بسم الله الرحمن الرحیم

۱ «سیستم عامل»

جلسه ۱۳: مدیریت حافظه (۱)

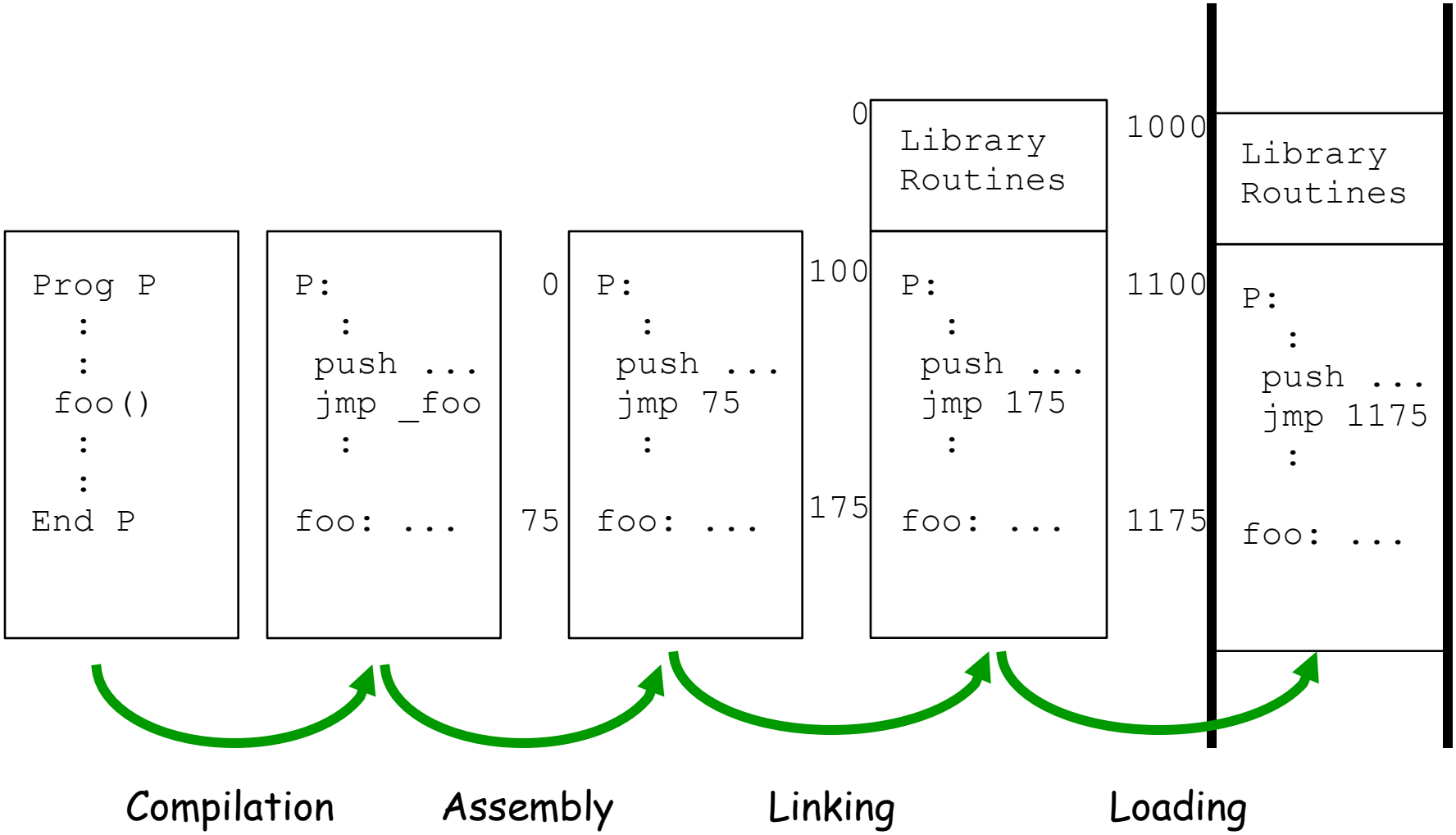
Memory management

- **Memory – a linear array of bytes**
  - Holds O.S. and programs (processes)
  - Each cell (byte) is named by a unique memory address
- **Recall, processes are defined by an address space, consisting of text, data, and stack regions**
- **Process execution**
  - CPU fetches instructions from the text region according to the value of the program counter (PC)
  - Each instruction may request additional operands from the data or stack region

Addressing memory

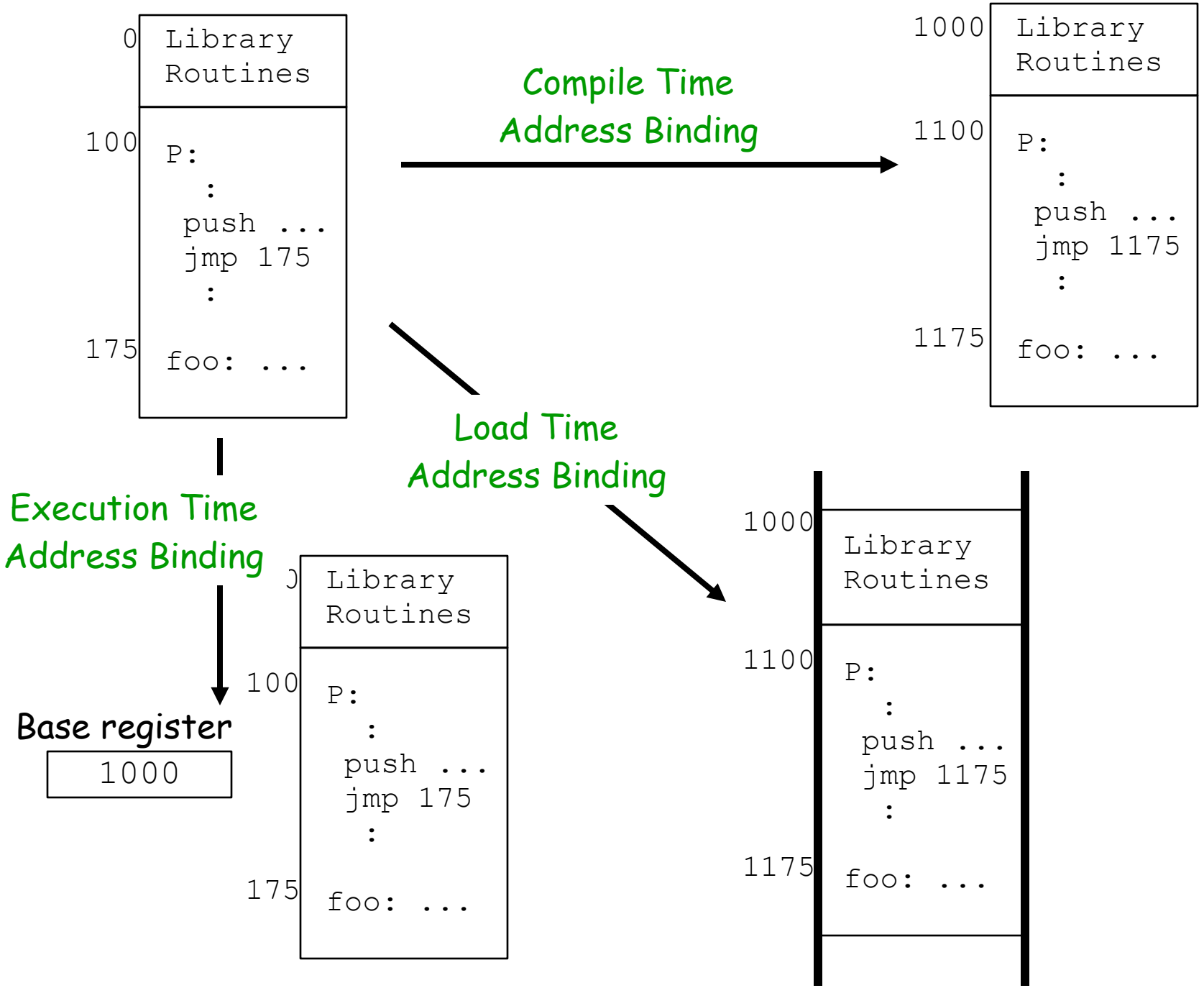
- **Cannot know ahead of time where in memory a program will be loaded!**
- **Compiler produces code containing embedded addresses**
  - these addresses can't be absolute (physical addresses)
- **Linker combines pieces of the program**
  - Assumes the program will be loaded at address 0
- **We need to bind the compiler/linker generated addresses to the actual memory locations**

Relocatable address generation



Address binding

- **Address binding**
  - fixing a physical address to the logical address of a process' address space
- **Compile time binding**
  - if program location is fixed and known ahead of time
- **Load time binding**
  - if program location in memory is unknown until run-time AND location is fixed
- **Execution time binding**
  - if processes can be moved in memory during execution
  - Requires hardware support!


0
Library Routines
100
P:
:
push ...
jmp 175
:
175
foo: ...
Compile Time Address Binding
1000
Library Routines
1100
P:
:
push ...
jmp 1175
:
1175
foo: ...
Load Time Address Binding
Execution Time Address Binding
Base register
1000
0
Library Routines
100
P:
:
push ...
jmp 175
:
175
foo: ...
1000
Library Routines
1100
P:
:
push ...
jmp 1175
:
1175
foo: ...

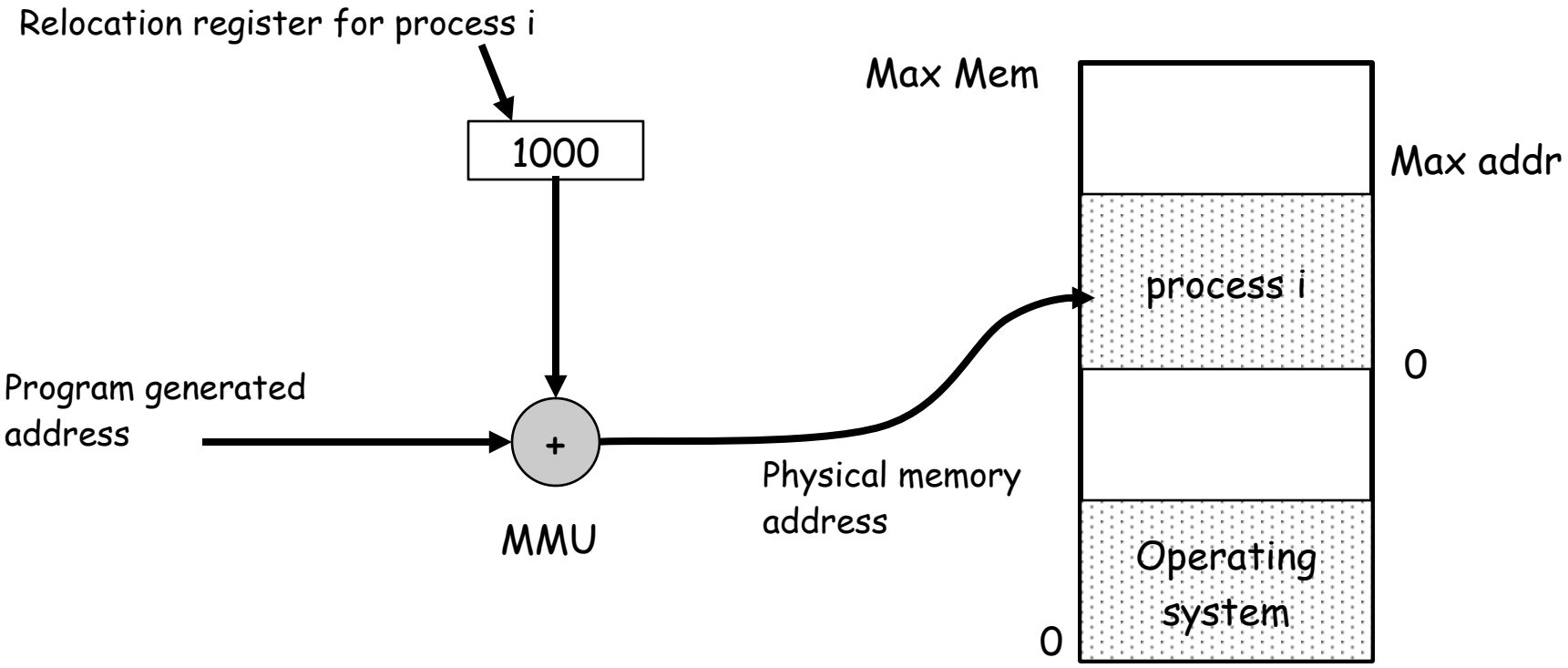
Runtime binding – base & limit registers

- **Simple runtime relocation scheme**
  - Use 2 registers to describe a partition

- **For every address generated, at runtime...**
  - Compare to the limit register (& abort if larger)
  - Add to the base register to give physical memory address

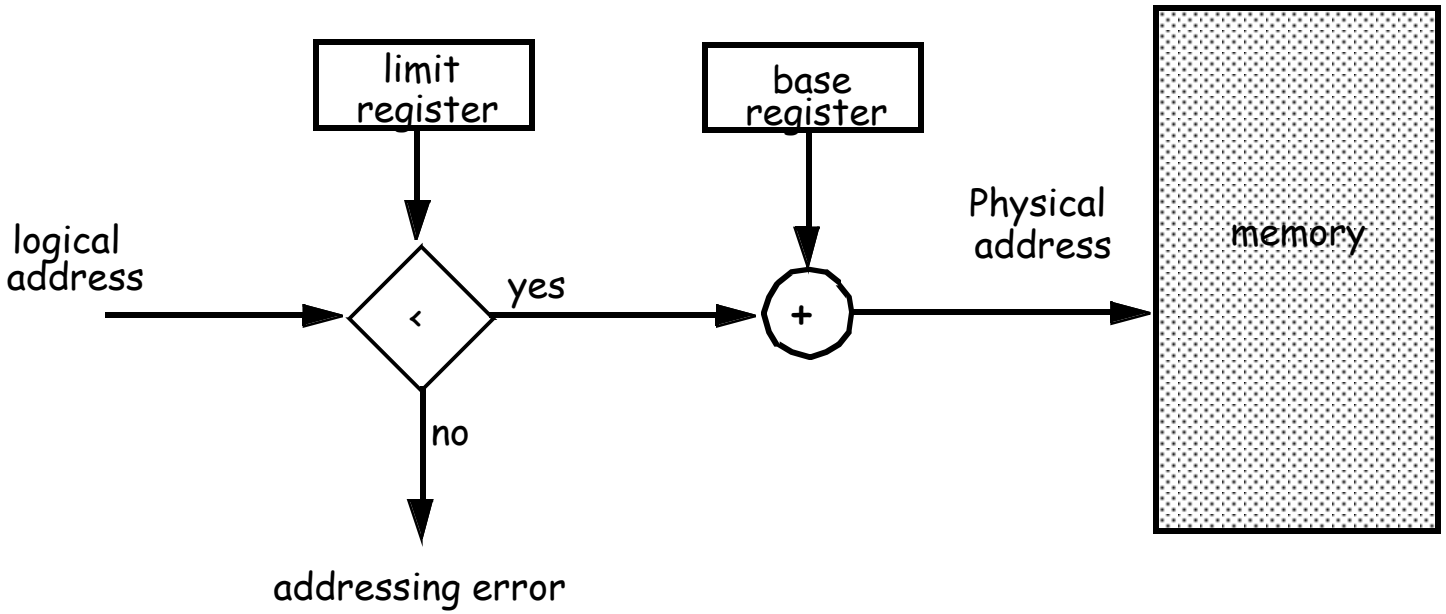
Dynamic relocation with a base register

- Memory Management Unit (MMU) - dynamically converts logical addresses into physical address
- MMU contains base address register for running process



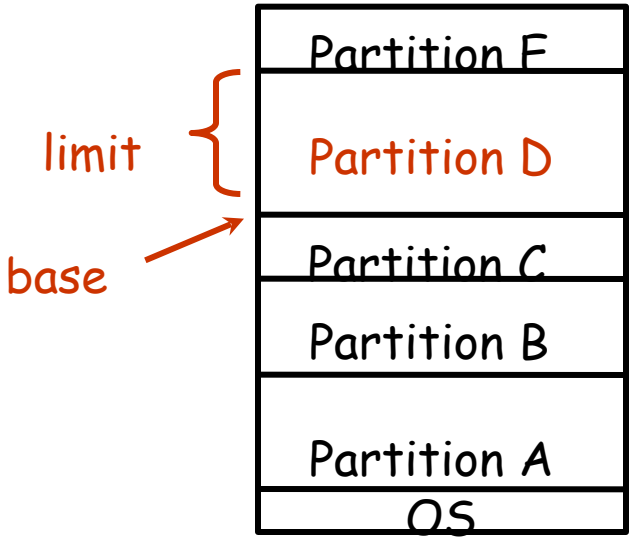
Protection using base & limit registers

- **Memory protection**
  - Base register gives starting address for process
  - Limit register limits the offset accessible from the relocation register



Multiprogramming with base and limit registers

- **Multiprogramming: a separate partition per process**
- **What happens on a context switch?**
  - Store process A's base and limit register values
  - Load new values into base and limit registers for process B



Swapping

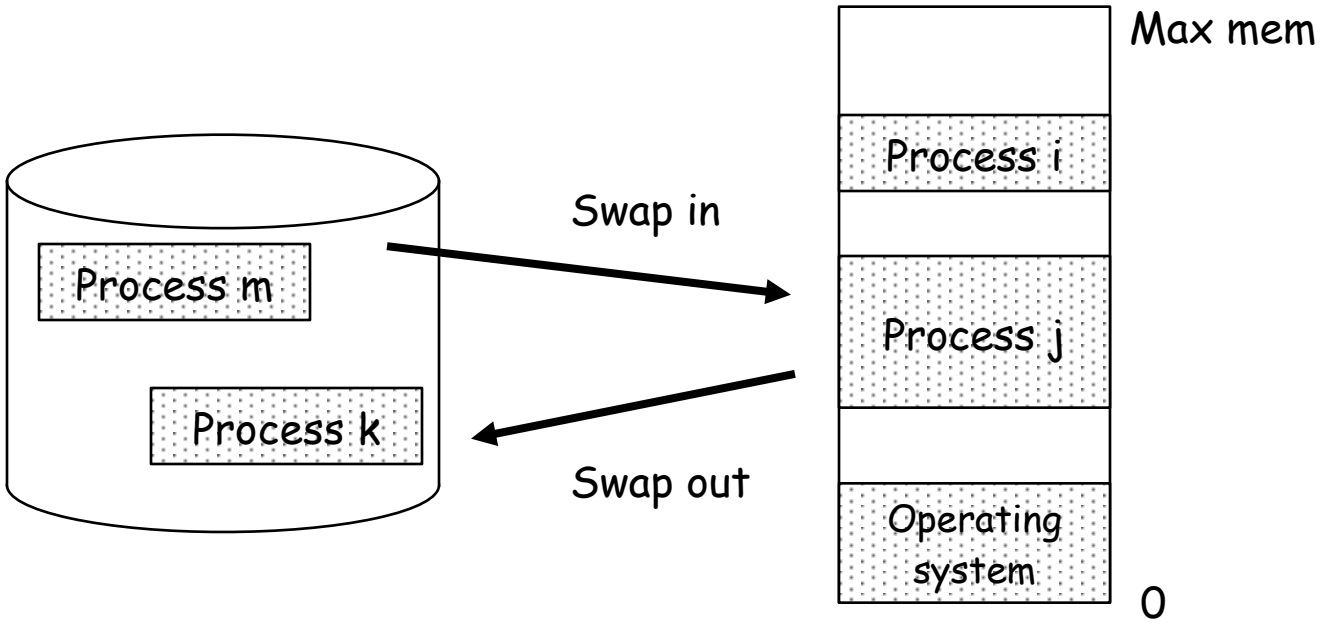
- **When a program is running...**
  - The entire program must be in memory
  - Each program is put into a single partition

- **When the program is not running...**
  - May remain resident in memory
  - May get "swapped" out to disk

- **Over time...**
  - Programs come into memory when they get swapped in
  - Programs leave memory when they get swapped out

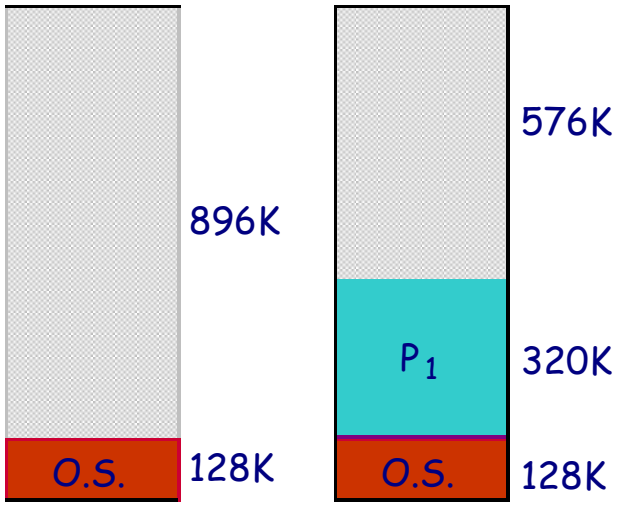
Basics - swapping

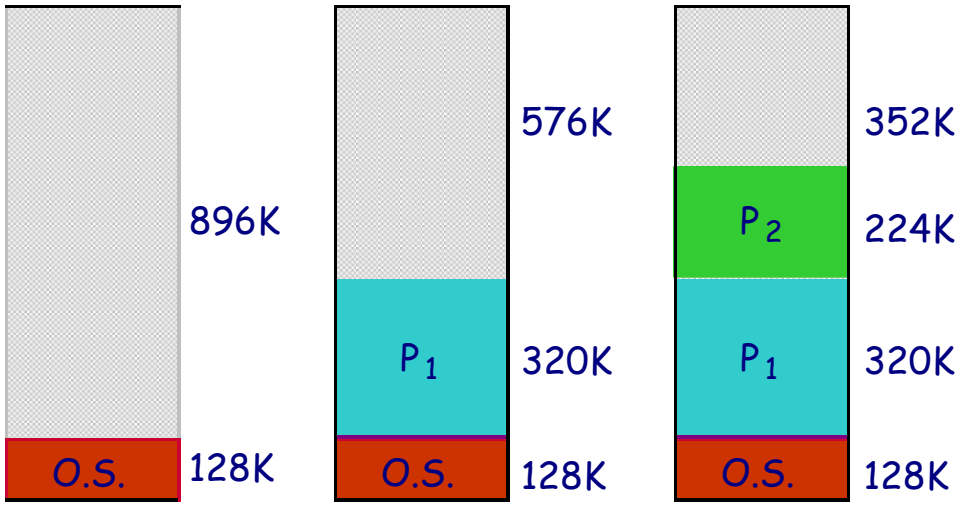
- **Benefits of swapping:**
  - Allows multiple programs to be run concurrently
  - … more than will fit in memory at once

