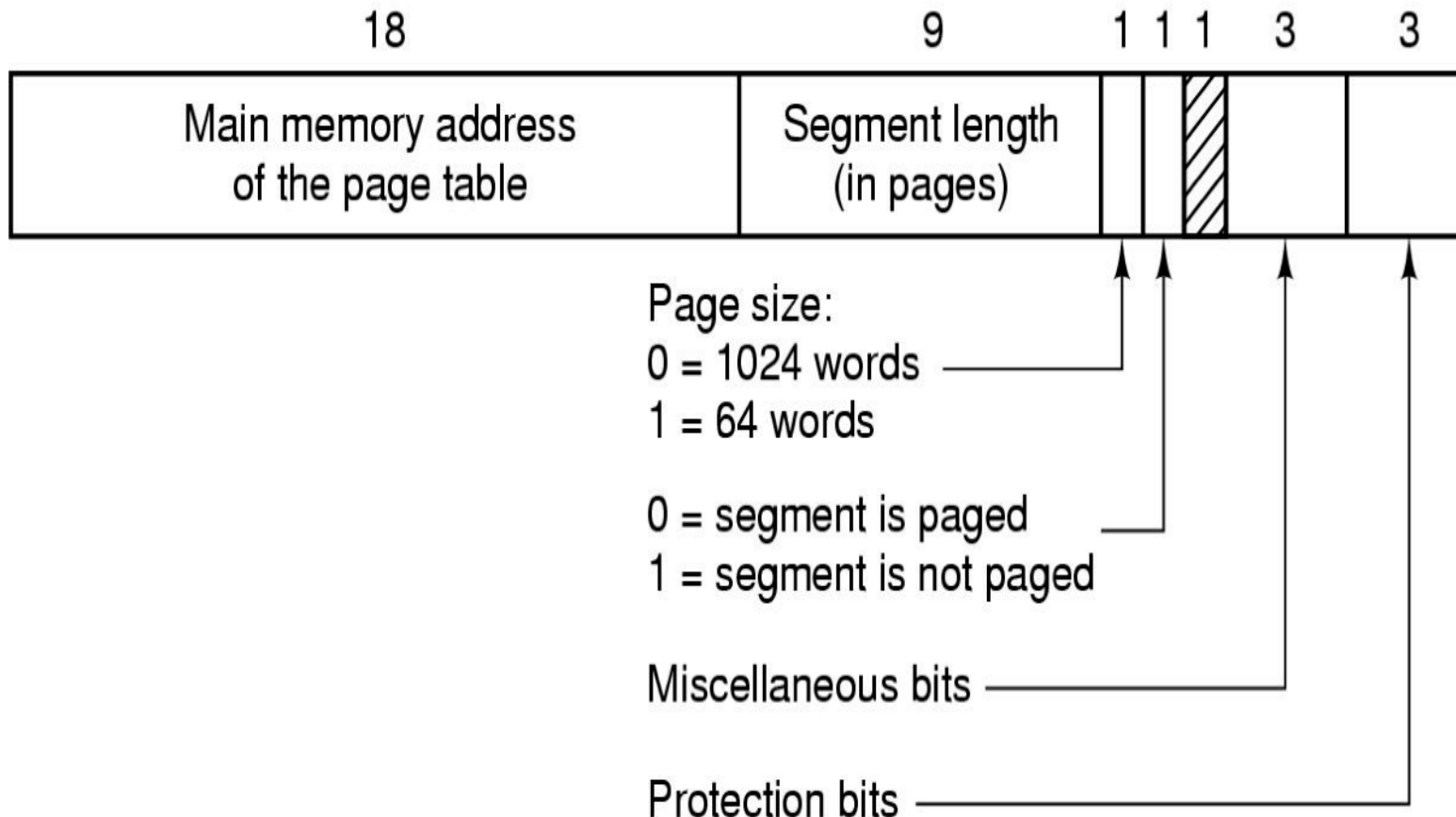
Segmentation with paging (MULTICS)

- ❑ Each segment is divided up into pages.
- ❑ Each segment descriptor points to a page table.