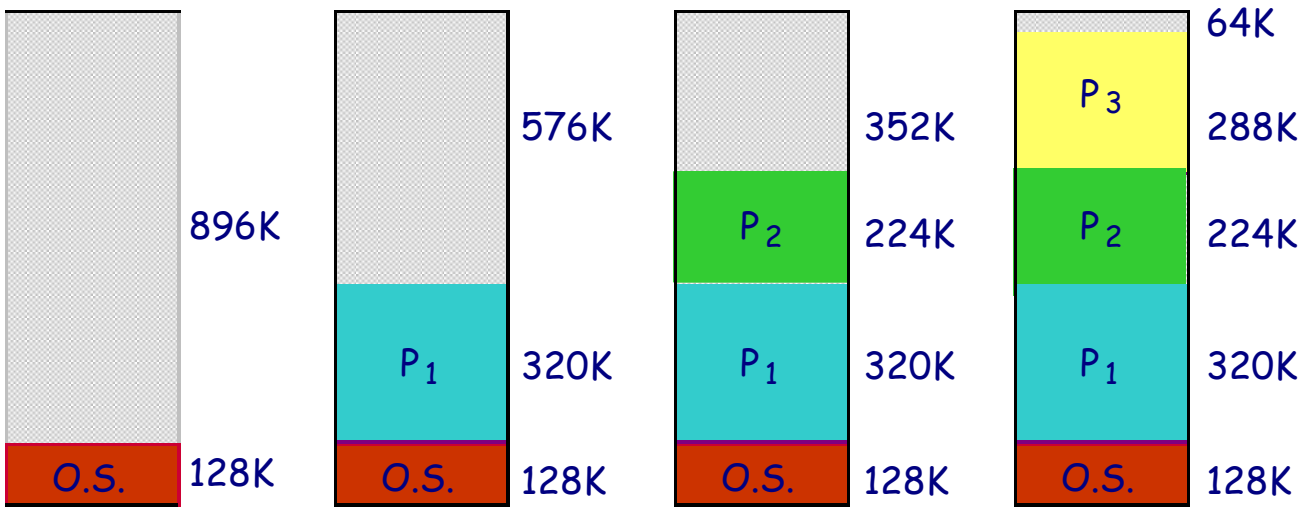


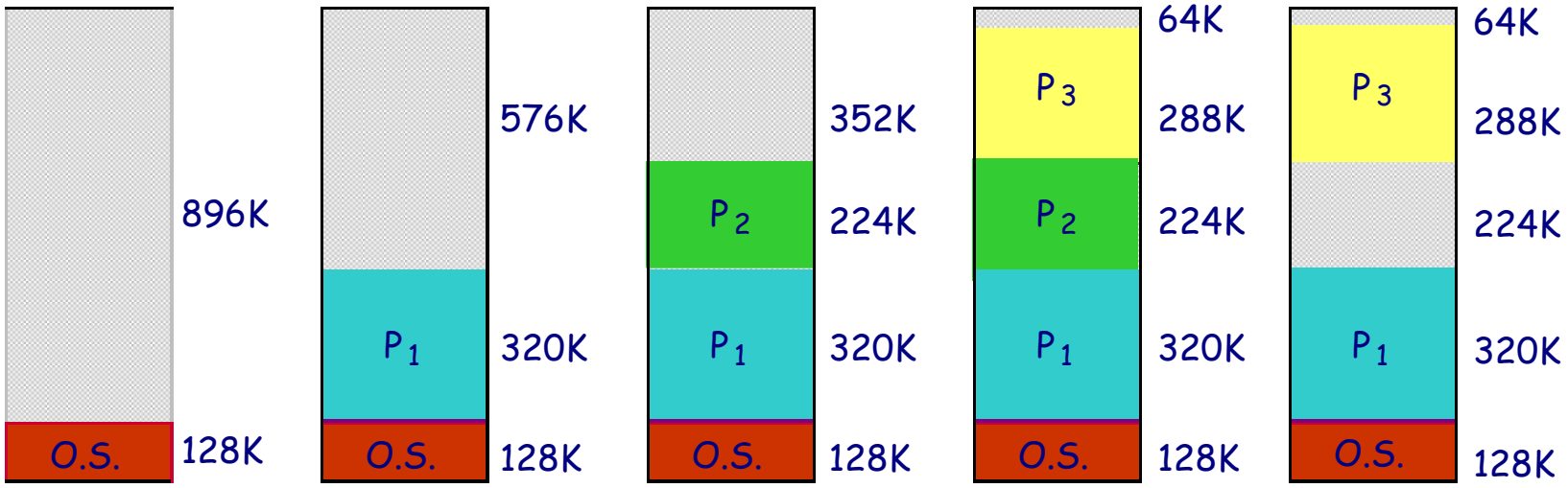
Swapping can lead to fragmentation

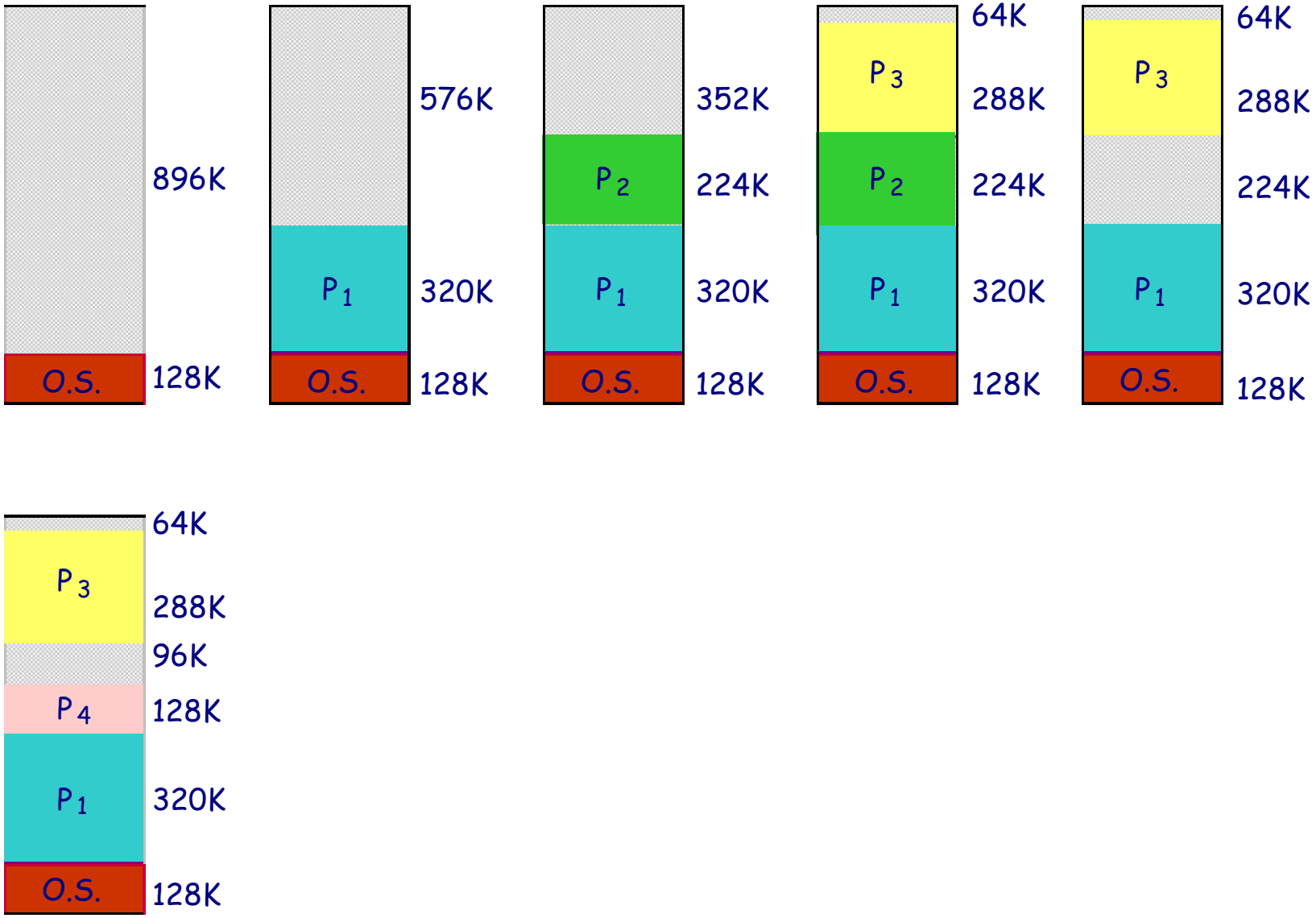

896K
O.S.
128K

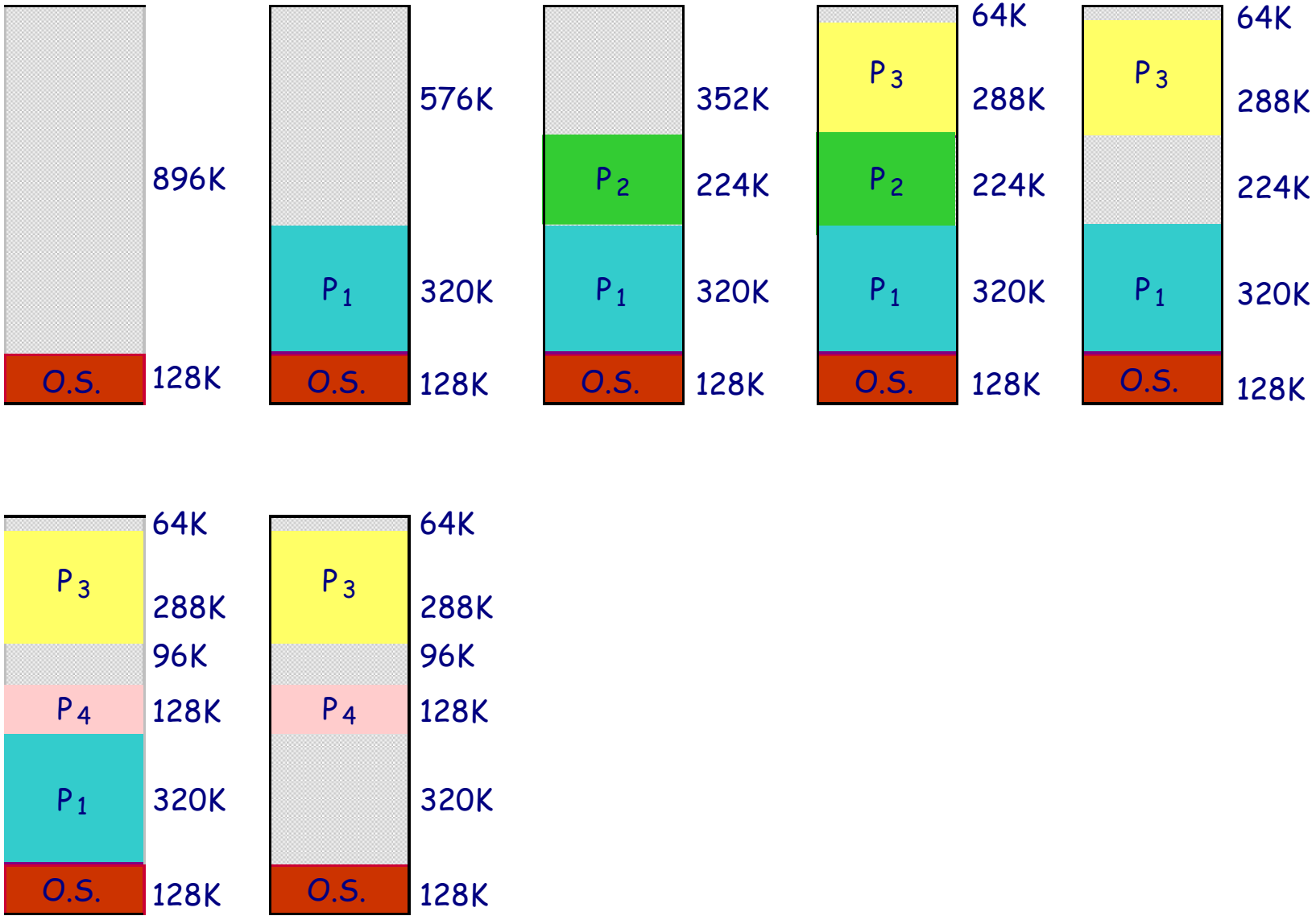

896K
O.S.
128K
576K
P1
320K
O.S.
128K

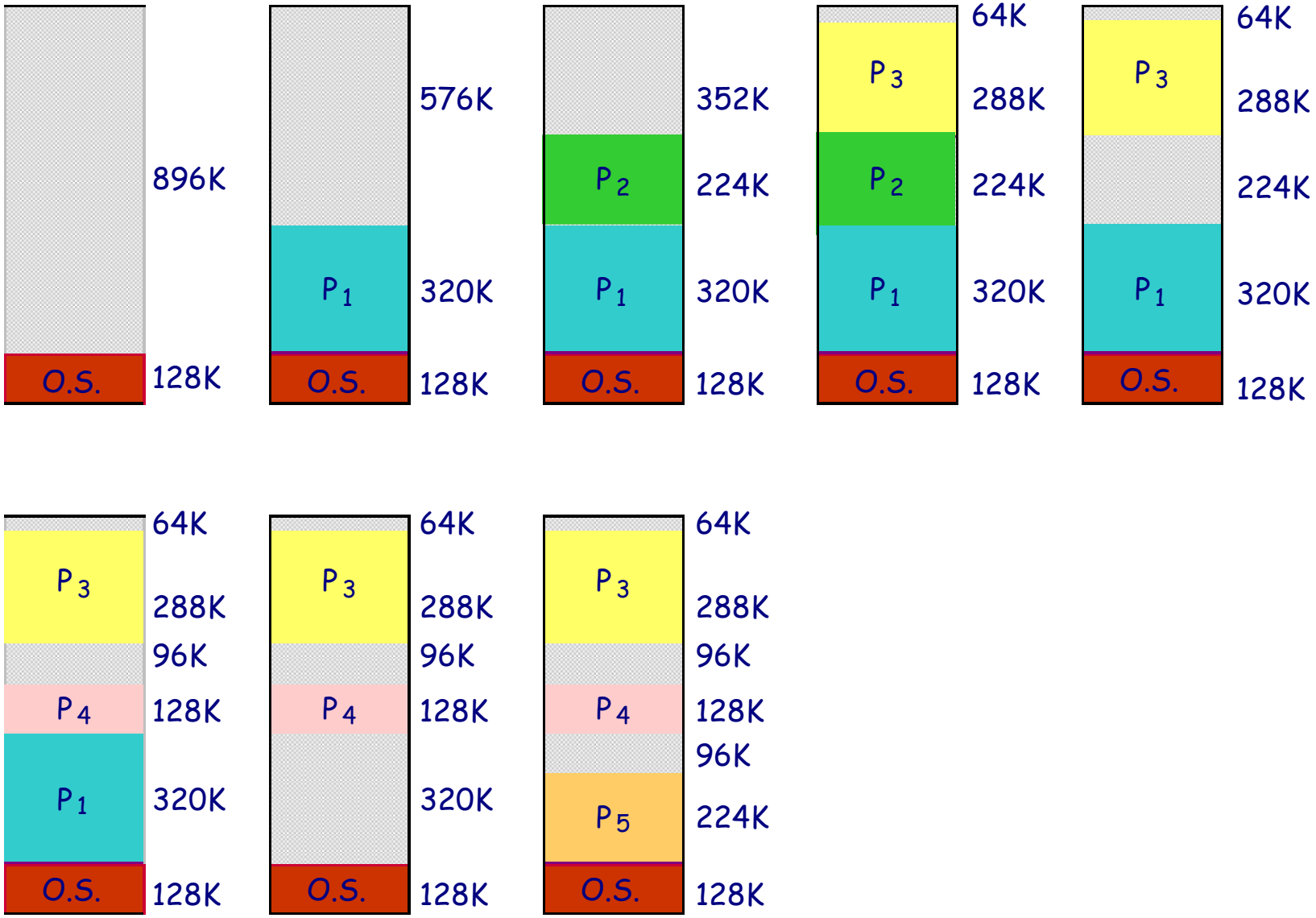

896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K

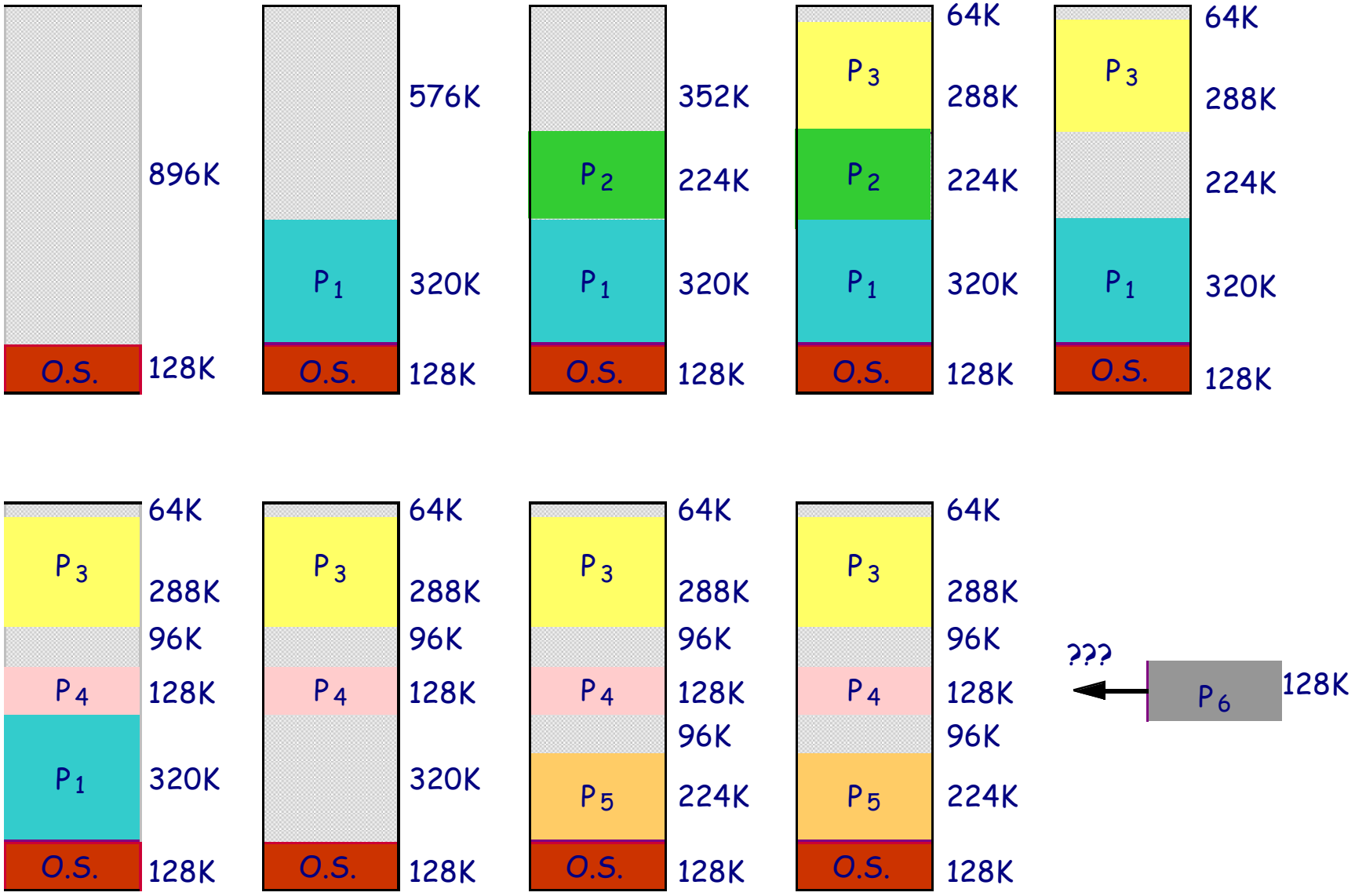

896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K


896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
224K
P1
320K
O.S.
128K


896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
224K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
P1
320K
O.S.
128K

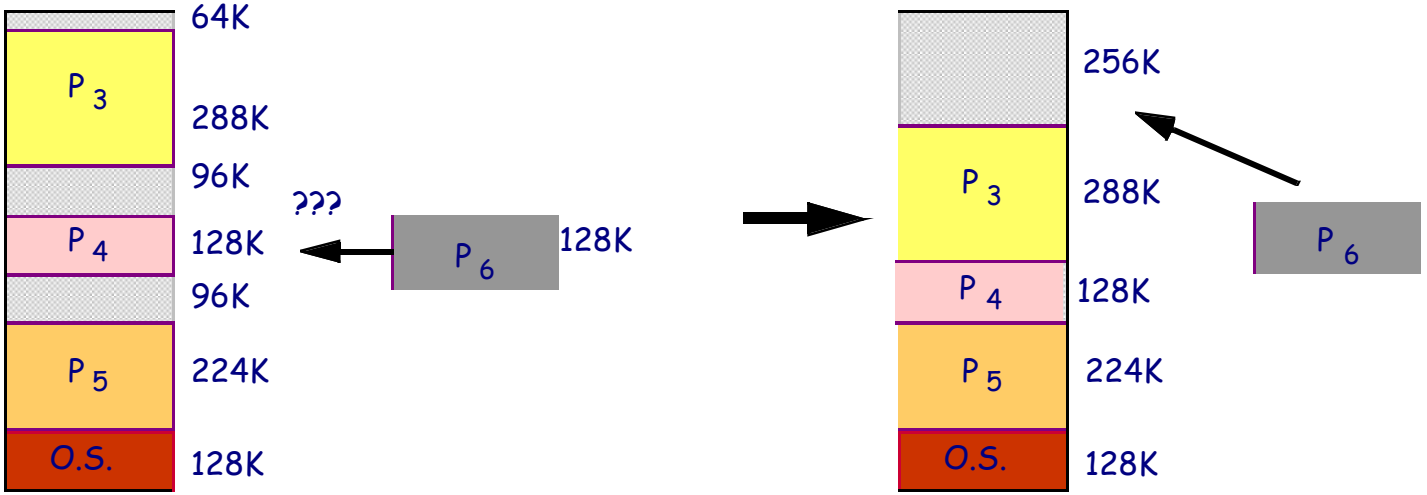

896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
224K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
320K
O.S.
128K


896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
224K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
96K
P5
224K
O.S.
128K


896K
128K
O.S.
576K
320K
P1
128K
O.S.
352K
224K
P2
320K
P1
128K
O.S.
64K
P3
288K
224K
P2
320K
P1
128K
O.S.
64K
P3
288K
224K
P1
320K
128K
O.S.
64K
P3
288K
96K
128K
P4
320K
P1
128K
O.S.
64K
P3
288K
96K
P4
320K
128K
O.S.
64K
P3
288K
96K
P4
128K
96K
224K
P5
128K
O.S.
64K
P3
288K
96K
P4
128K
96K
224K
P5
128K
O.S.
???
P6
128K

Dealing with fragmentation

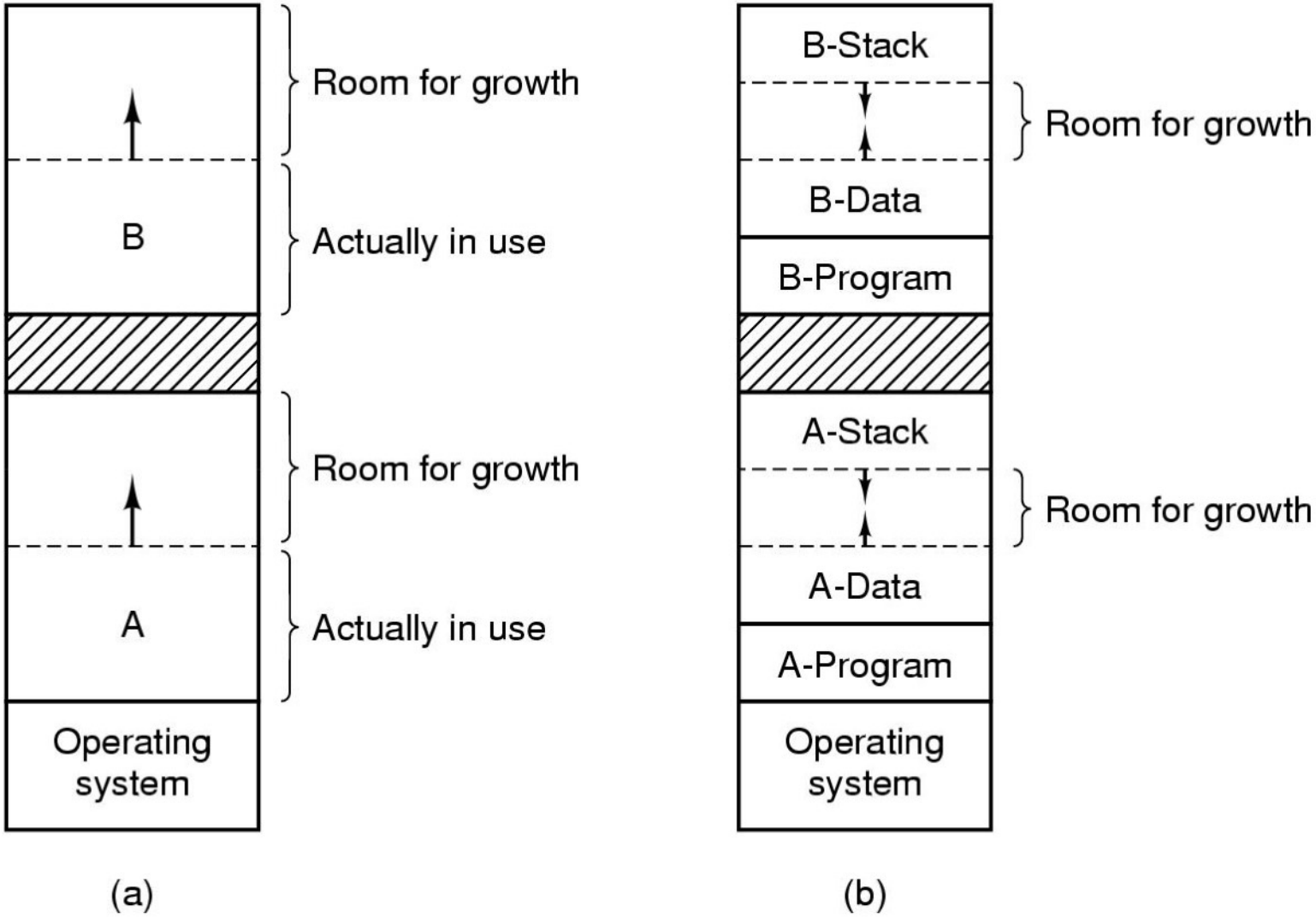
- **Compaction – from time to time shift processes around to collect all free space into one contiguous block**
  - Memory to memory copying overhead
    - **memory to disk to memory for compaction via swapping**



How big should partitions be?

- **Programs may want to grow during execution**
  - More room for stack, heap allocation, etc
- **Problem:**
  - If the partition is too small programs must be moved
  - Requires copying overhead
  - Why not make the partitions a little larger than necessary to accommodate "some" cheap growth?