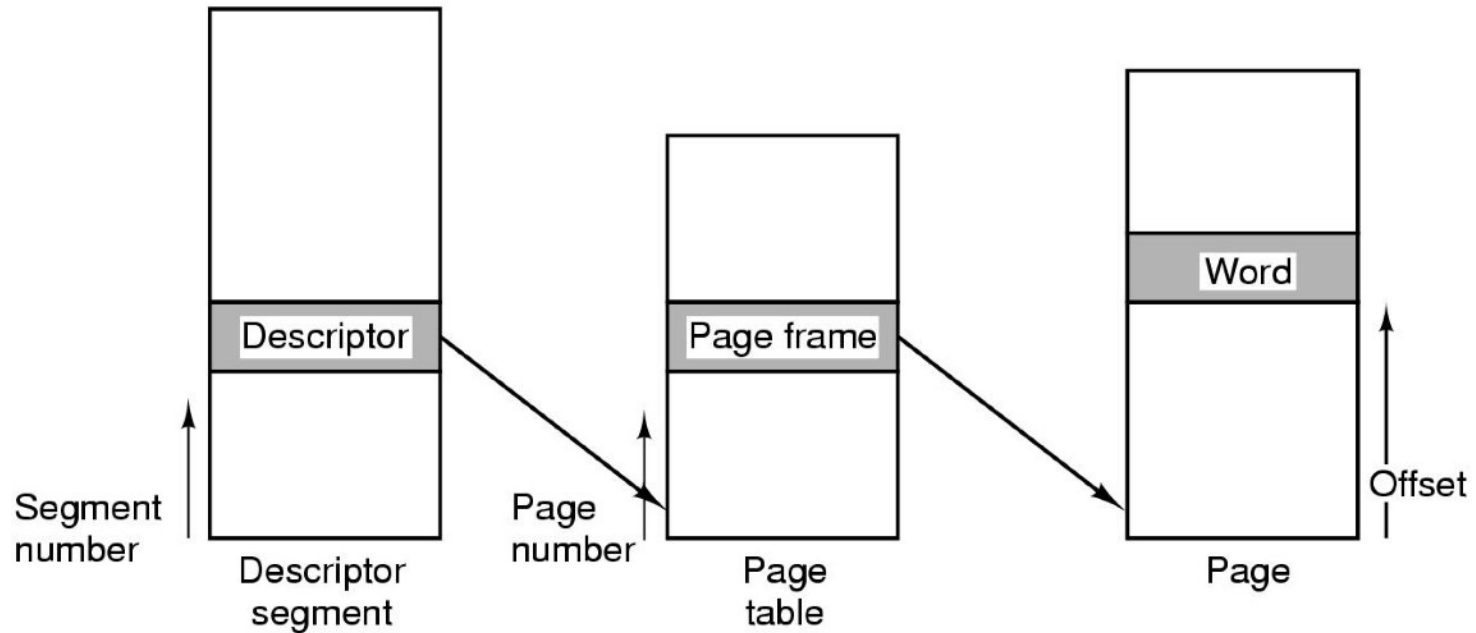
Segmentation with paging (MULTICS)

- Each entry in segment table...