Allocating extra space within partitions



Data structures for managing memory

- **Each chunk of memory is either**
  - Used by some process or unused ("free")
- **Operations**
  - Allocate a chunk of unused memory big enough to hold a new process
  - Free a chunk of memory by returning it to the free pool after a process terminates or is swapped out

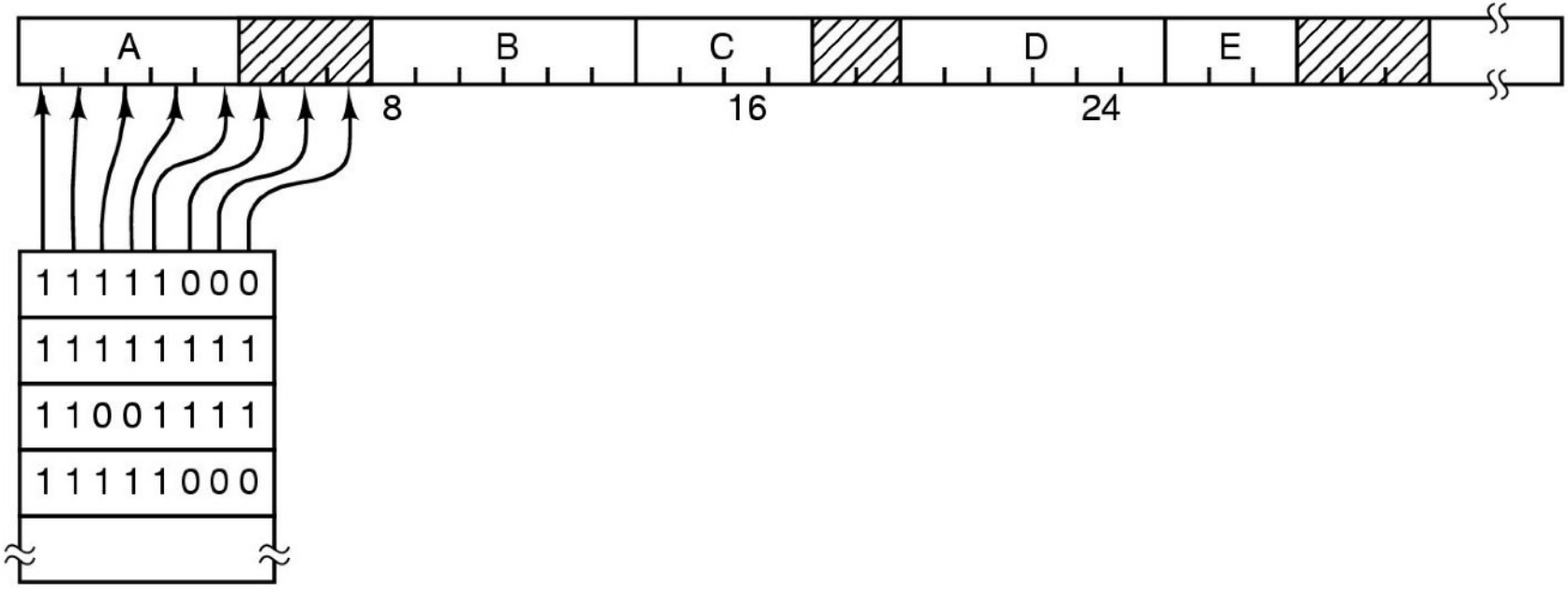
Managing memory with bit maps

- **Problem - how to keep track of used and unused memory?**
- **Technique 1 - Bit Maps**
  - A long bit string
  - One bit for every chunk of memory

    1 = in use

    0 = free
  - Size of allocation unit influences space required
    - **Example: unit size = 32 bits**
      - overhead for bit map: 1/33 = 3%
    - **Example: unit size = 4Kbytes**
      - overhead for bit map: 1/32,769

Managing memory with bit maps

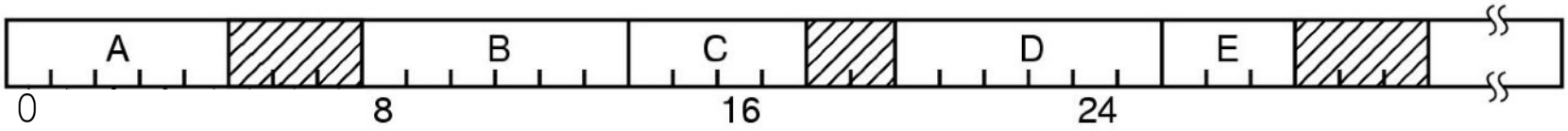


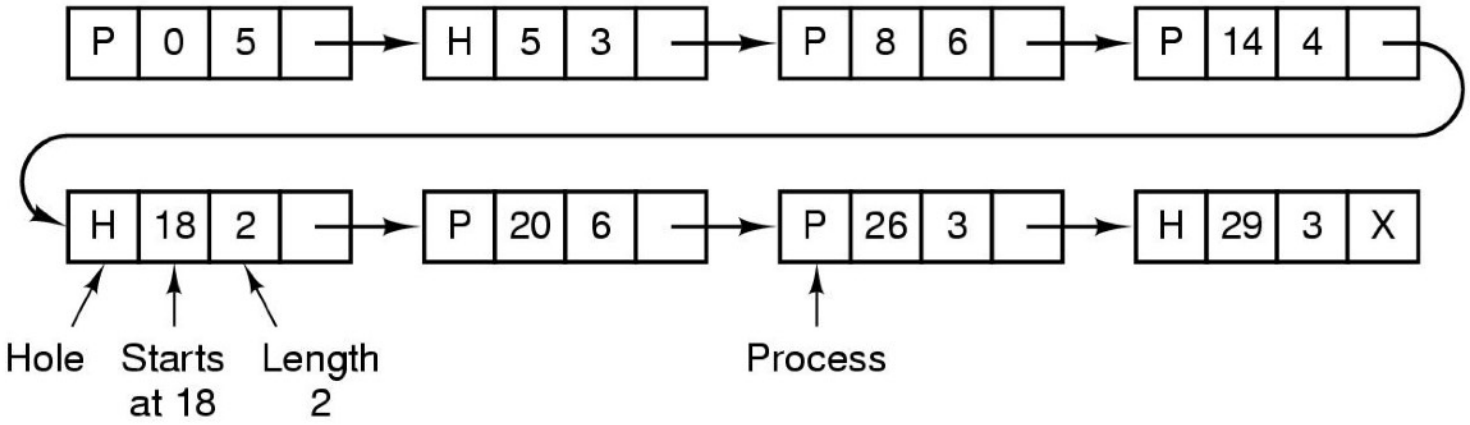
Managing memory with linked lists

- **Technique 2 - Linked List**

- **Keep a list of elements**
- **Each element describes one unit of memory**
  - Free / in-use Bit ("P=process, H=hole")
  - Starting address
  - Length
  - Pointer to next element

Managing memory with linked lists





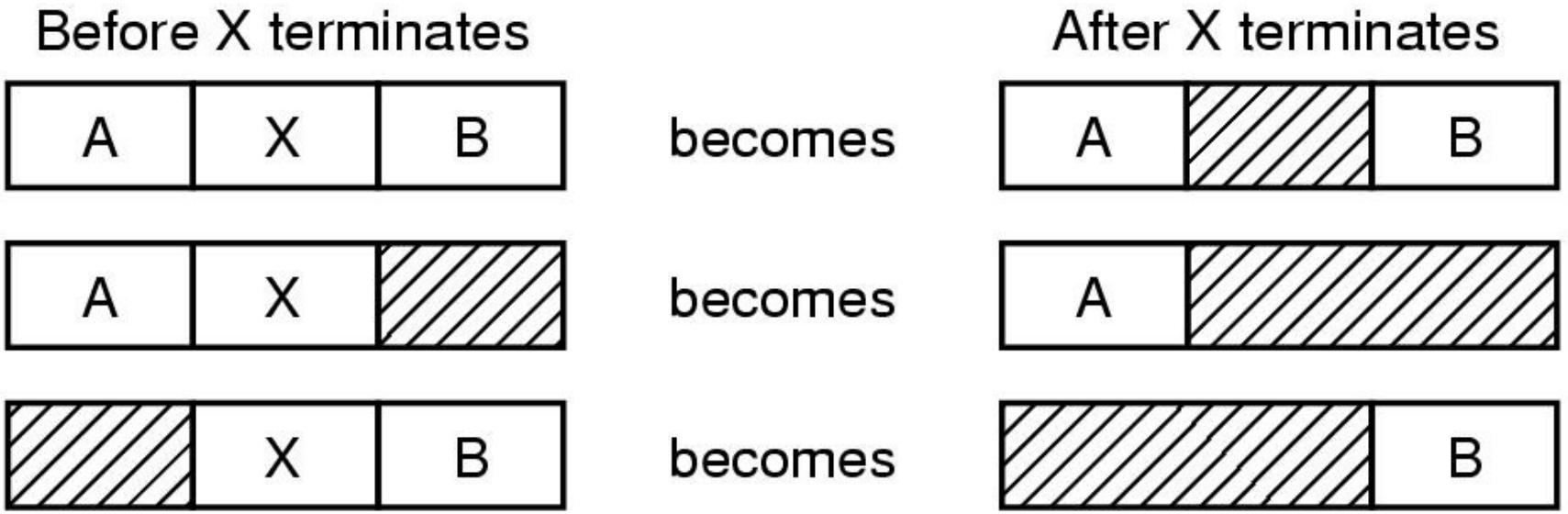
Merging holes

- Whenever a unit of memory is freed we want to merge adjacent holes!

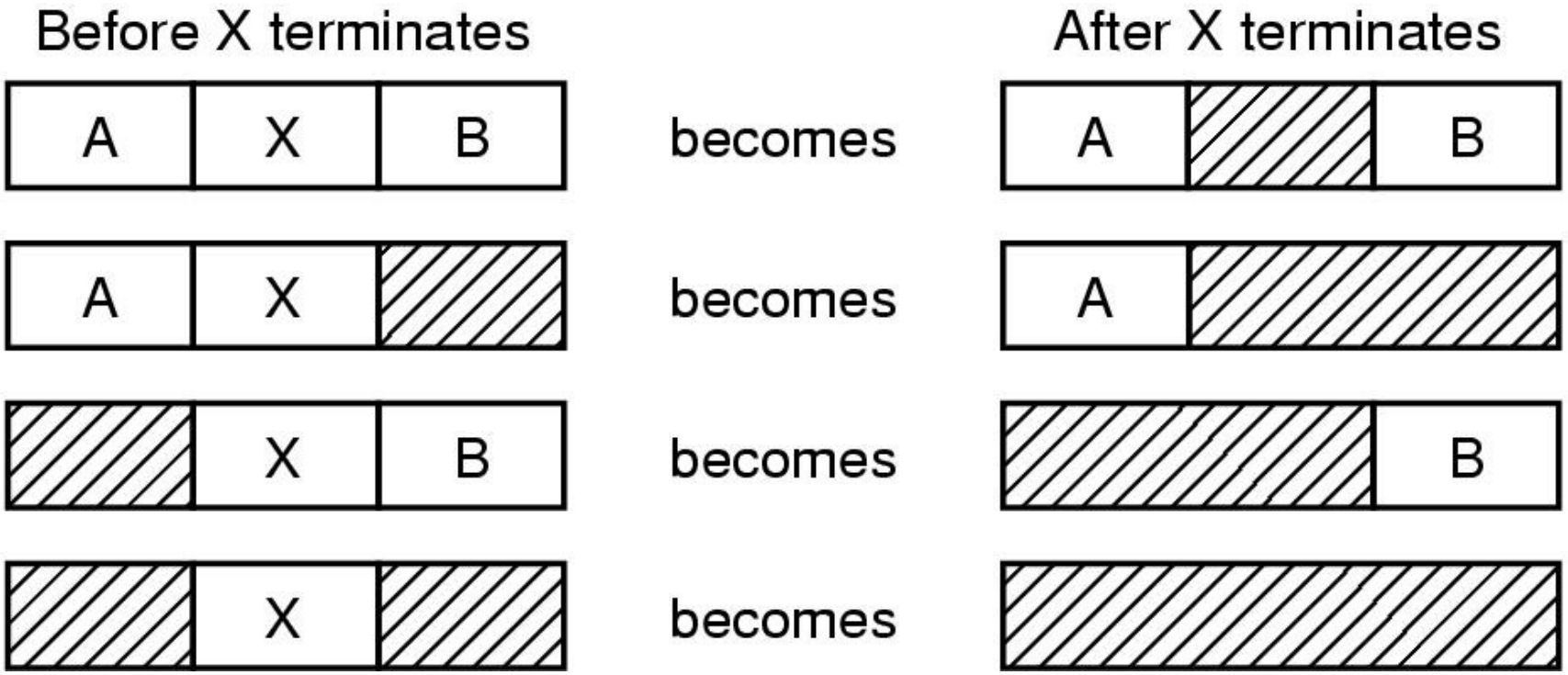
Merging holes

Merging holes

Merging holes



Merging holes



Managing memory with linked lists

- **Searching the list for space for a new process**
  - First Fit
  - Next Fit
    - **Start from current location in the list**
  - Best Fit
    - **Find the smallest hole that will work**
    - **Tends to create lots of really small holes**
  - Worst Fit
    - **Find the largest hole**
    - **Remainder will be big**
  - Quick Fit
    - **Keep separate lists for common sizes**

Back to Fragmentation

- **Memory is divided into partitions**
- **Each partition has a different size**
- **Processes are allocated space and later freed**
- **After a while memory will be full of small holes!**
  - No free space large enough for a new process even though there is enough free memory in total
- **If we allow free space within a partition we have internal fragmentation**
- **Fragmentation:**
  - External fragmentation = unused space between partitions
  - Internal fragmentation = unused space within partitions

Solution to fragmentation?

- **Compaction requires high copying overhead**
- **Why not allocate memory in non-contiguous equal fixed size units?**
  - no external fragmentation!
  - internal fragmentation < 1 unit per process
- **How big should the units be?**
  - The smaller the better for internal fragmentation
  - The larger the better for management overhead
- **The key challenge for this approach:**
  "How can we do secure dynamic address translation?"

Using pages for non-contiguous allocation

- **Memory divided into fixed size page frames**
  - Page frame size = 2^n bytes
  - Lowest n bits of an address specify byte offset in a page
- **But how do we associate page frames with processes?**
  - And how do we map memory addresses within a process to the correct memory byte in a page frame?
- **Solution – address translation**
  - Processes use virtual addresses
  - CPU uses physical addresses
  - hardware support for virtual to physical address translation

بسم الله الرحمن الرحیم

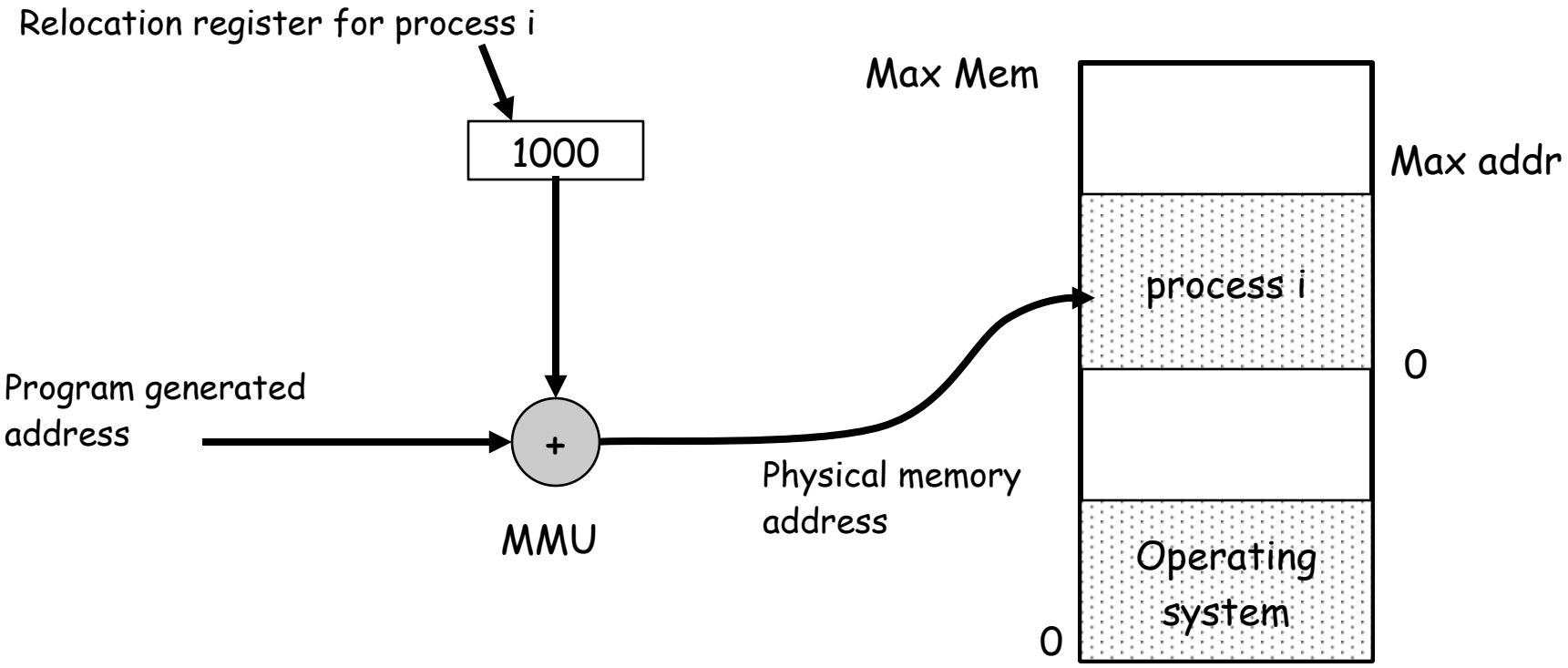
٤٠ «سیستم عامل»

جلسه ۱۴: مدیریت حافظه (۲)

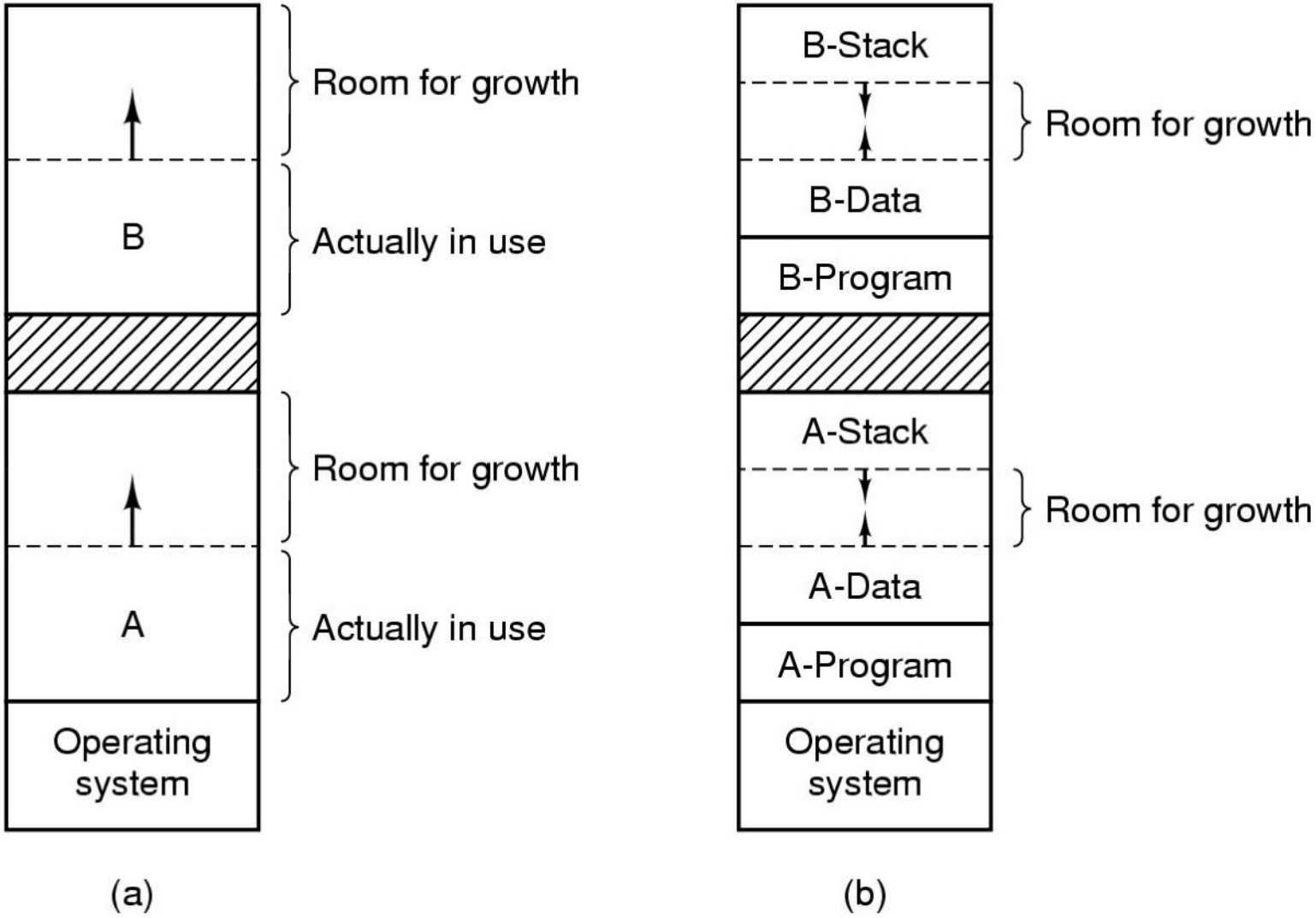
روش ۱: حافظه پیوسته

Dynamic relocation with a base register

- Memory Management Unit (MMU) - dynamically converts logical addresses into physical address
- MMU contains base address register for running process



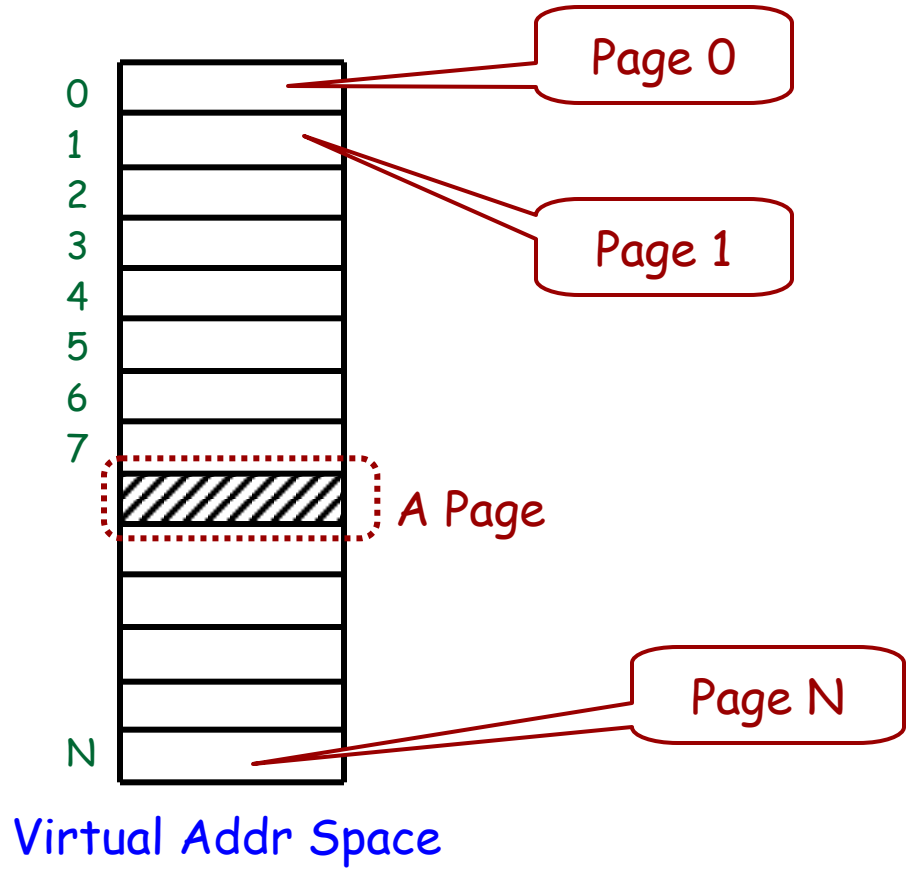
Allocating extra space within partitions



روش ۲: صفحه‌بندی

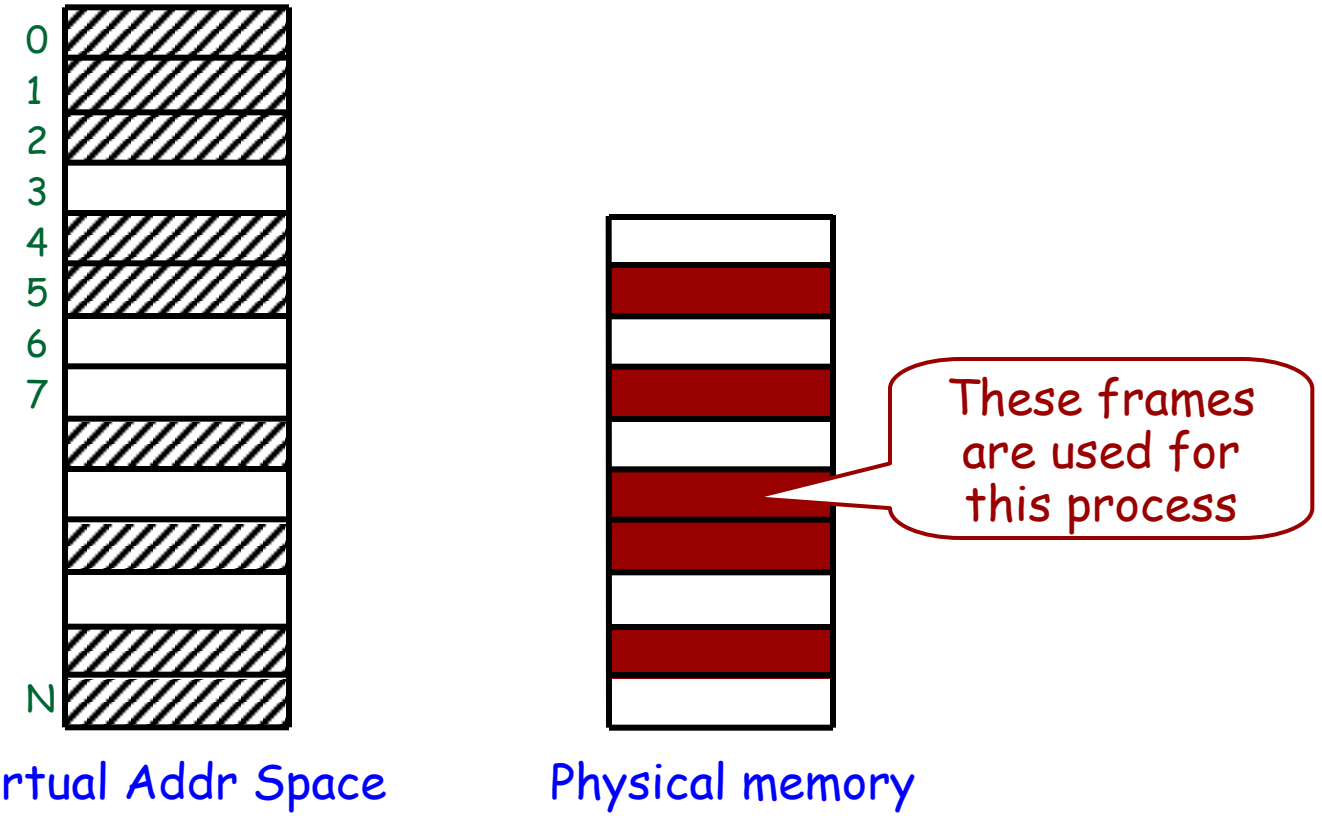
Virtual address spaces

- **The address space is divided into "pages"**
  - In BLITZ, the page size is 8K



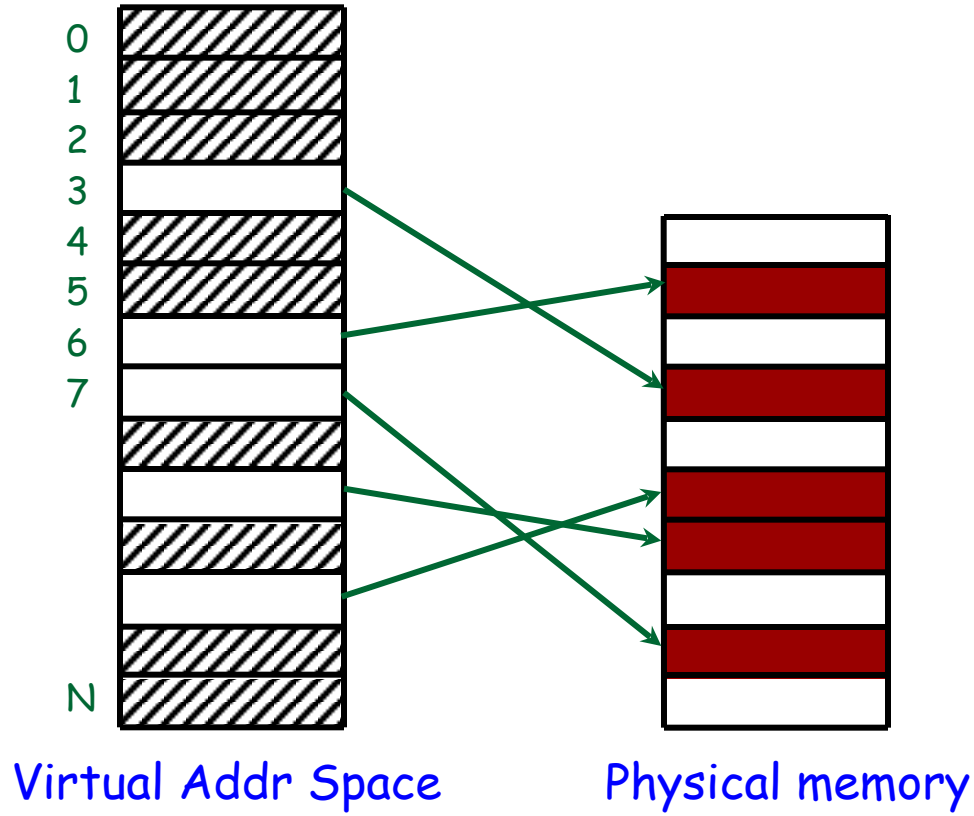
Virtual and physical address spaces

- Some frames are used to hold the pages of this process

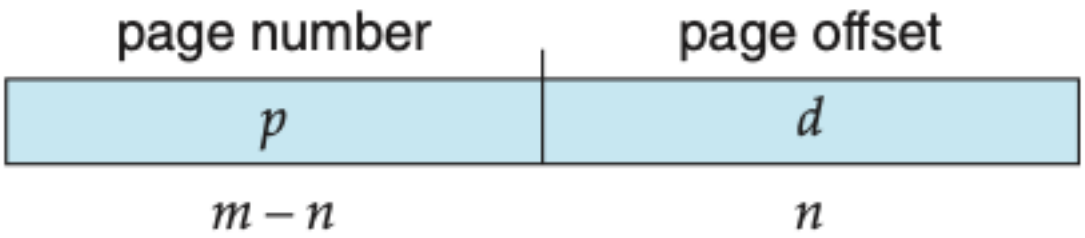


Page tables

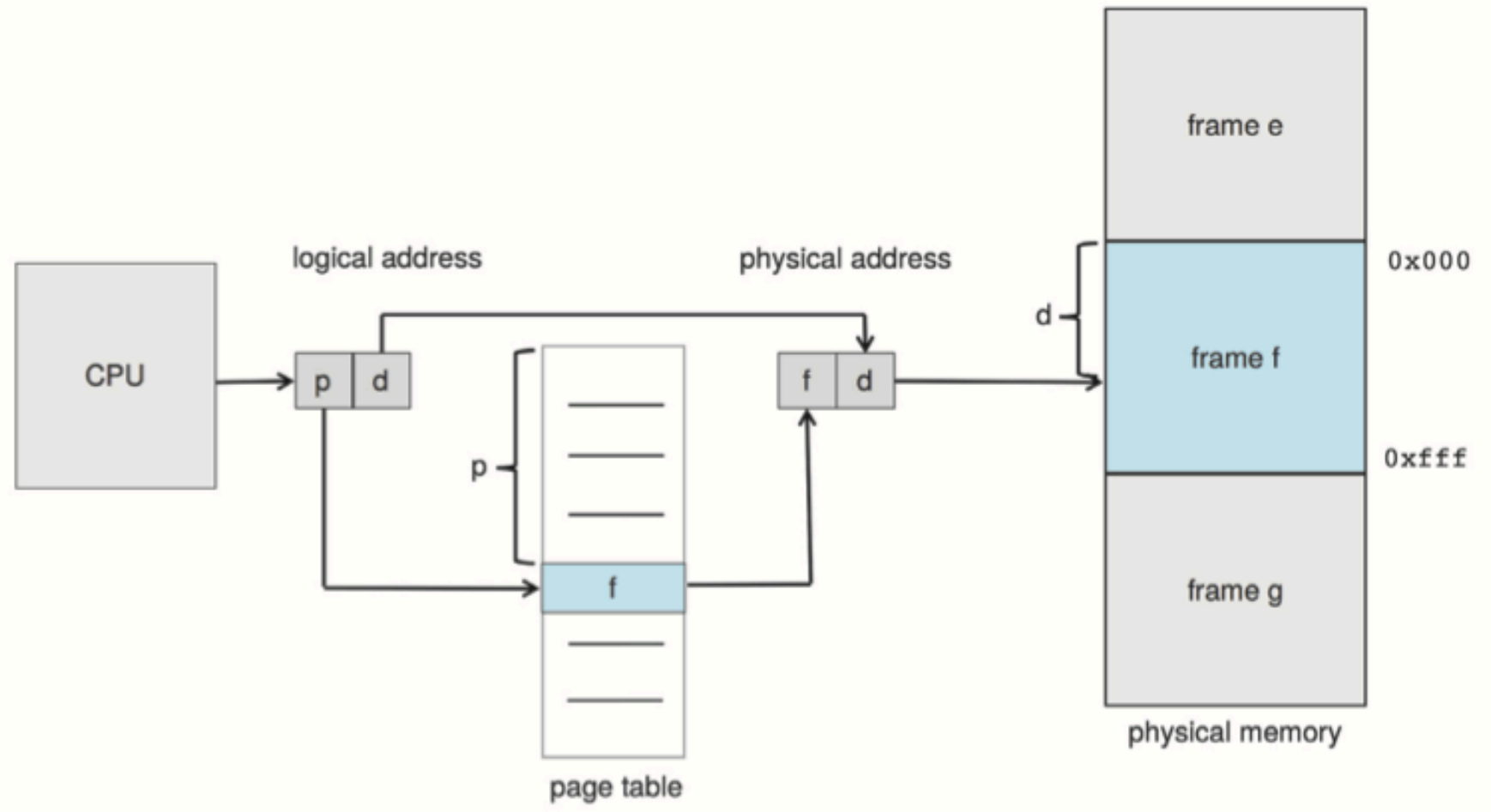
- Address mappings are stored in a page table in memory
- One page table entry per page...
  - Is this page in memory? If so, which frame is it in?




page number
page offset
p
d


page number
page offset
p
d
m − n
n

Logical Mapping



Address mappings and translation

- **Address mappings are stored in a page table in memory**
  - Typically one page table for each process
- **Address translation is done by hardware (ie the MMU)**
- **How does the MMU get the address mappings?**
  - Either the MMU holds the entire page table (too expensive)
    - **or it knows where it is in physical memory and goes there for every translation (too slow)**
  - Or the MMU holds a portion of the page table
    - **MMU caches page table entries**
    - **Cache is called a translation look-aside buffer (TLB)**
    - **… and knows how to deal with TLB misses**

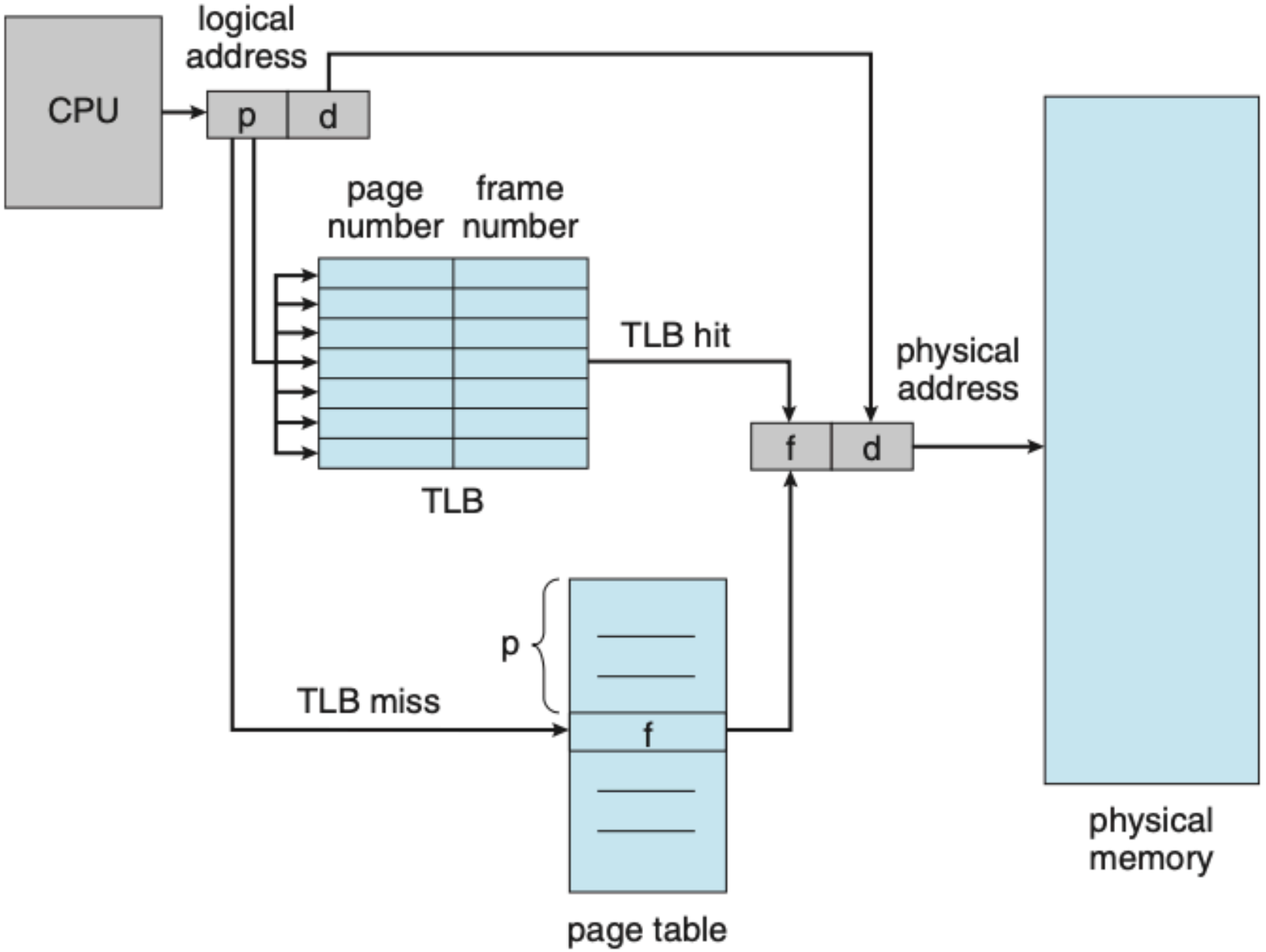


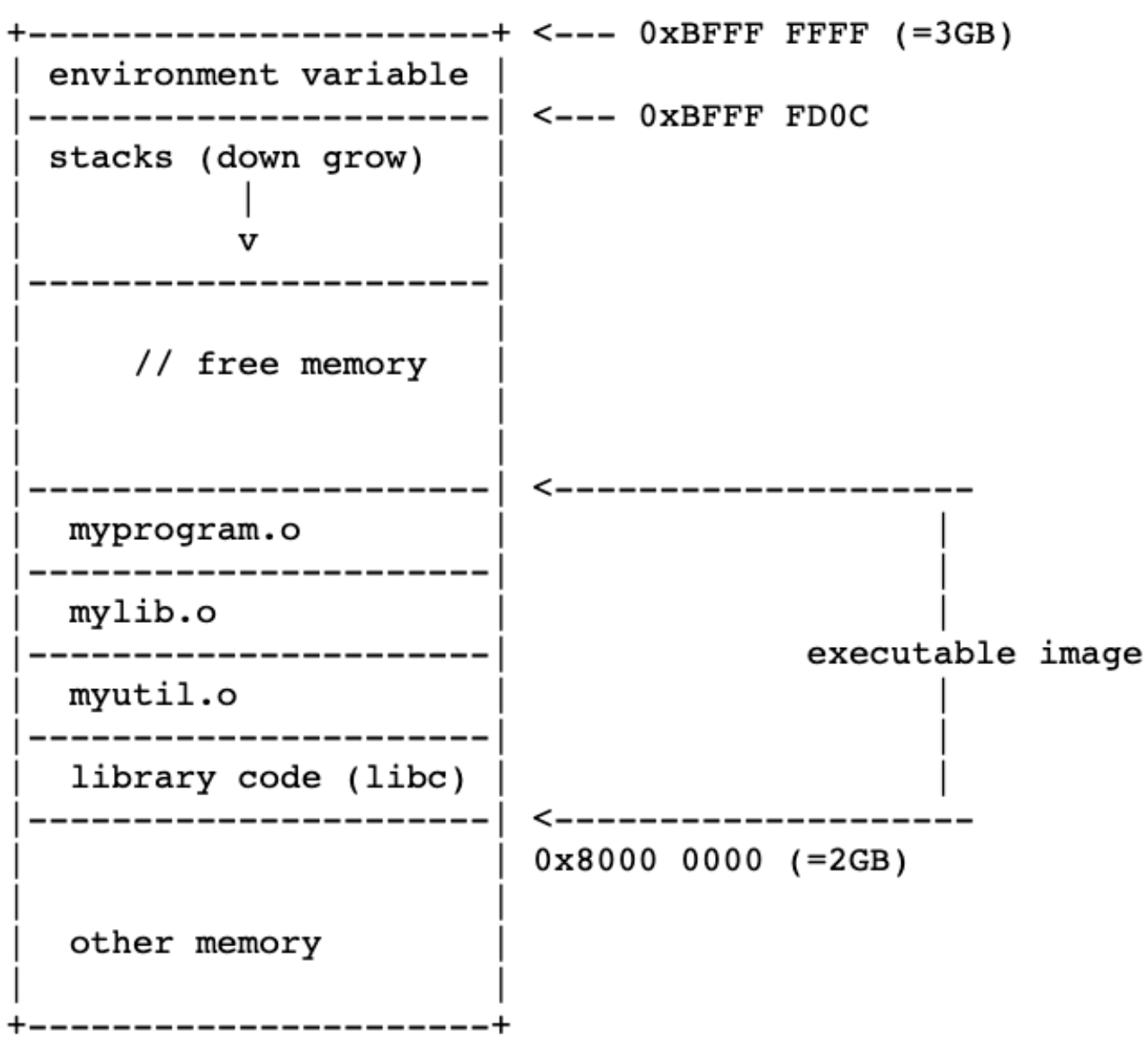
Figure 9.12 Paging hardware with TLB.

Address mappings and translation

- **What if the TLB needs a mapping it doesn't have?**
- **Software managed TLB**
  - it generates a TLB-miss fault which is handled by the operating system (like interrupt or trap handling)
  - The operating system looks in the page tables, gets the mapping from the right entry, and puts it in the TLB
- **Hardware managed TLB**
  - it looks in a pre-specified physical memory location for the appropriate entry in the page table
  - The hardware architecture defines where page tables must be stored in physical memory
    - **OS must load current process page table there on context switch!**

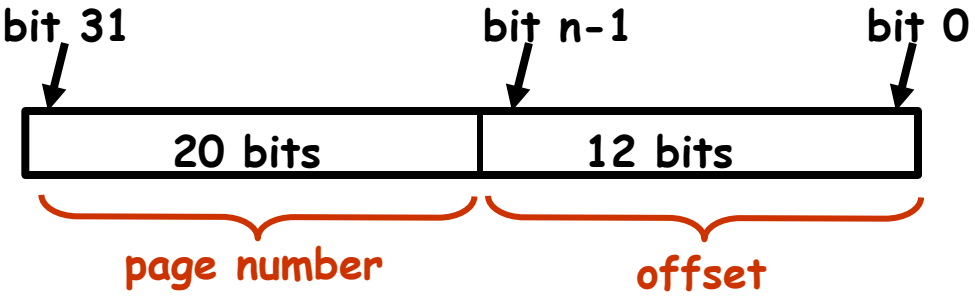
In real world #2

From a user process point of view



Virtual addresses

- **Virtual memory addresses (what the process uses)**
  - Page number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page number



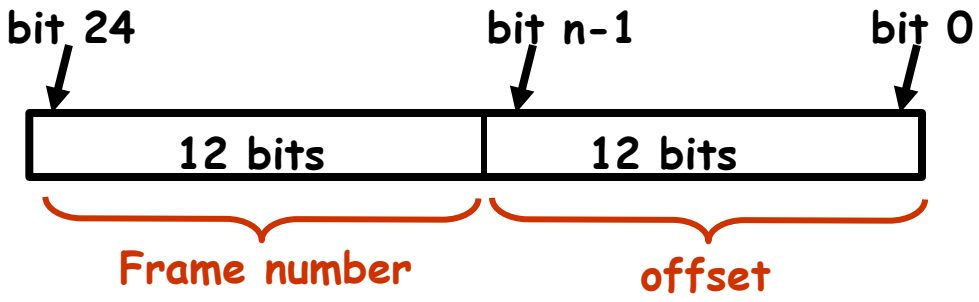
Example: 32 bit virtual address

Page size = 2^{12} = 4KB

Address space size = 2^{32} bytes = 4GB

Physical addresses

- **Physical memory addresses (what memory uses)**
  - Frame number plus byte offset in frame
  - Low order n bits are the byte offset
  - Remaining high order bits are the frame number



Example: 24 bit physical address

Frame size = 2^{12} = 4KB

Max physical memory size = 2^{24} bytes = 16MB

Address translation

- **Complete set of address mappings for a process are stored in a page table in memory**
  - But accessing the table for every address translation is too expensive
  - So hardware support is used to map page numbers to frame numbers at full CPU speed
    - Memory management unit (MMU) has multiple registers for multiple pages and knows how to access page tables
    - Also called a translation look aside buffer (TLB)
    - Essentially a cache of page table entries

بسم الله الرحمن الرحیم

۵۷

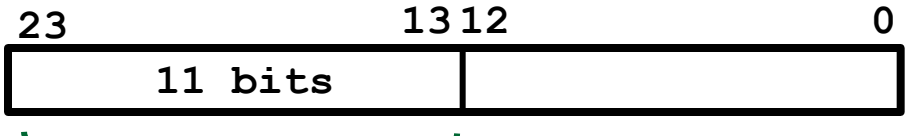
«سیستم عامل»

جلسه ۱۵: مدیریت حافظه (۳)

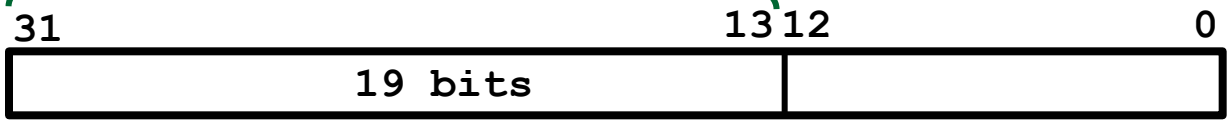
The BLITZ architecture

- The page table mapping:
  - Page --> Frame

- Virtual Address (24 bit in Blitz):



- Physical Address (32 bit in Blitz):



The BLITZ page table

- **An array of "page table entries"**
  - Kept in memory
- **2^{11} pages in a virtual address space**
  - ---> 2K entries in the table
- **Each entry is 4 bytes long**
  - 19 bits The Frame Number
  - 1 bit Valid Bit
  - 1 bit Writable Bit
  - 1 bit Dirty Bit
  - 1 bit Referenced Bit
  - 9 bits Unused (and available for OS algorithms)

The BLITZ page table

- **Two page table related registers in the CPU**
  - Page Table Base Register
  - Page Table Length Register
- **These define the page table for the "current" process**
  - Must be saved and restored on process context switch
- **Bits in the CPU "status register"**

  "System Mode"

  "Interrupts Enabled"

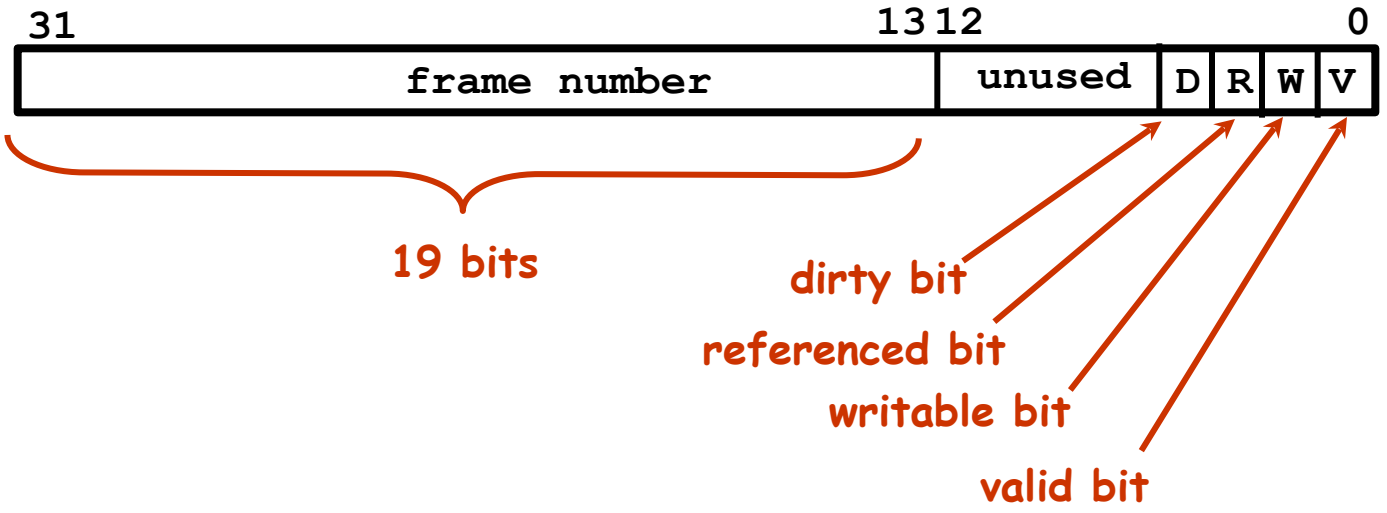
  "Paging Enabled"