Segmentation with paging: MULTICS

MULTICS virtual address



Conversion of a 2-part MULTICS address into a main memory address

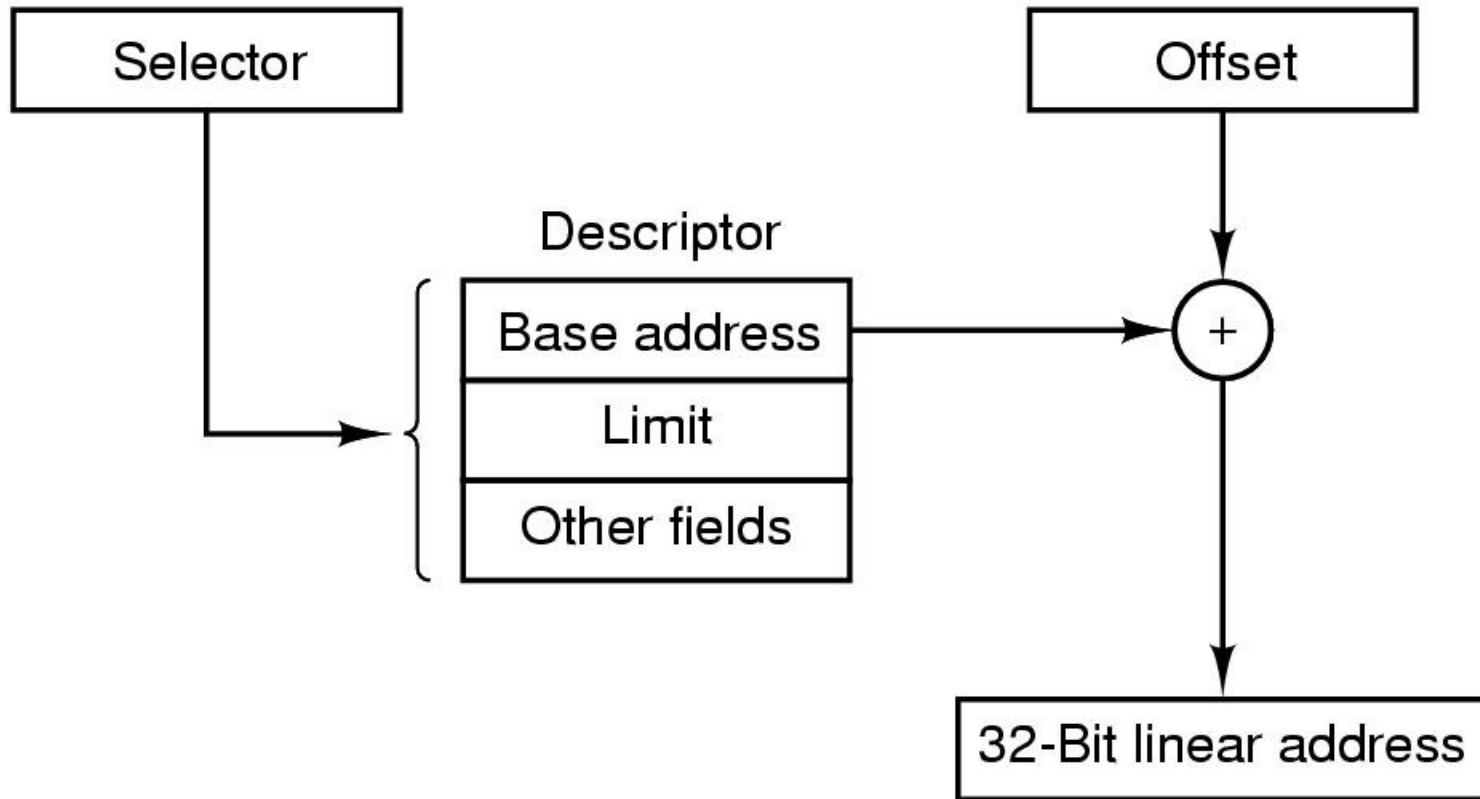
Segmentation with Paging: TLB operation

Comparison field		Page frame	Protection	Age	Is this entry used?
Segment number	Virtual page				↓
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

- ❑ Simplified version of the MULTICS TLB
- ❑ Existence of 2 page sizes makes actual TLB more complicated

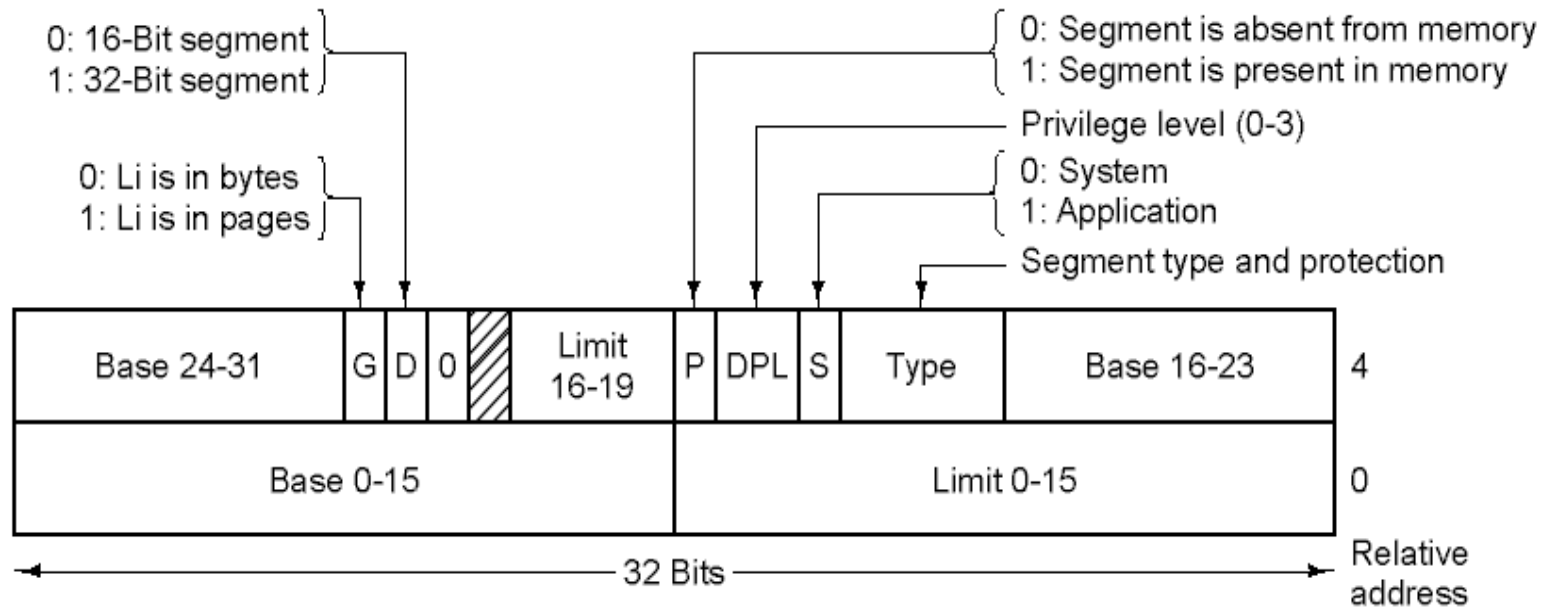
Spare slides

Segmentation & paging in the Pentium



Conversion of a (selector, offset) pair to a linear address

Segmentation & paging in the Pentium



- ❑ Pentium segment descriptor