  1 = Perform page table translation for every memory access

  0 = Do not do translation

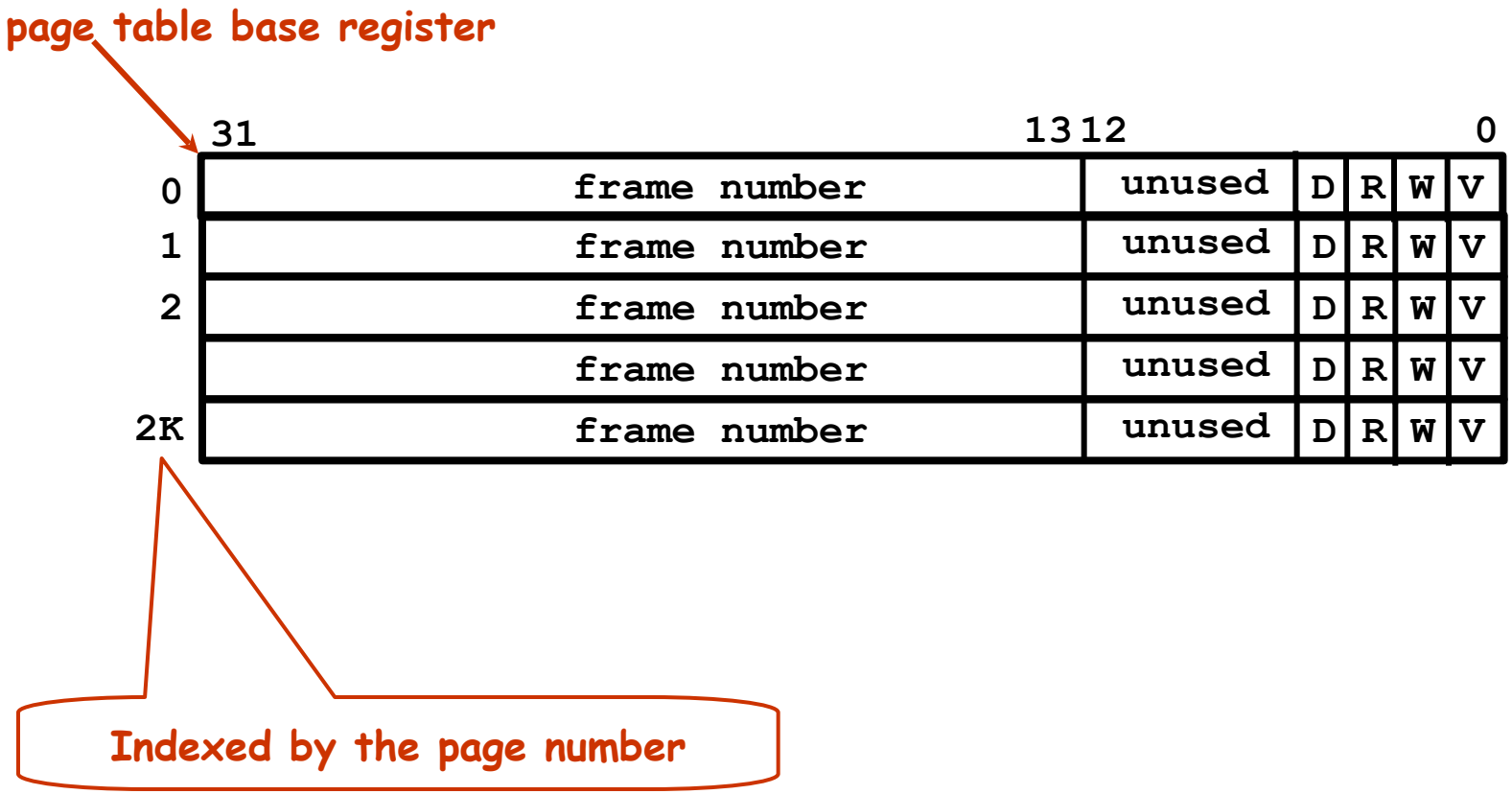
The BLITZ page table

- A page table entry

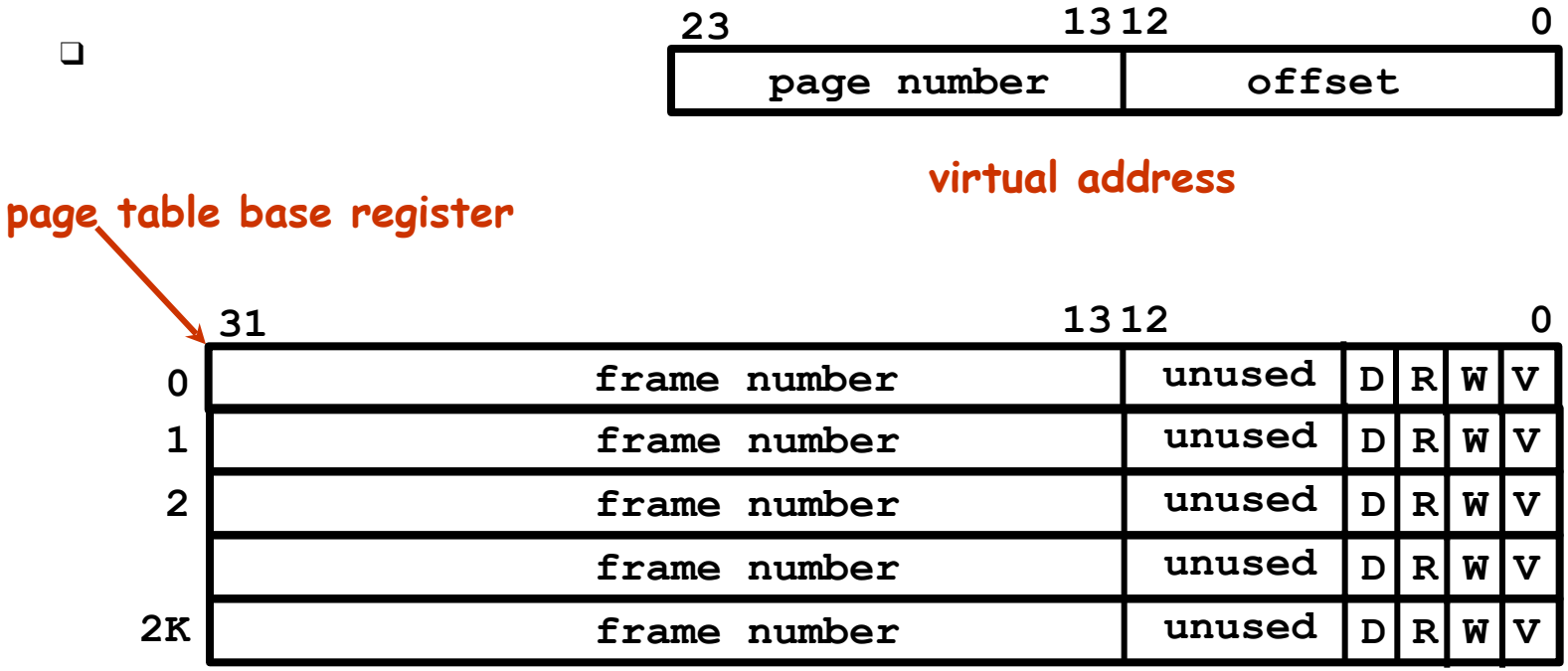


The BLITZ page table

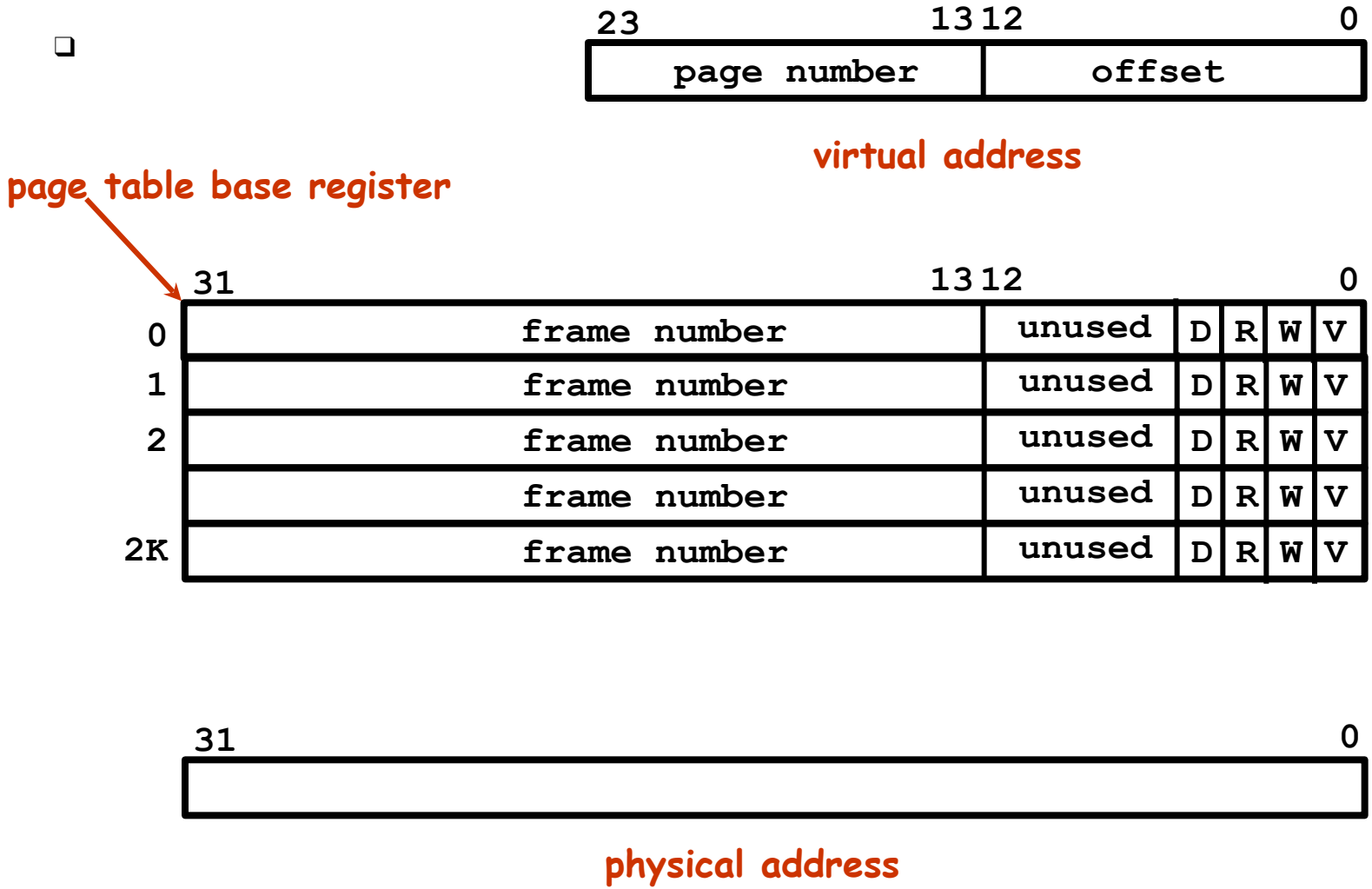
- The full page table



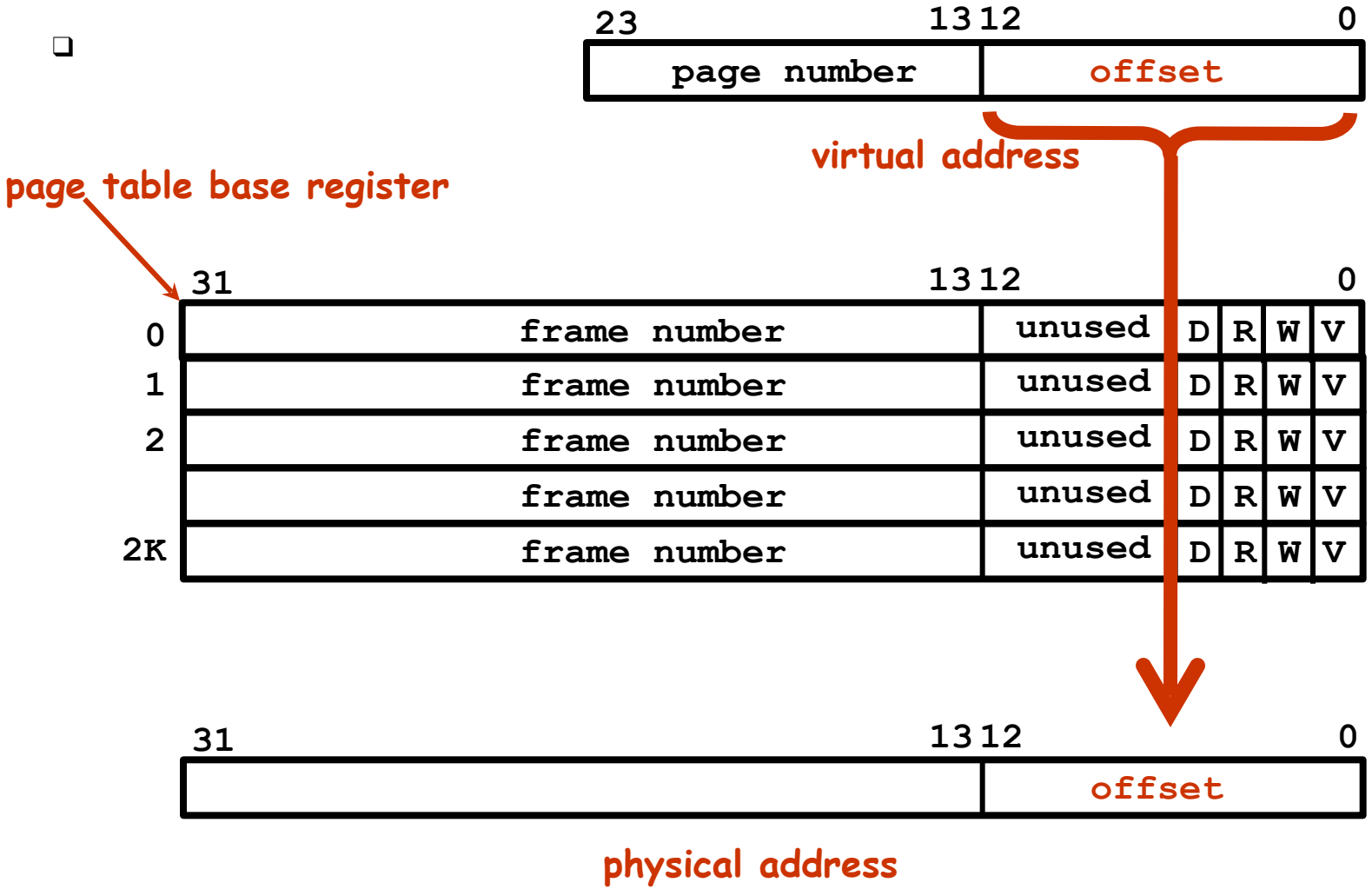
The BLITZ page table



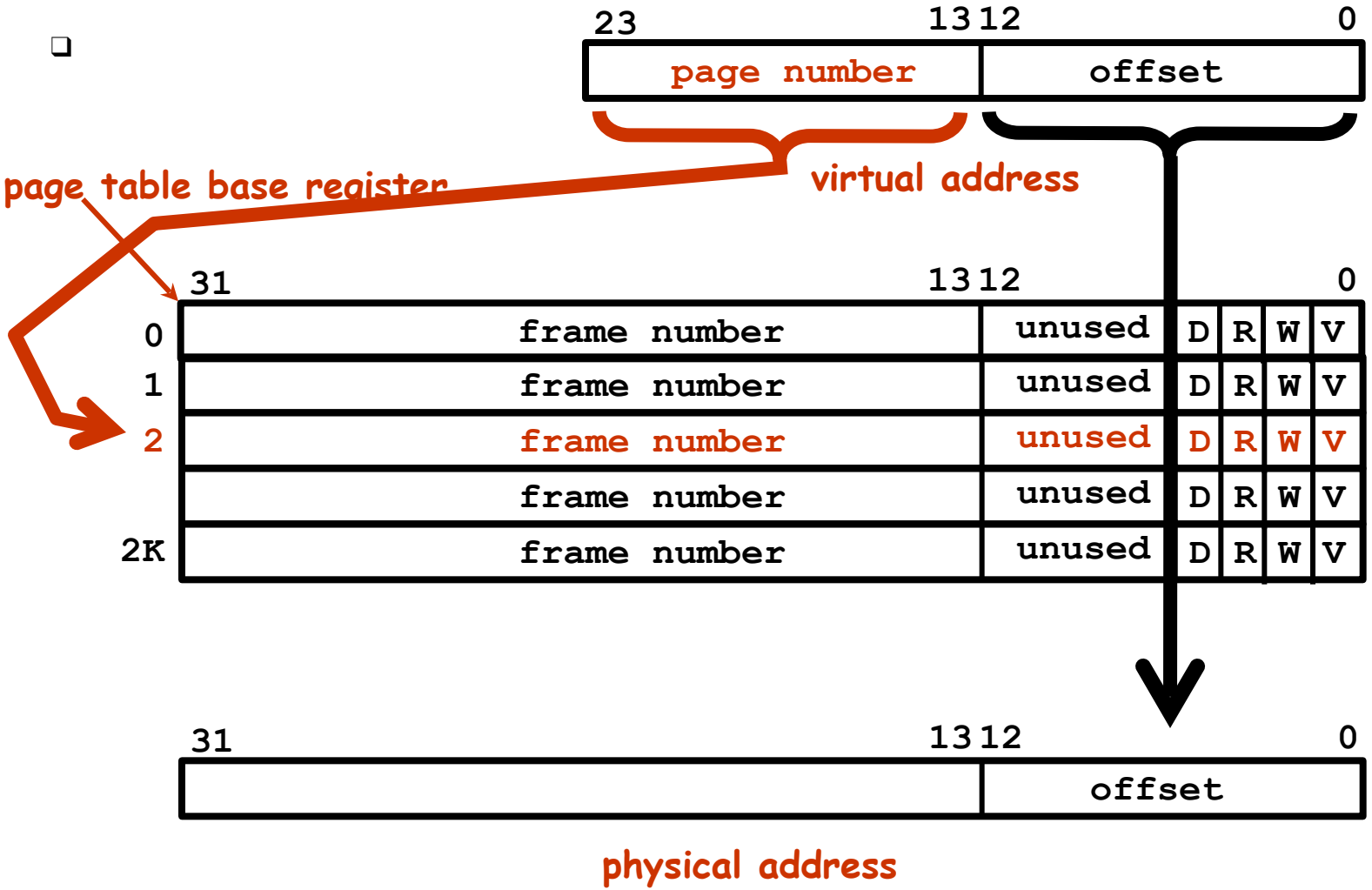
The BLITZ page table



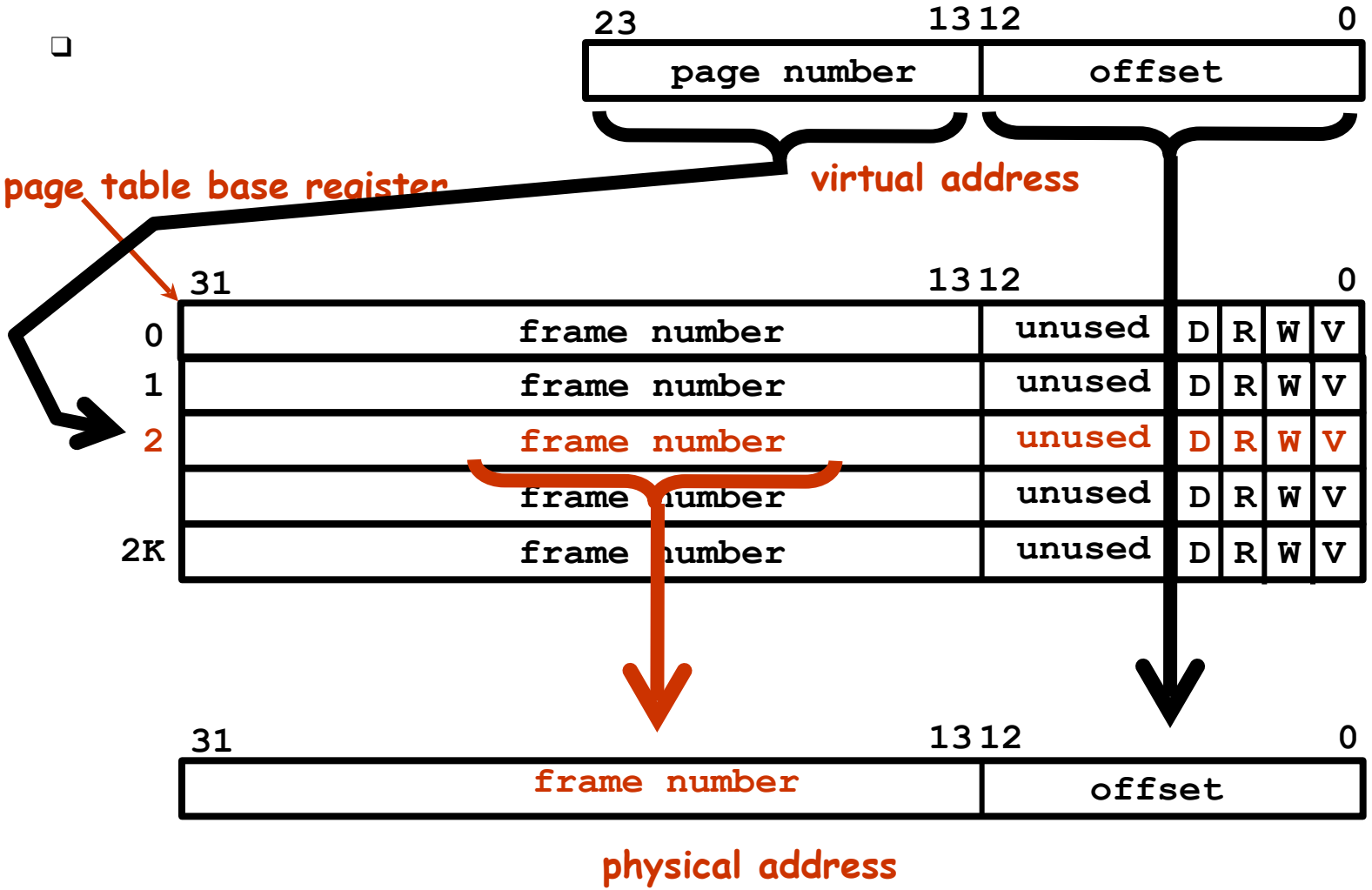
The BLITZ page table



The BLITZ page table



The BLITZ page table



Page tables

- **When and why do we access a page table?**
  - On every instruction to translate virtual to physical addresses?

Page tables

- **When and why do we access a page table?**
  - On every instruction to translate virtual to physical addresses?
  - In Blitz, YES, but in real machines NO!
  - In real machines it is only accessed
    - On TLB miss faults to refill the TLB
    - During process creation and destruction
    - When a process allocates or frees memory?

Translation Lookaside Buffer (TLB)

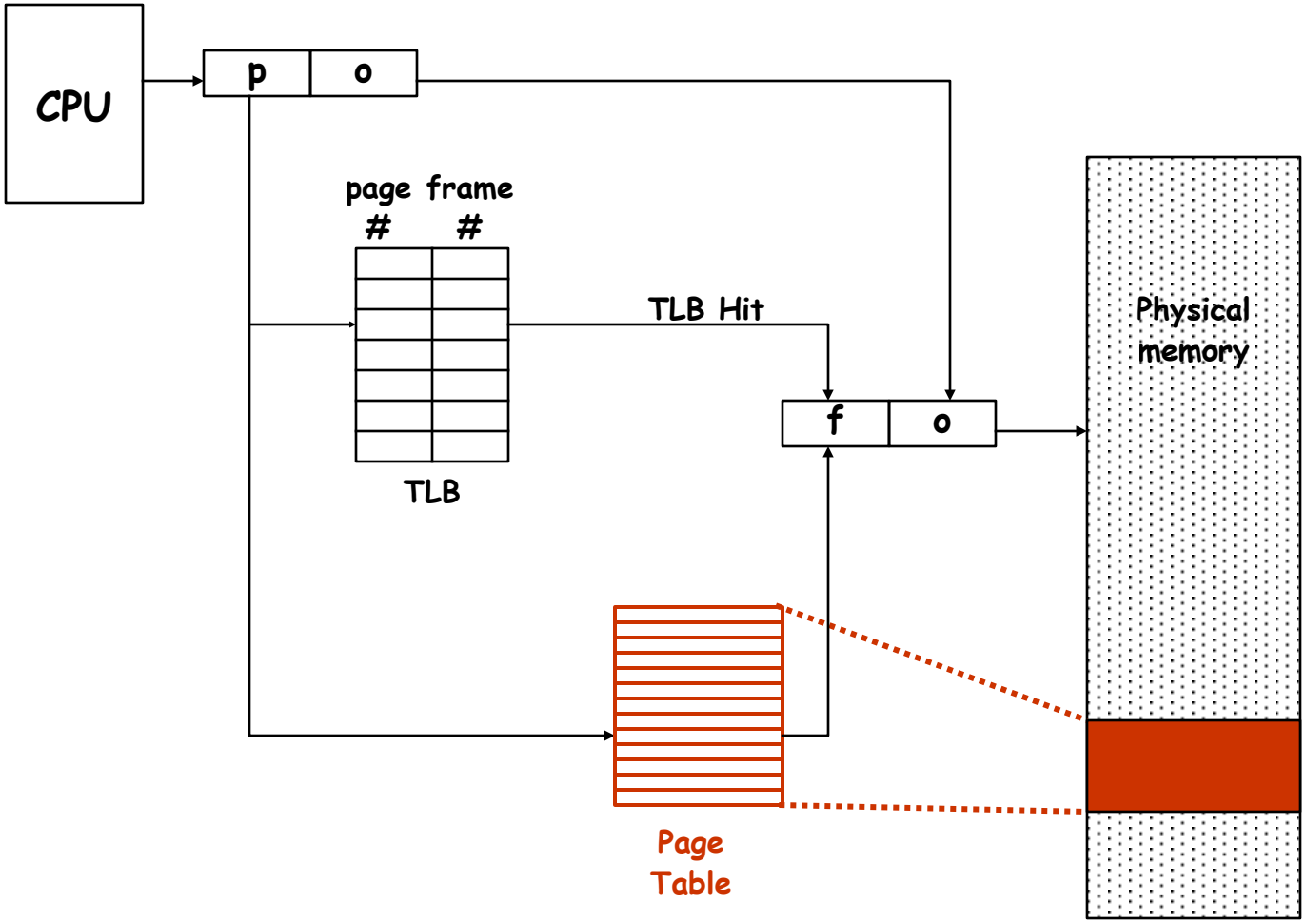
- **Problem:**
  - MMU can't keep up with the CPU if it goes to the page table on every memory access!

Translation Lookaside Buffer (TLB)

- **Problem:**
  - MMU can't keep up with the CPU if it goes to the page table on every memory access!

- **Solution:**
  - Cache the page table entries in a hardware cache
  - Small number of entries (e.g., 64)
  - Each entry contains
    - **Page number**
    - **Other stuff from page table entry**
  - Associatively indexed on page number
    - **ie. You can do a lookup in a single cycle**

Translation lookaside buffer



Hardware operation of TLB

-

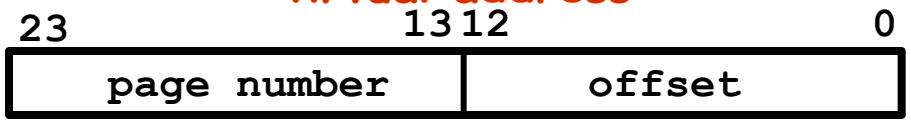
Key

Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

Hardware operation of TLB

-

virtual address

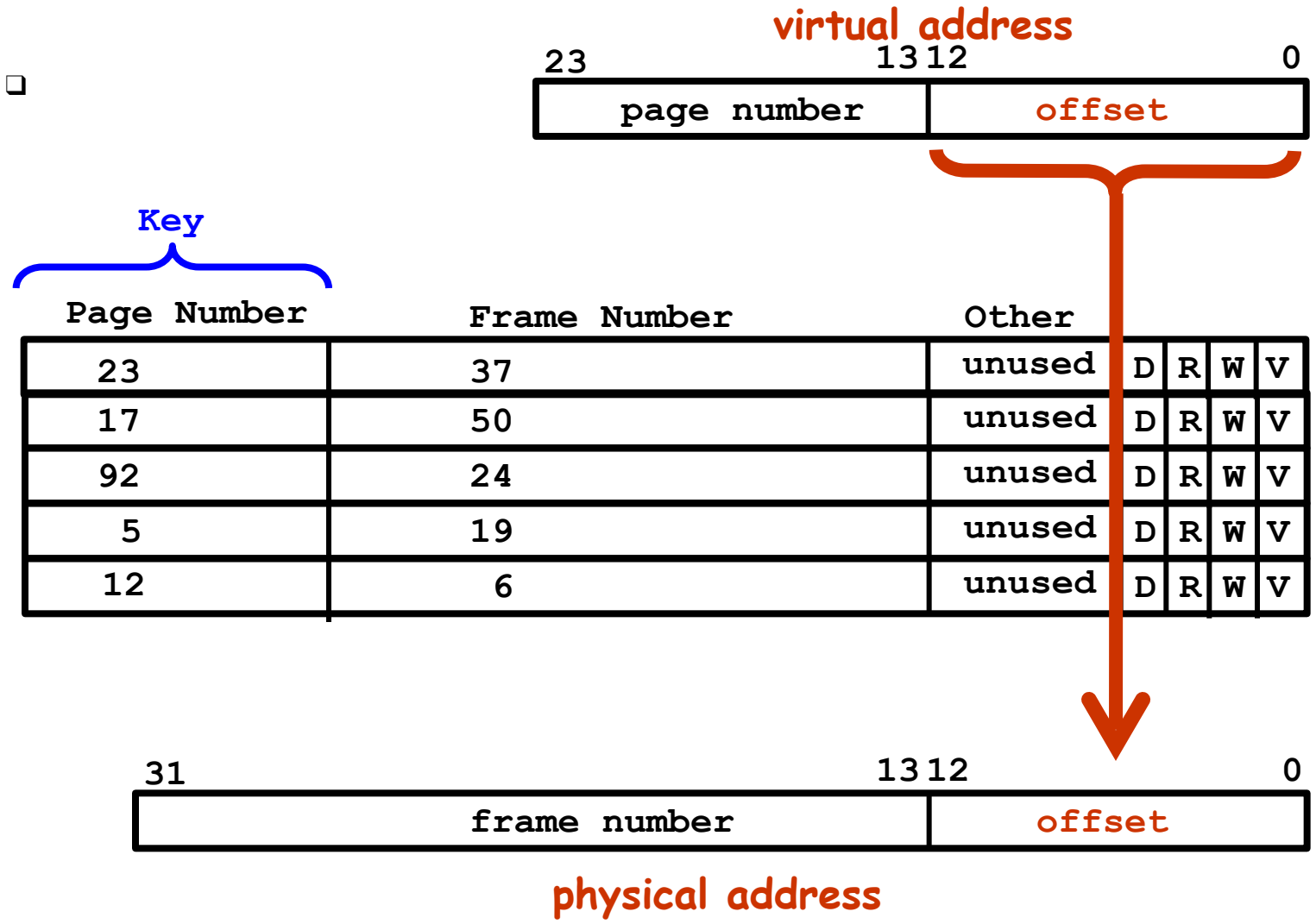


Key

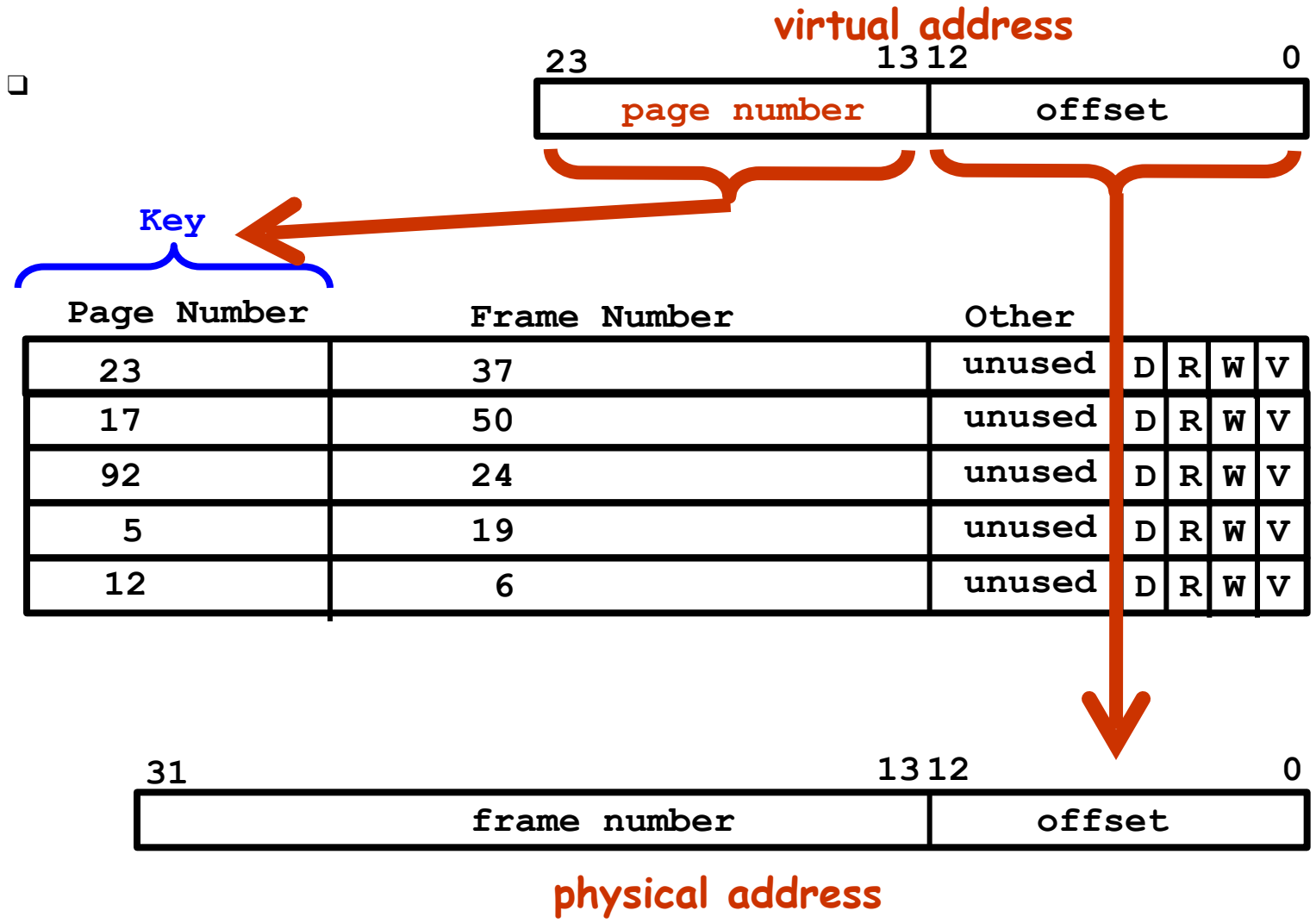
Page Number	Frame Number	Other	D	R	W	V
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

physical address

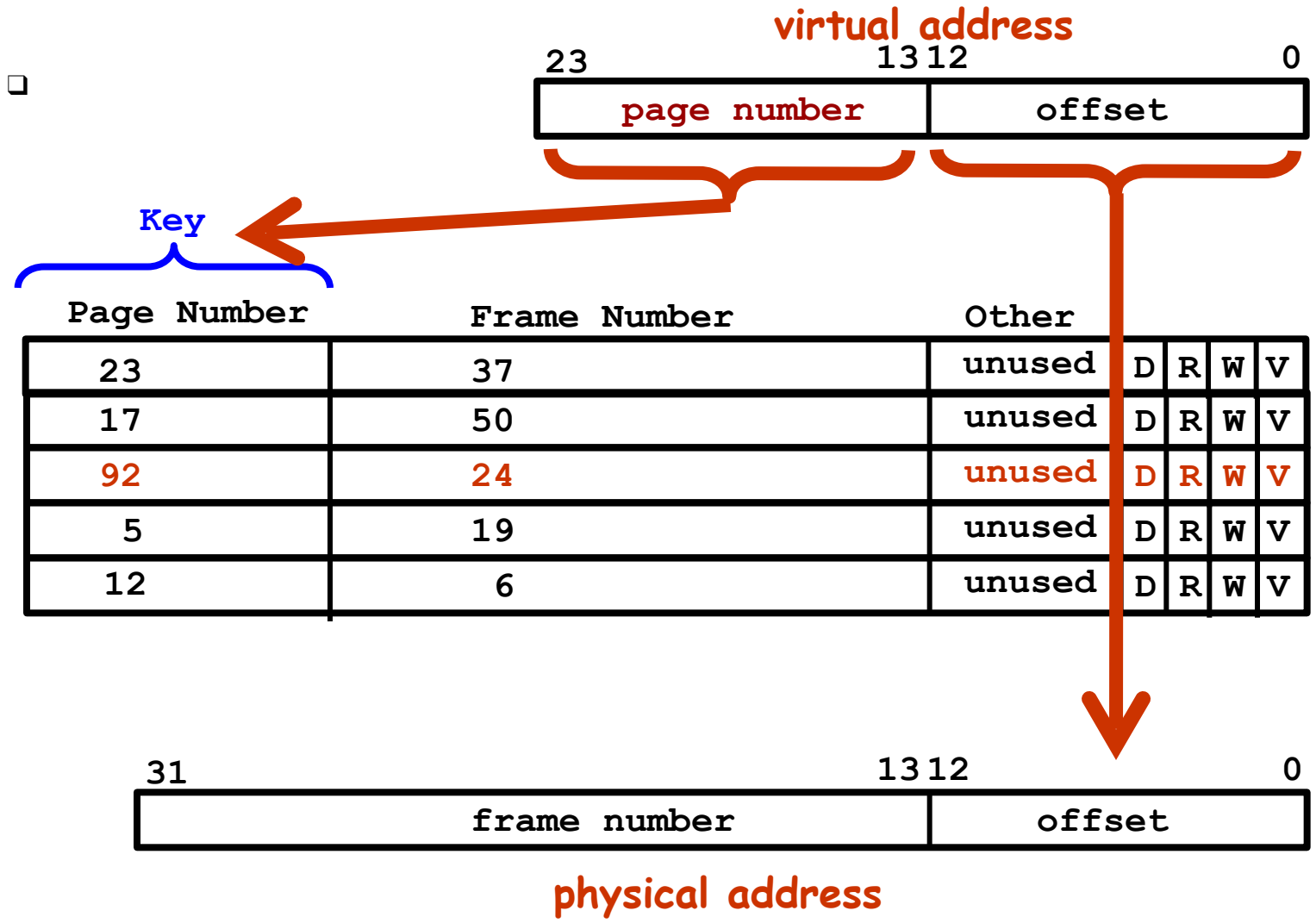
Hardware operation of TLB



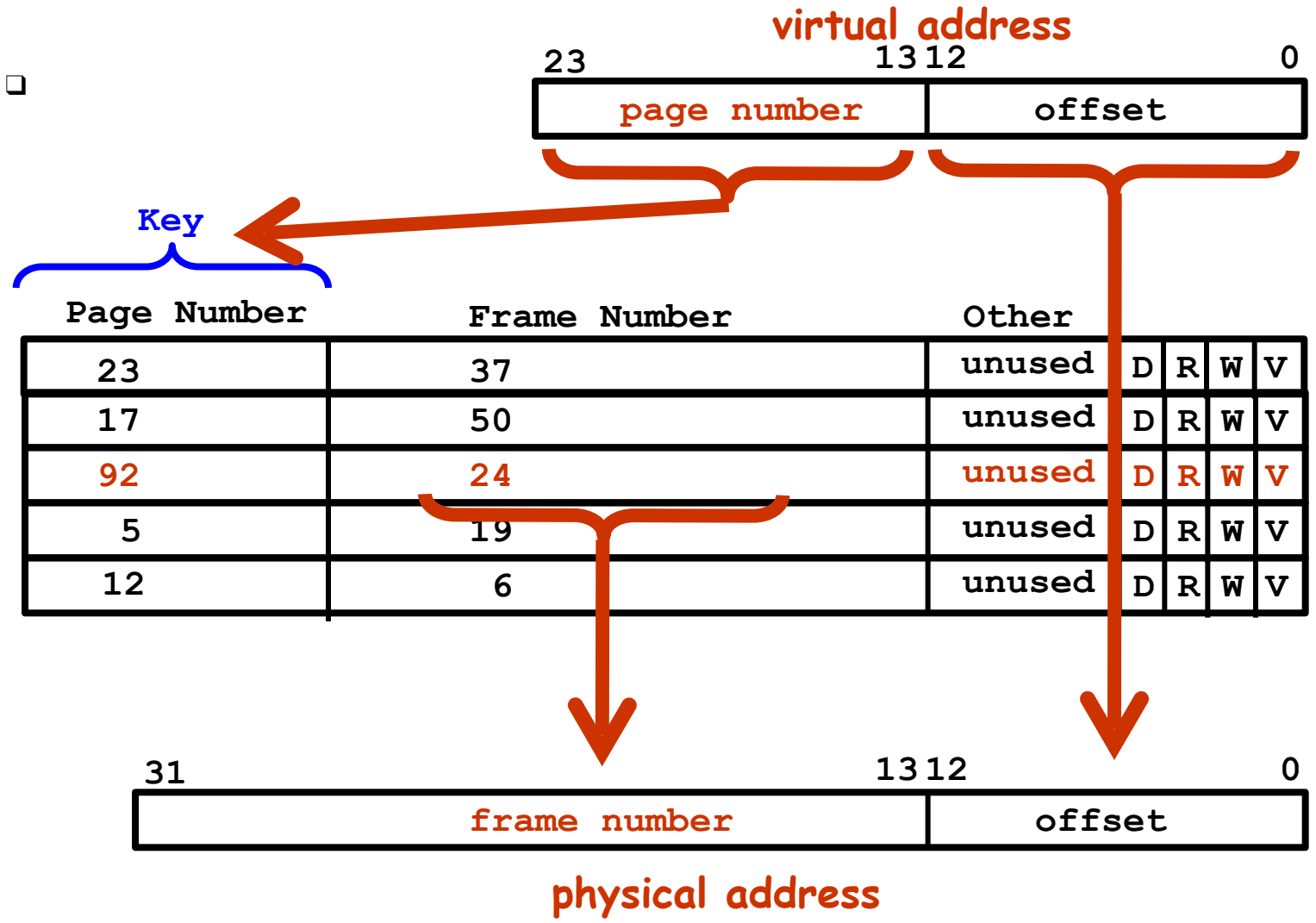
Hardware operation of TLB



Hardware operation of TLB



Hardware operation of TLB



Software operation of TLB

- **What if the entry is not in the TLB?**
  - Go look in the page table in memory
  - Find the right entry
  - Move it into the TLB
  - But which TLB entry should be replaced?

Software operation of TLB

- **Hardware TLB refill**
  - Page tables in specific location and format
  - TLB hardware handles its own misses
  - Replacement policy fixed by hardware

- **Software refill**
  - Hardware generates trap (TLB miss fault)
  - Lets the OS deal with the problem
  - Page tables become entirely a OS data structure!
  - Replacement policy managed in software

Software operation of TLB

- **What should we do with the TLB on a context switch?**
- **How can we prevent the next process from using the last process's address mappings?**
  - Option 1: empty the TLB
    - **New process will generate faults until its pulls enough of its own entries into the TLB**
  - Option 2: just clear the "Valid Bit"
    - **New process will generate faults until its pulls enough of its own entries into the TLB**
  - Option 3: the hardware maintains a process id tag on each TLB entry
    - **Hardware compares this to a process id held in a specific register … on every translation**

Page tables

- Do we access a page table when a process allocates or frees memory?

Page tables

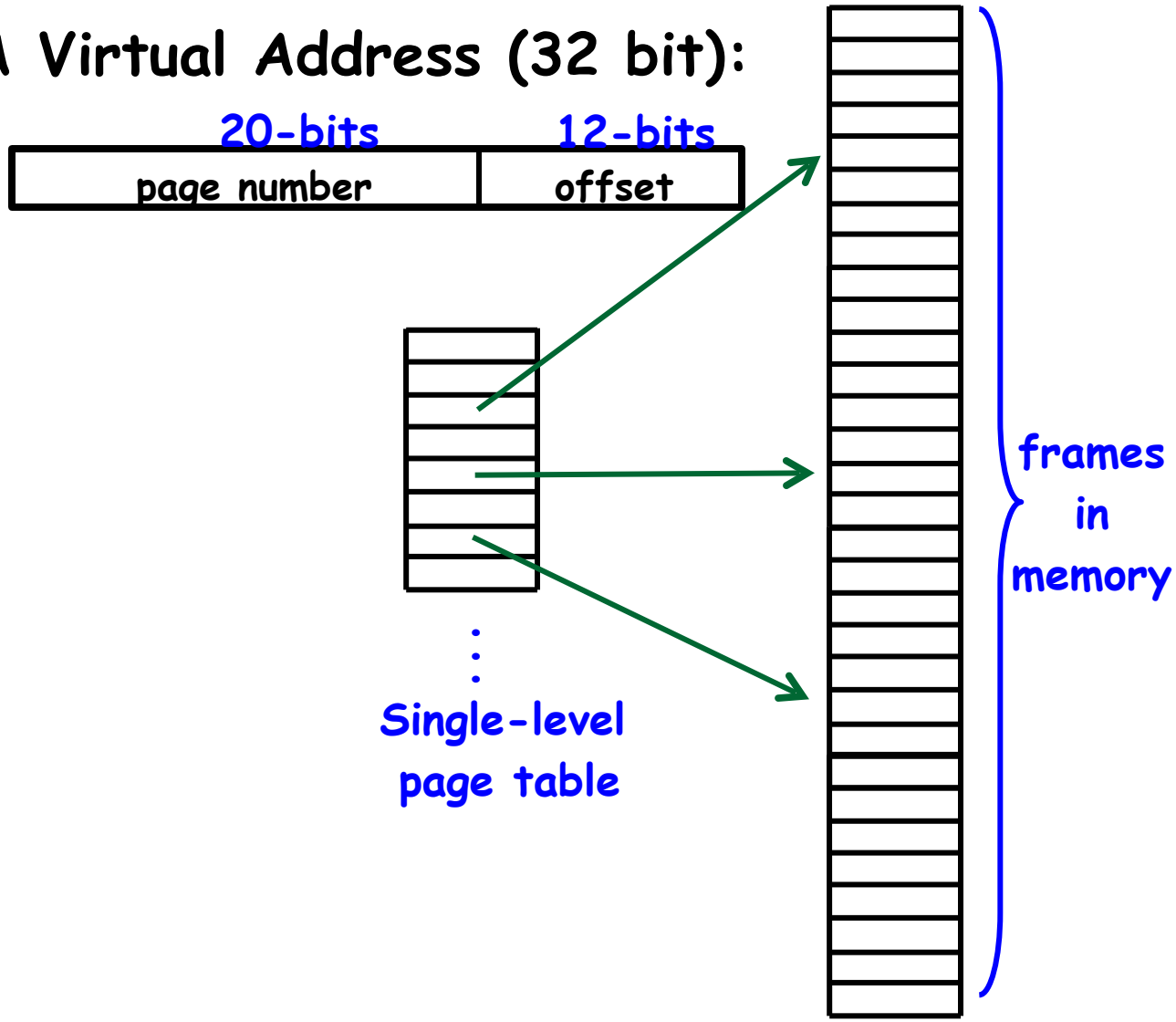
- **Do we access a page table when a process allocates or frees memory?**
  - Not necessarily
  - Library routines (malloc) can service small requests from a pool of free memory already allocated within a process address space
  - When these routines run out of space a new page must be allocated and its entry inserted into the page table
    - **This allocation is requested using a system call**

Page table design issues

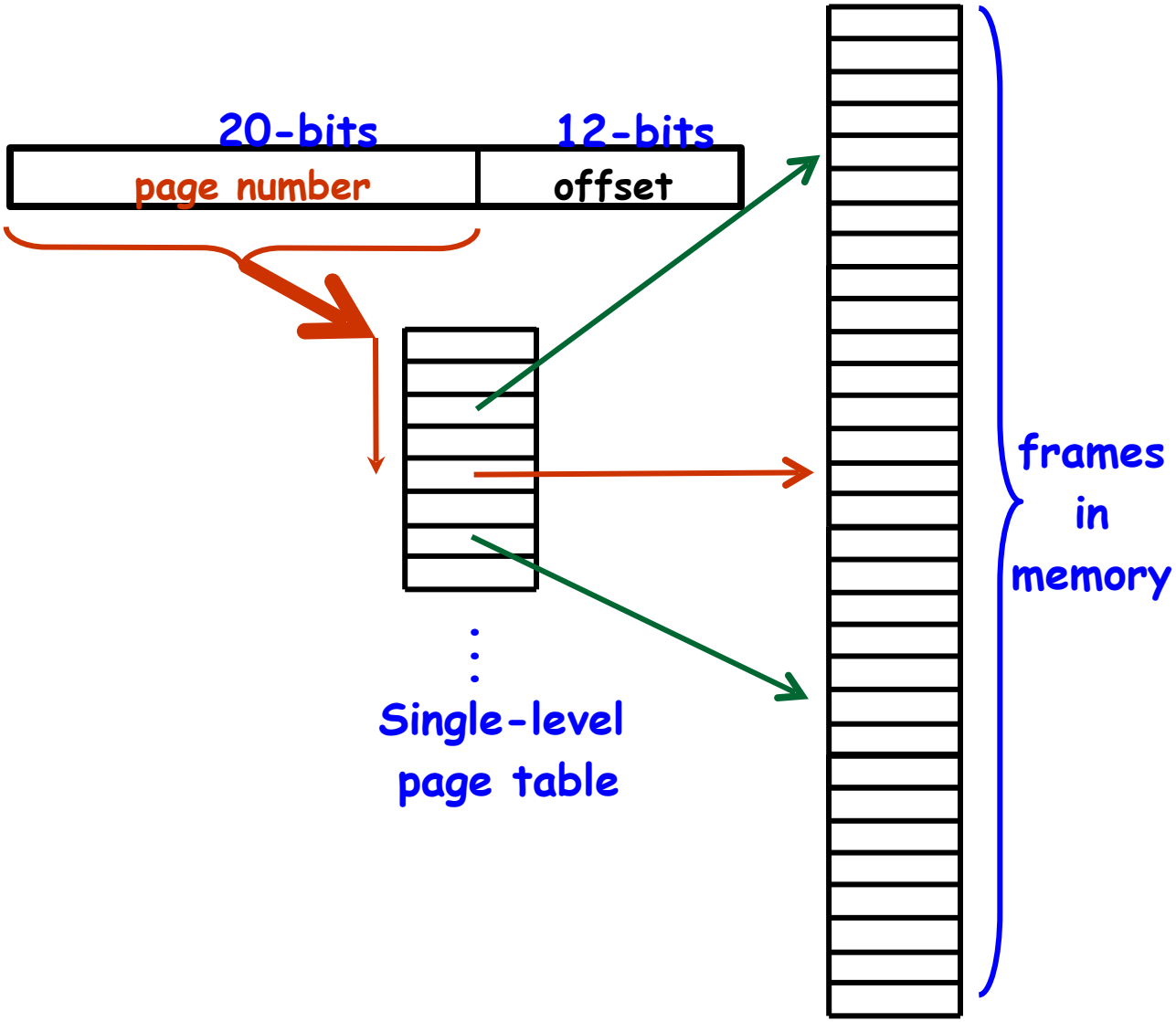
- **Page table size depends on**
  - Page size
  - Virtual address length
- **Memory used for page tables is overhead!**
  - How can we save space?
  - … and still find entries quickly?
- **Three options**
  - Single-level page tables
  - Multi-level page tables
  - Inverted page tables

Single-level page tables

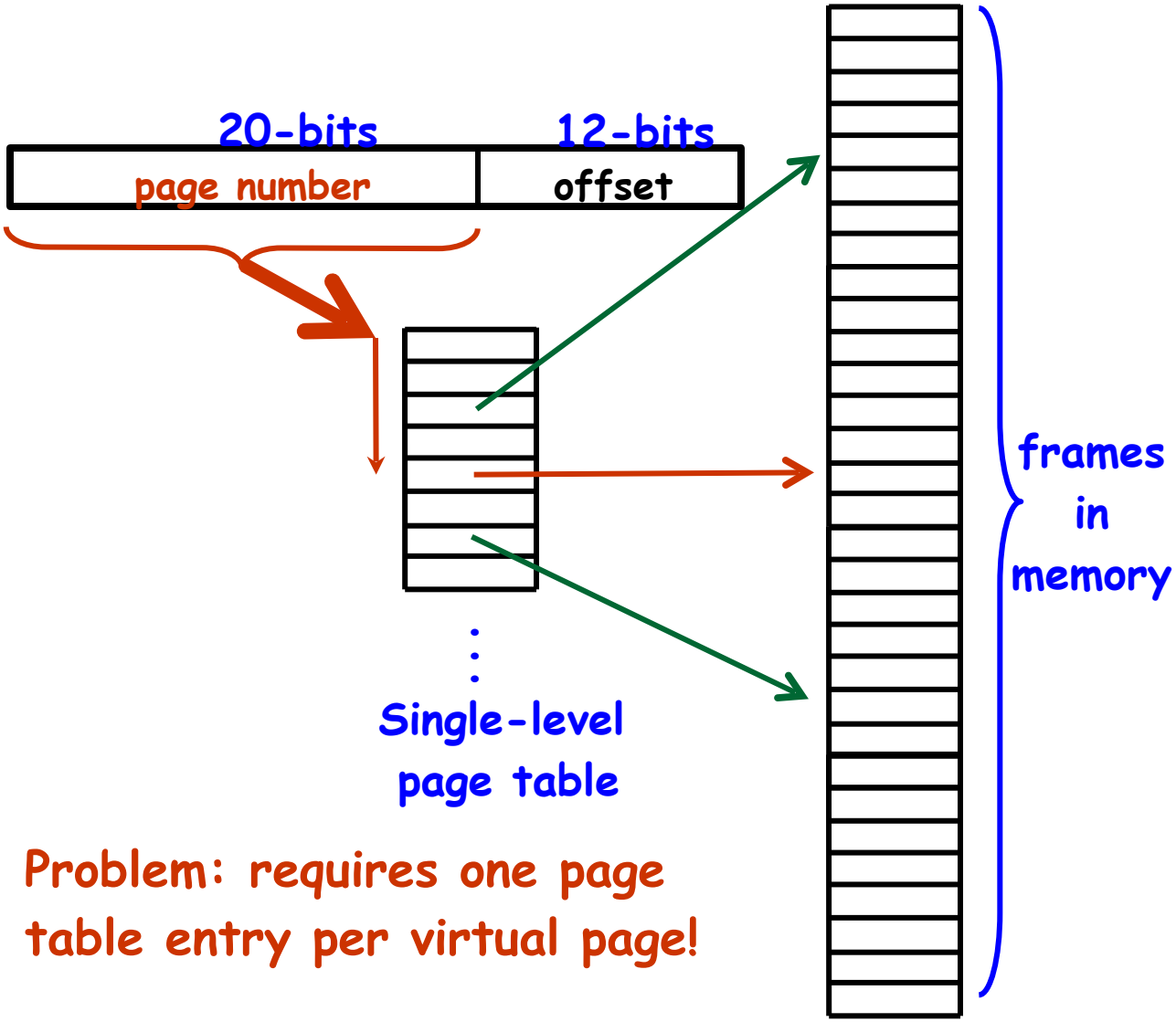
A Virtual Address (32 bit):



Single-level page tables

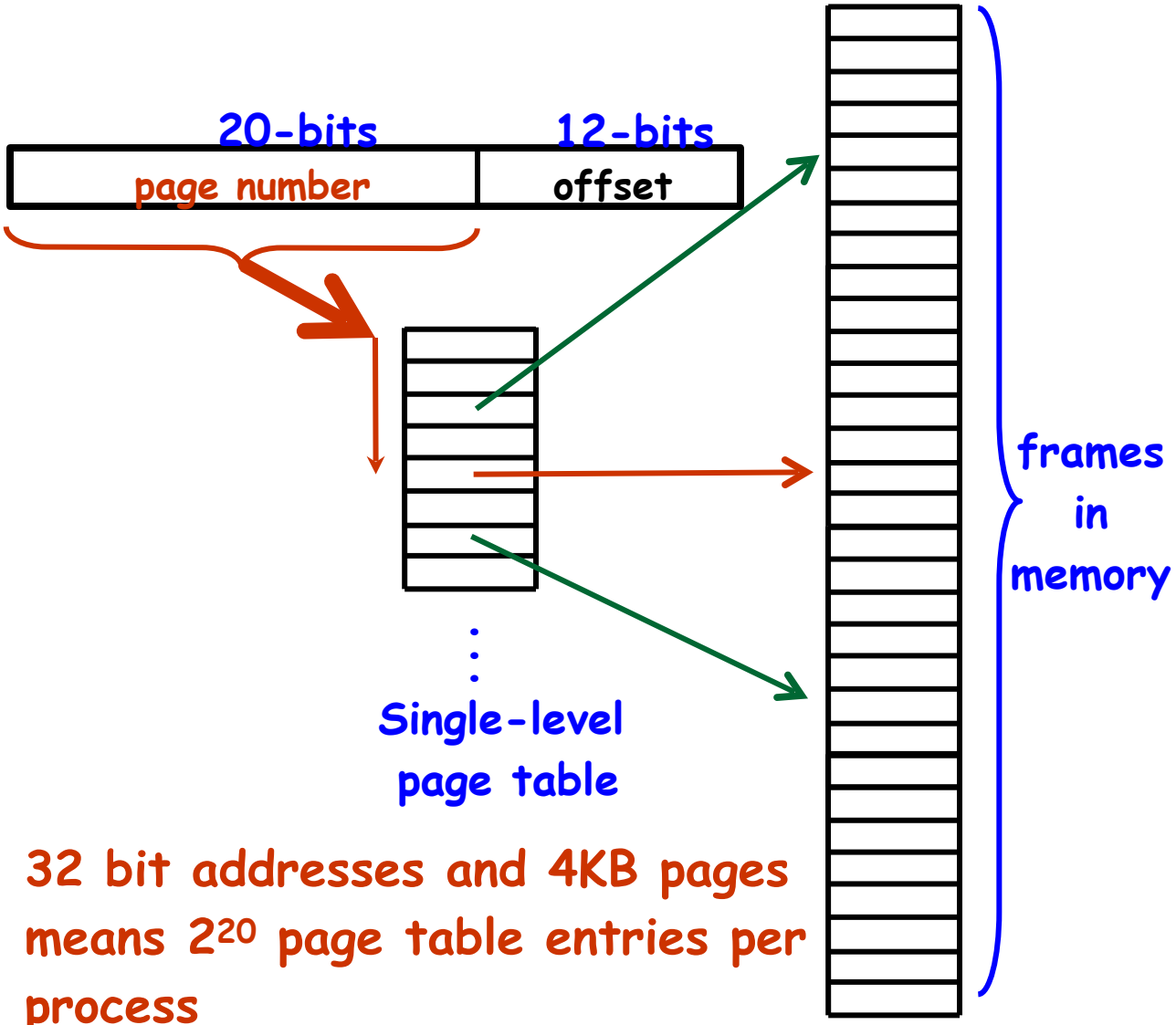


Single-level page tables



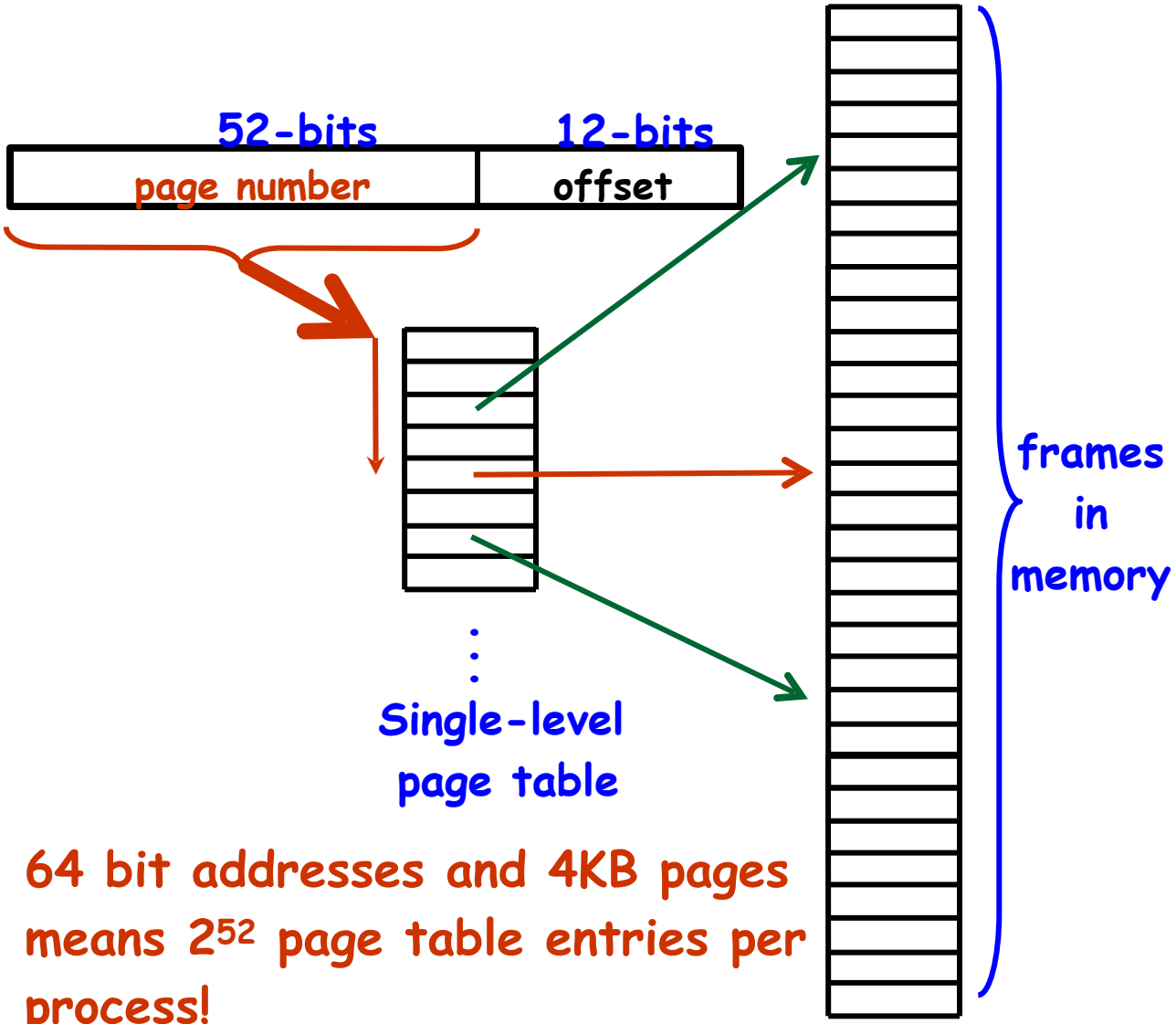
Problem: requires one page table entry per virtual page!

Single-level page tables



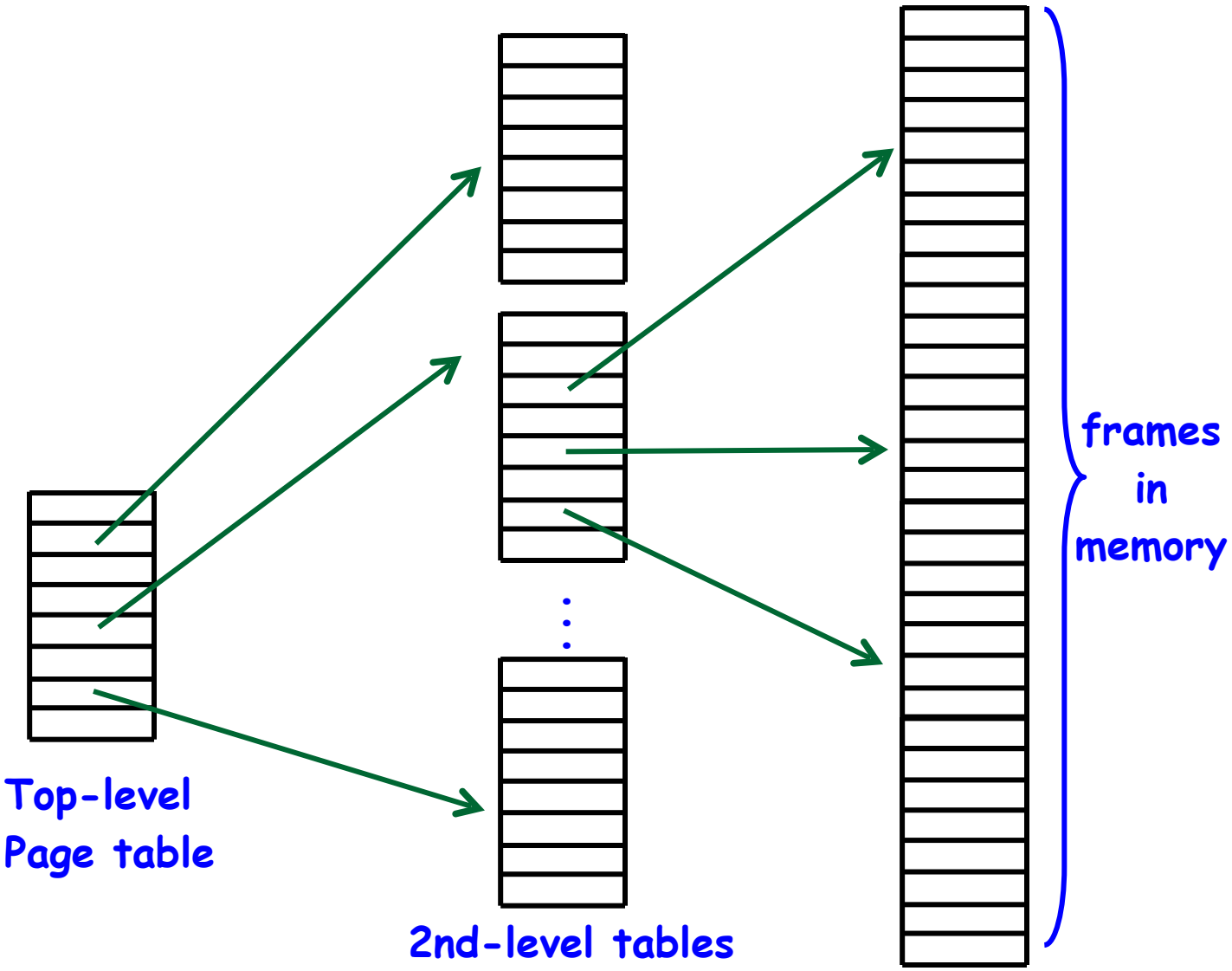
32 bit addresses and 4KB pages means 2^{20} page table entries per process

Single-level page tables



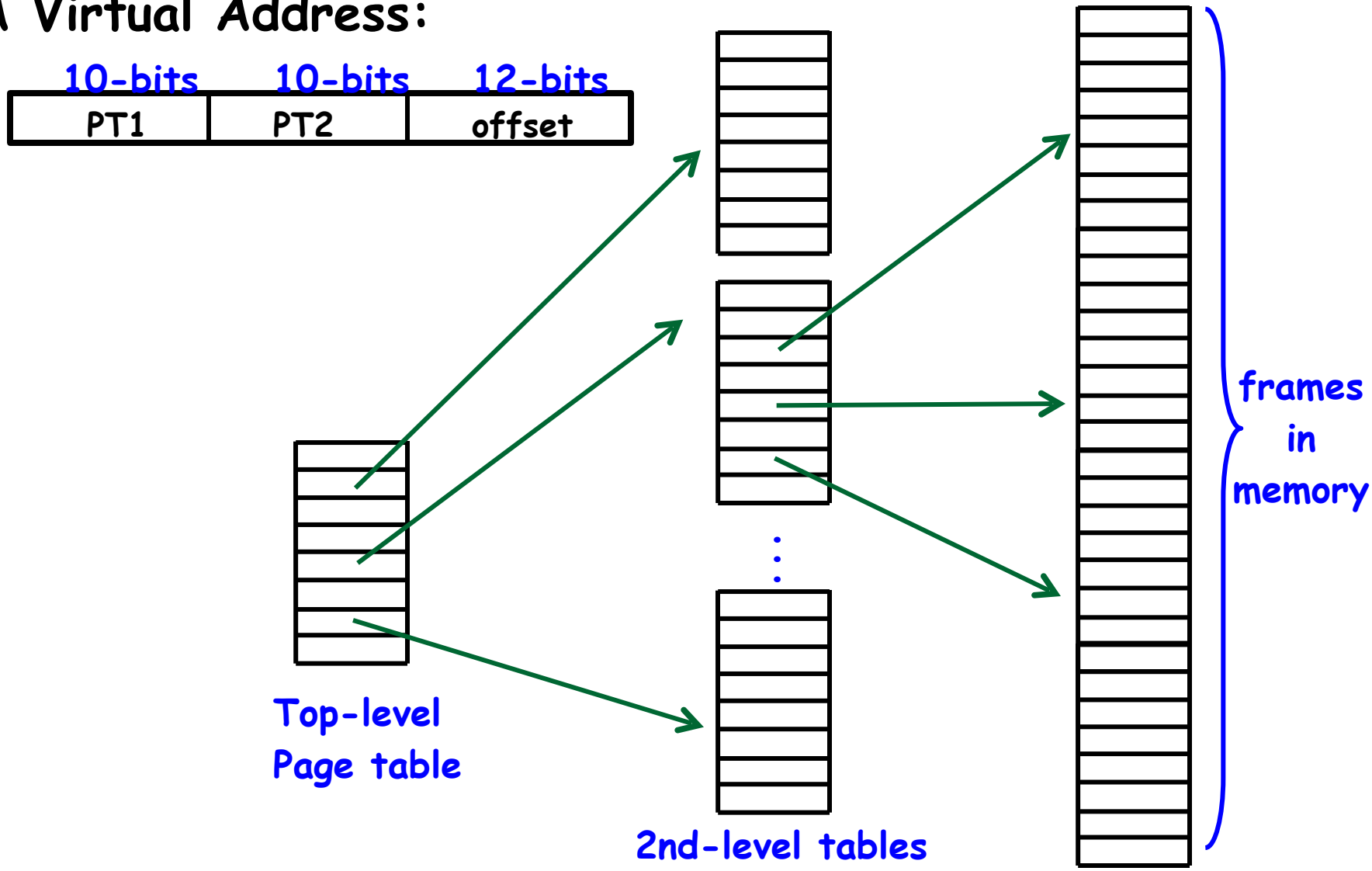
64 bit addresses and 4KB pages means 2^{52} page table entries per process!

Multi-level page tables



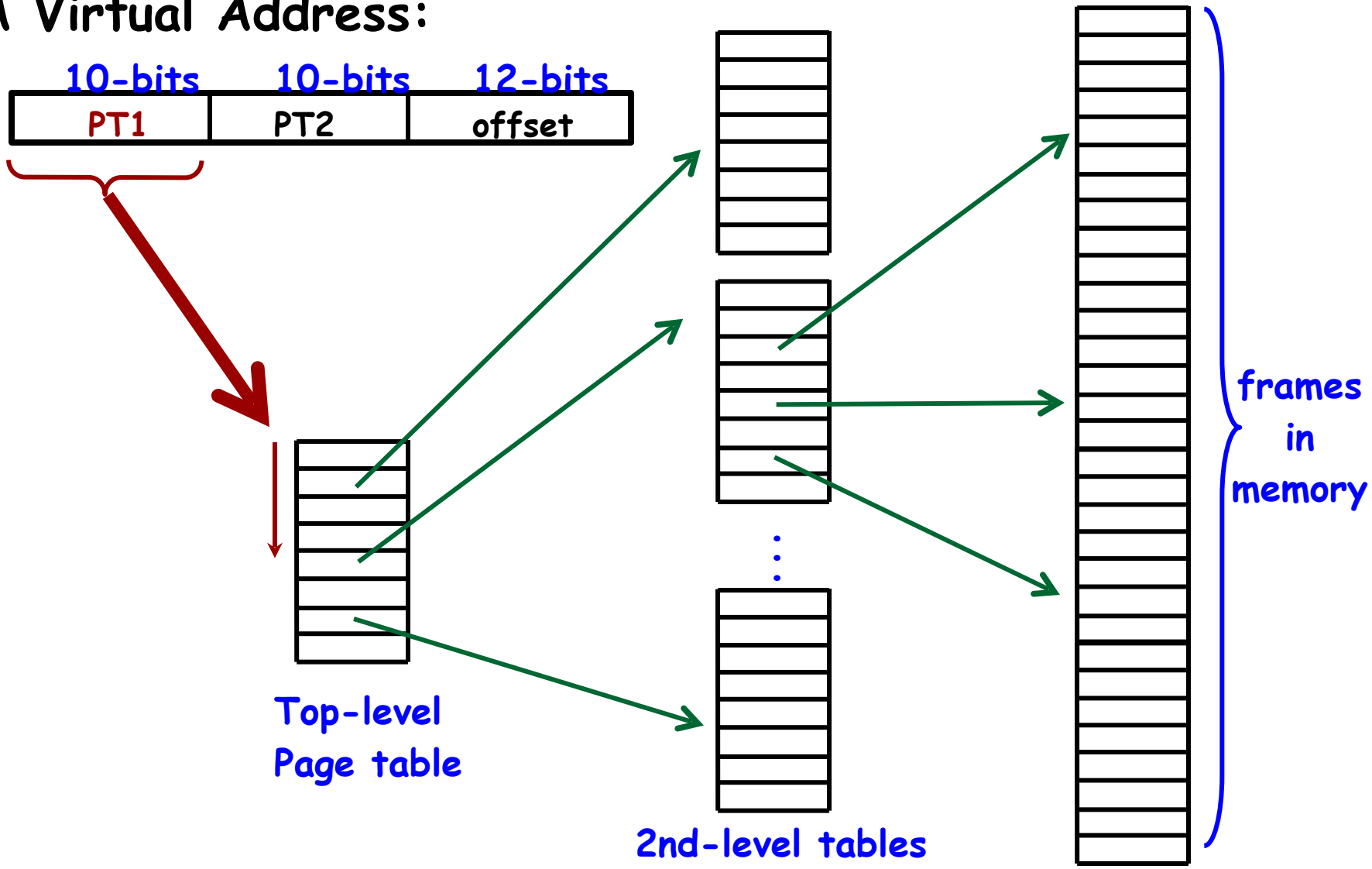
Multi-level page tables

- A Virtual Address:



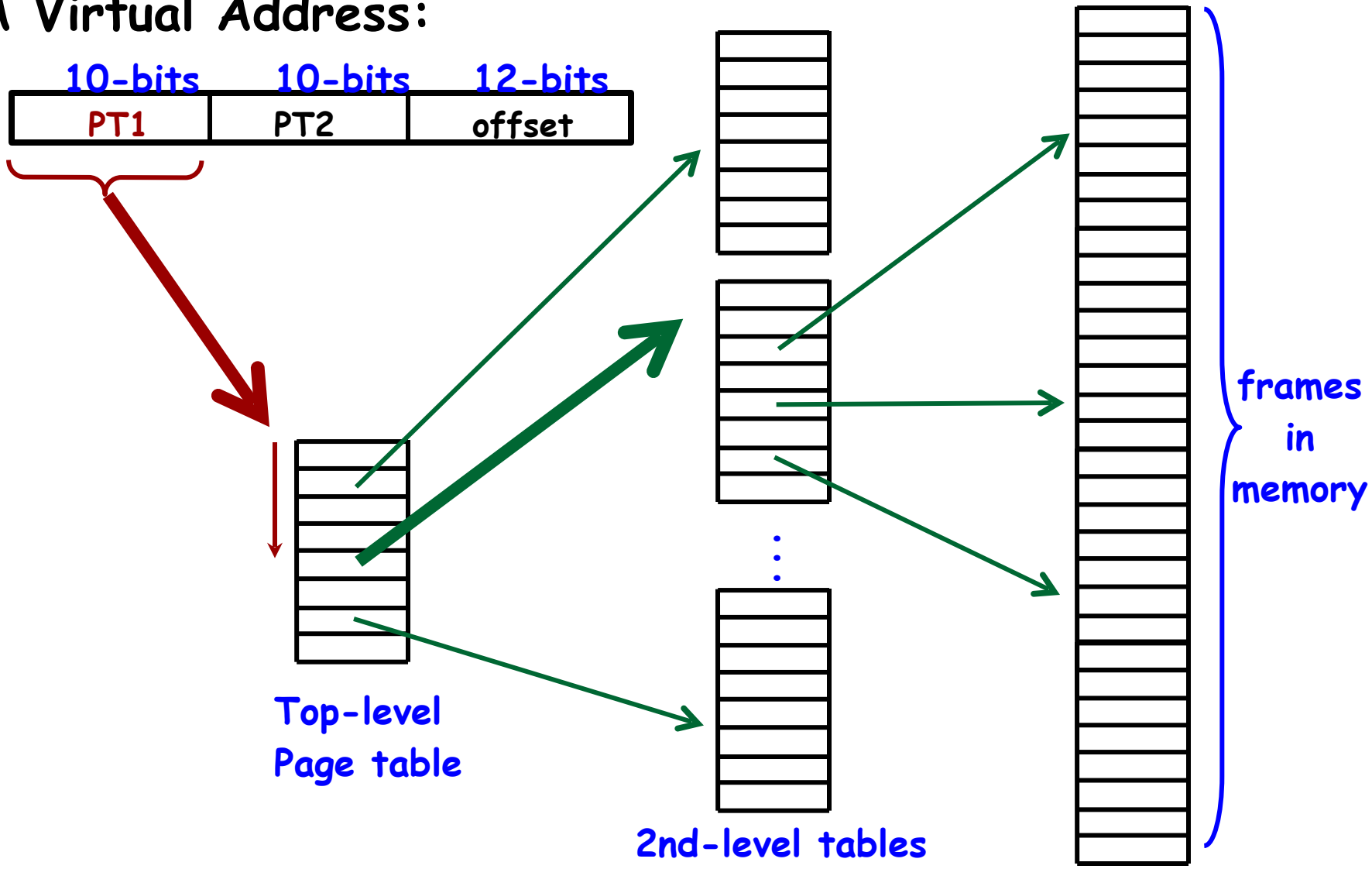
Multi-level page tables

- A Virtual Address:



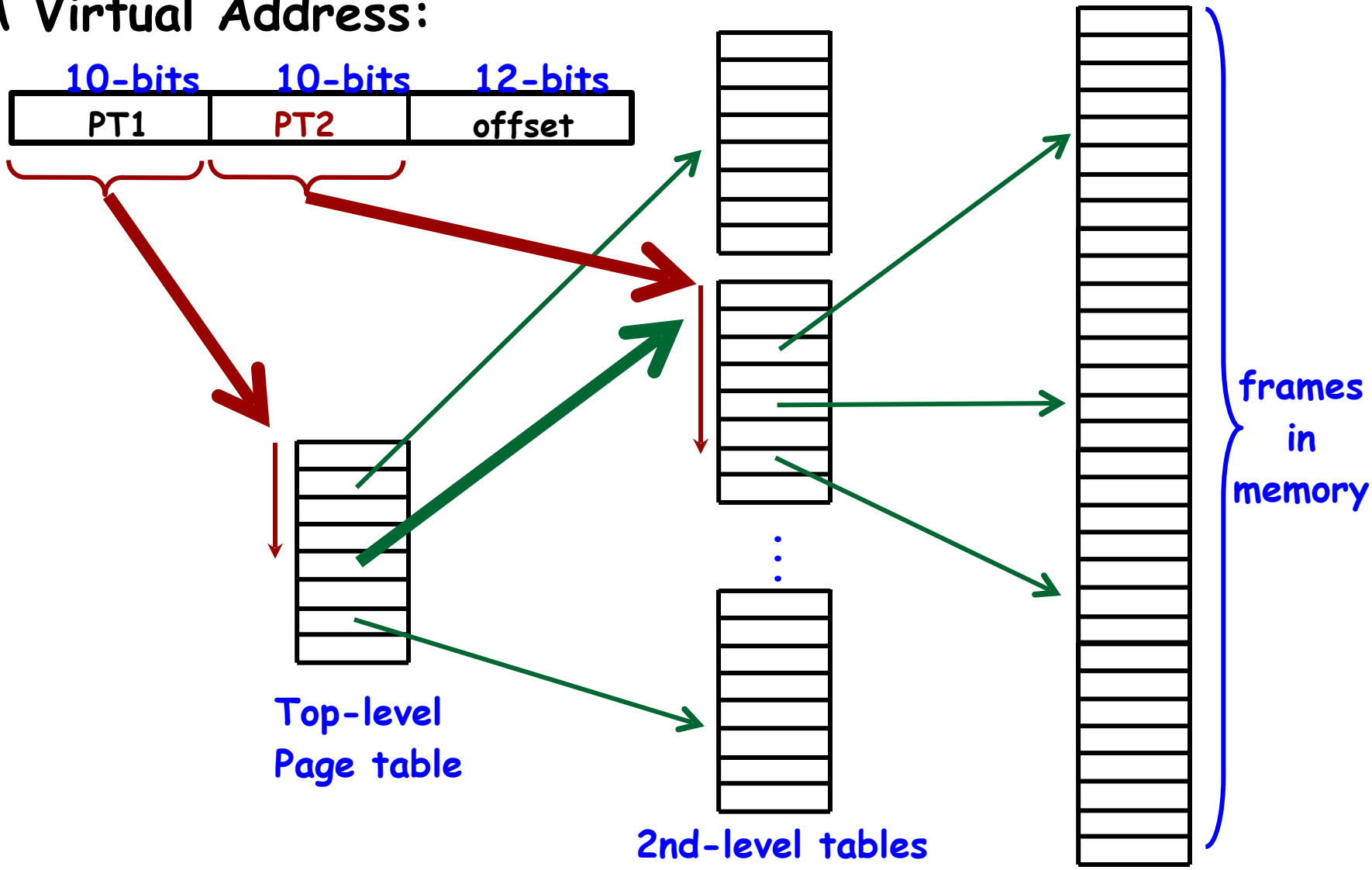
Multi-level page tables

- A Virtual Address:



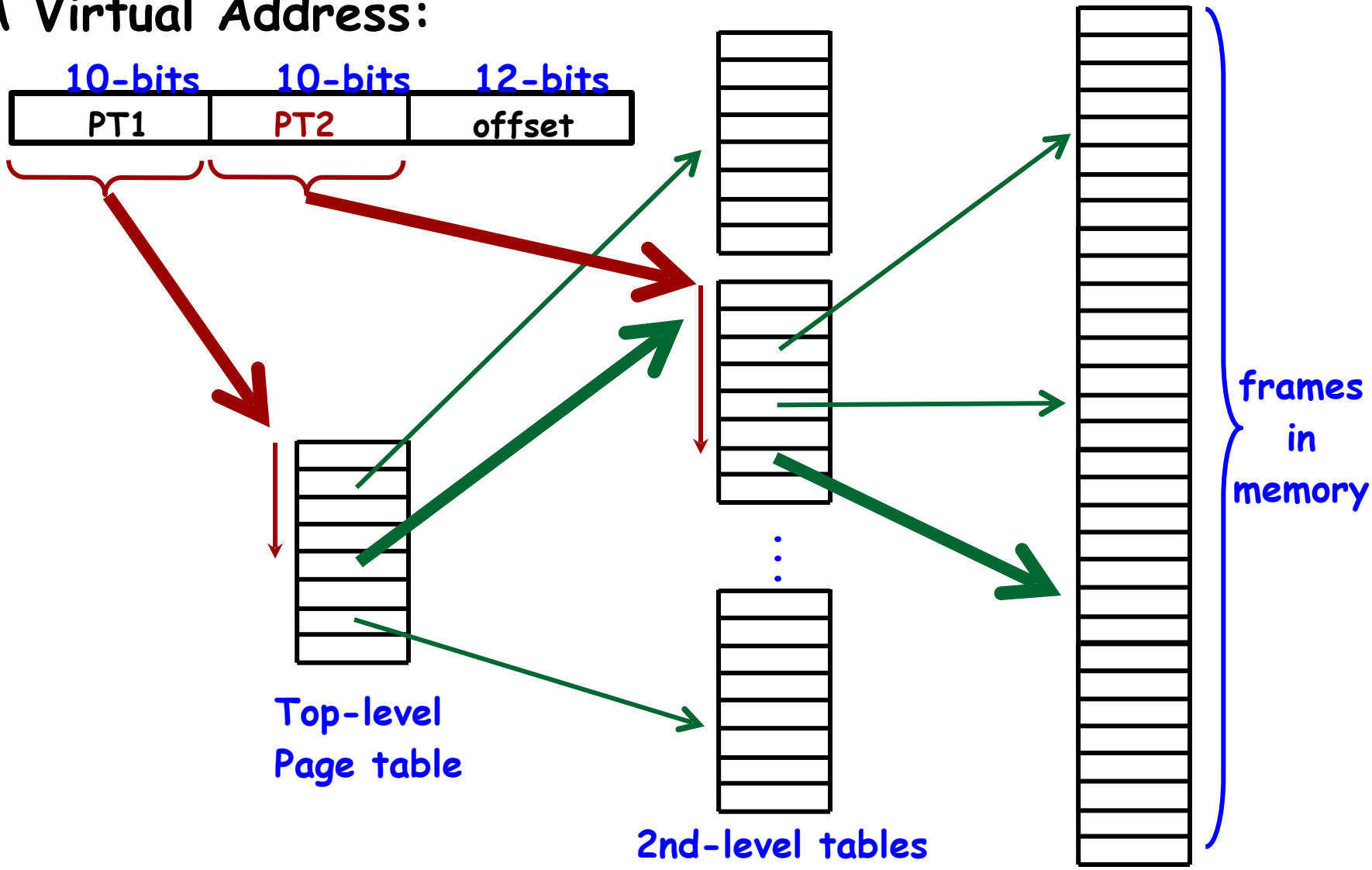
Multi-level page tables

- A Virtual Address:



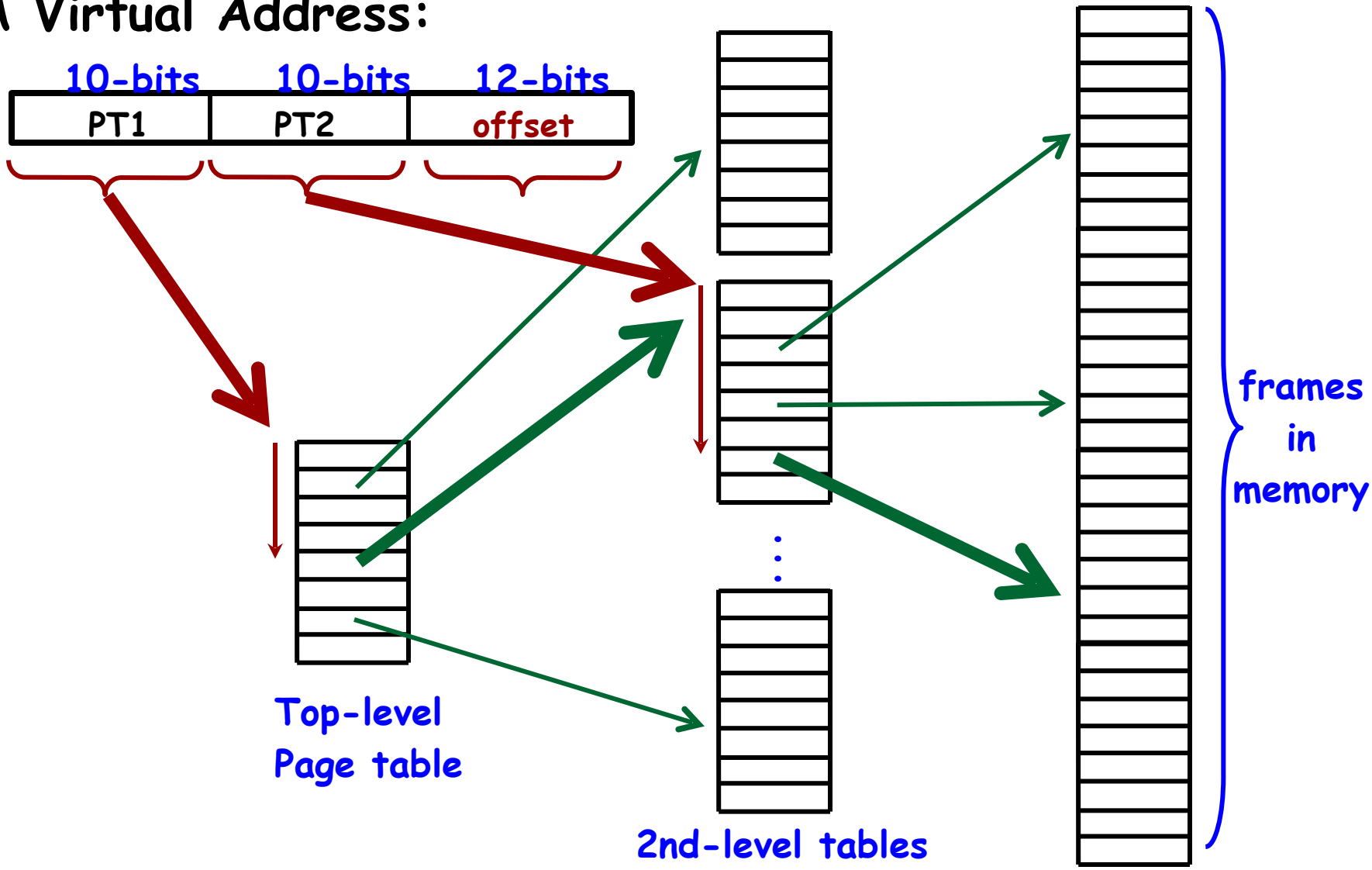
Multi-level page tables

- A Virtual Address:



Multi-level page tables

- A Virtual Address:



Multi-level page tables

- Ok, but how exactly does this save space?

Multi-level page tables

- **Ok, but how exactly does this save space?**
- **Not all pages within a virtual address space are allocated**
  - Not only do they not have a page frame, but that range of virtual addresses is not being used
  - So no need to maintain complete information about it
  - Some intermediate page tables are empty and not needed
- **We could also page the page table**
  - This saves space but slows access … a lot!

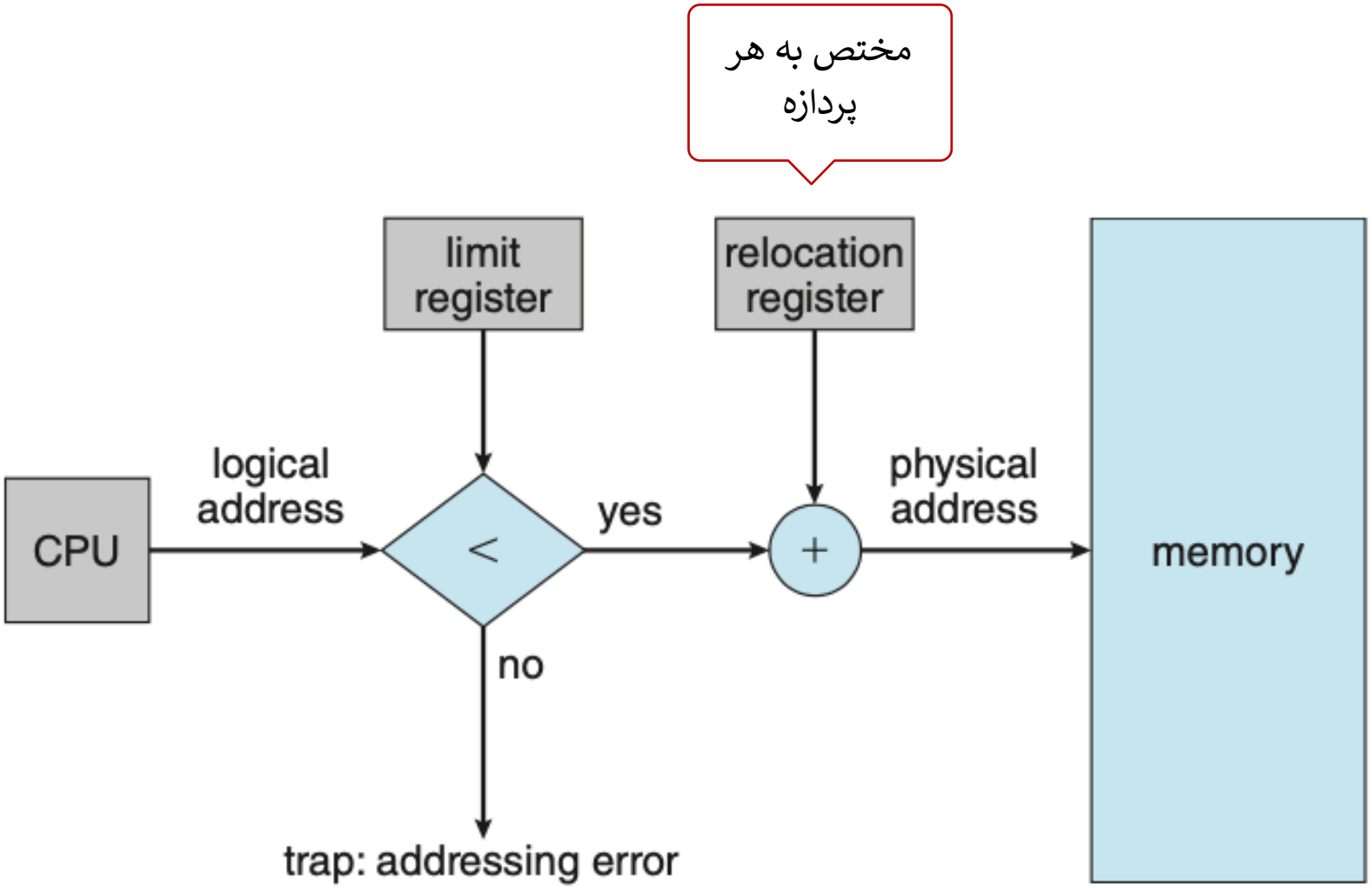
بسم الله الرحمن الرحیم

۹۹

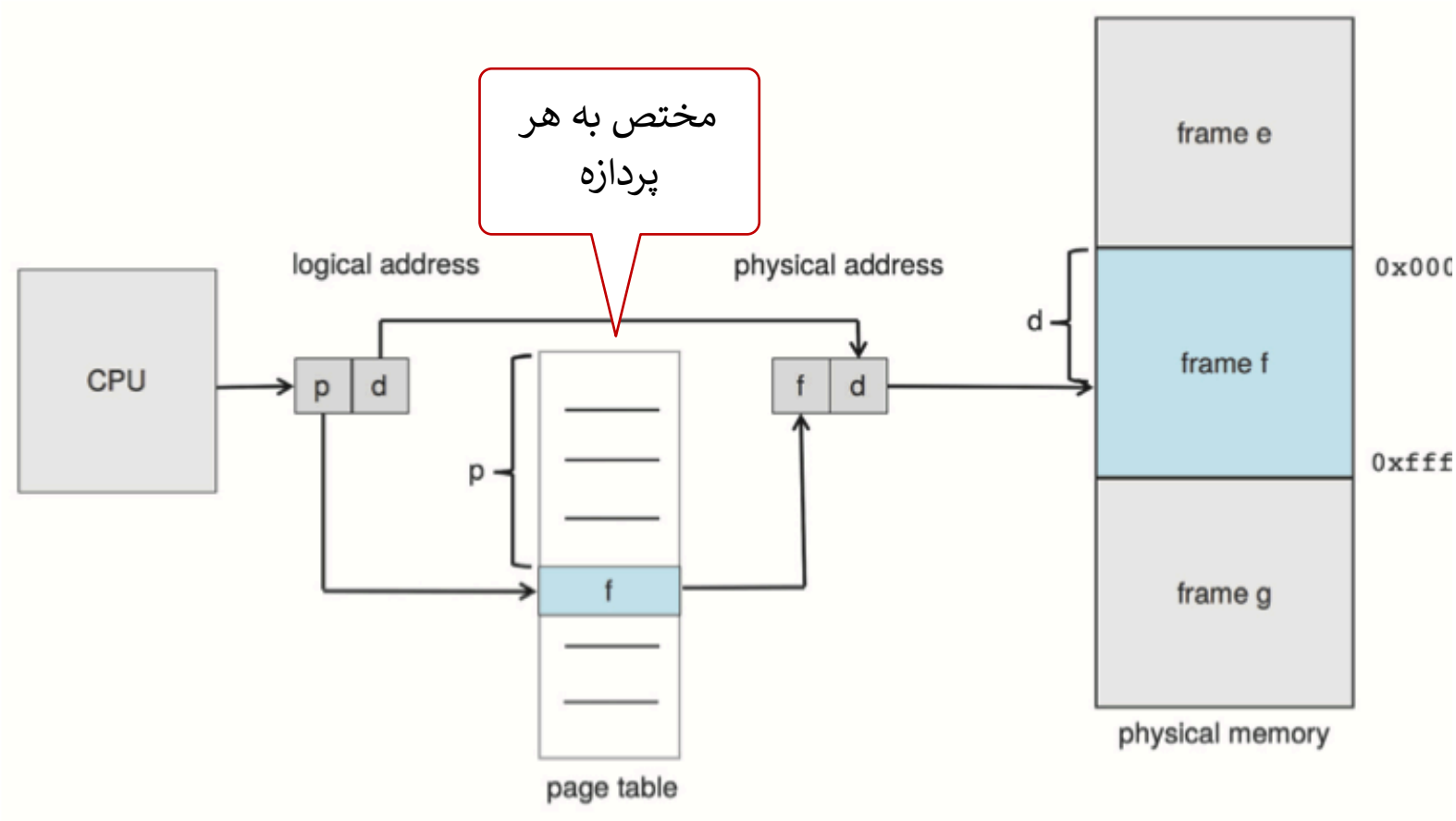
«سیستم عامل»

جلسه ۱۶: مدیریت حافظه (۴)

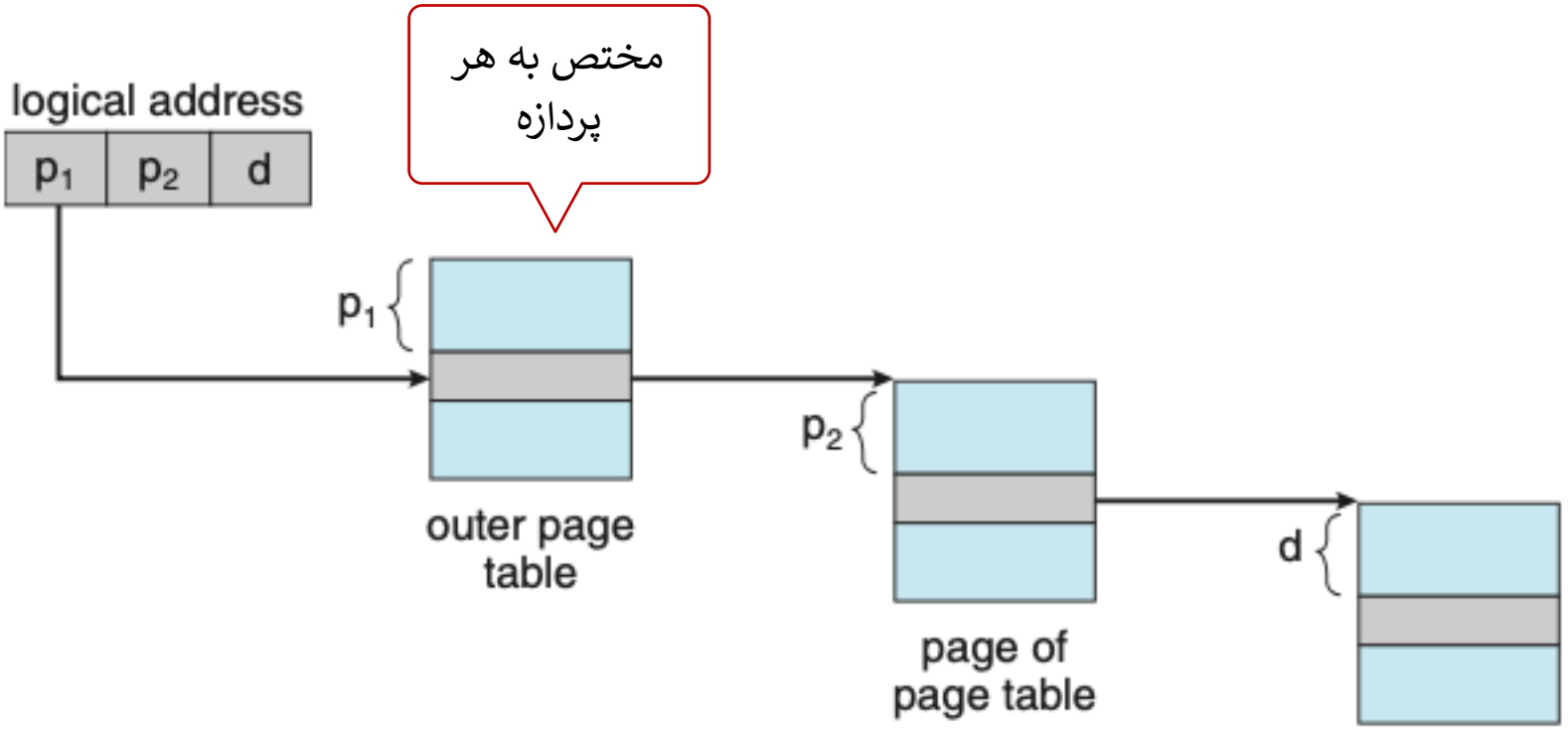
مرور ـ حافظه متوالی (Contiguous)



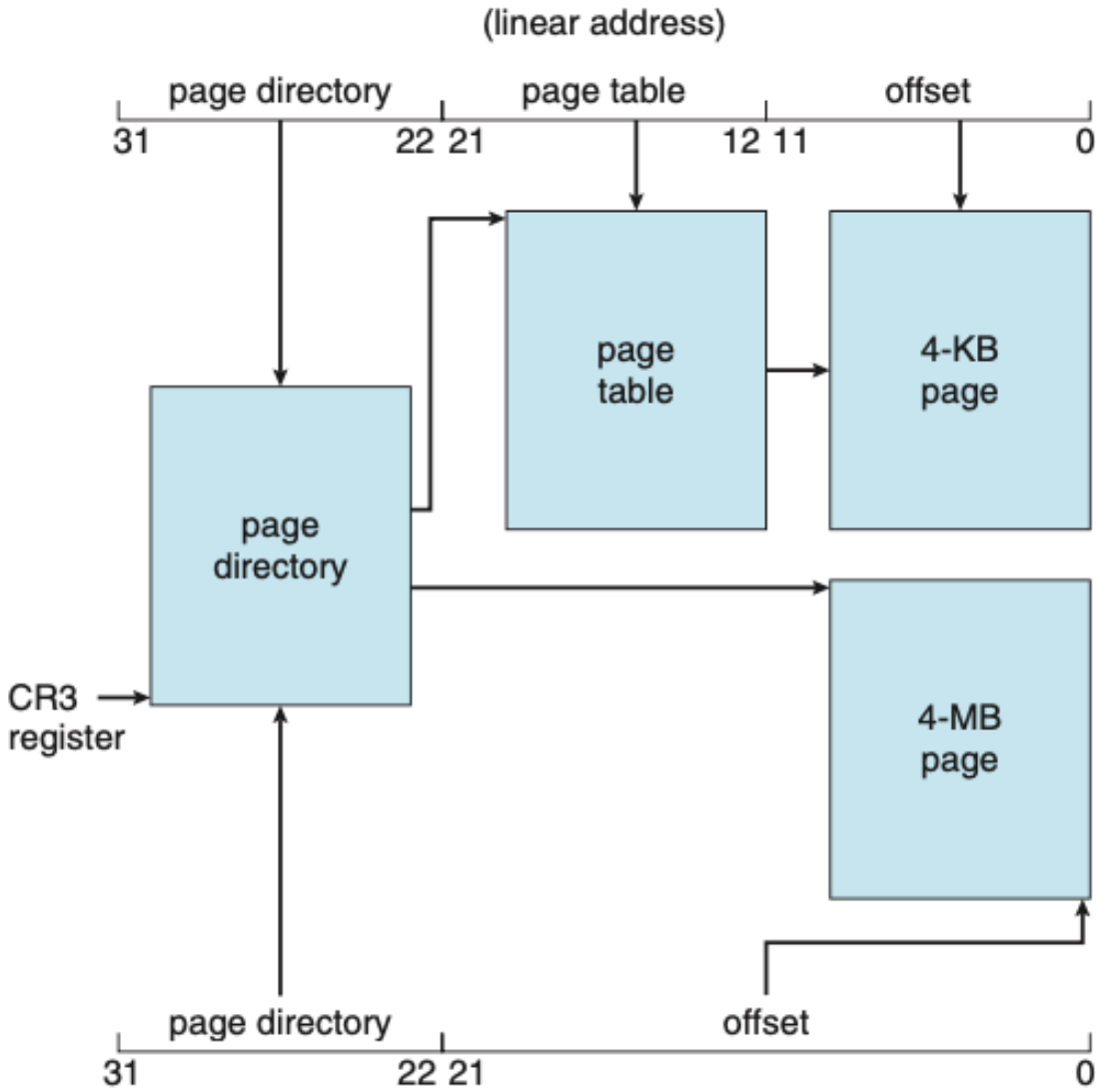
مرور ـ حافظه صفحه‌بندی‌شده



مرور ـ حافظه صفحه‌بندی‌شده ـ دو سطحی

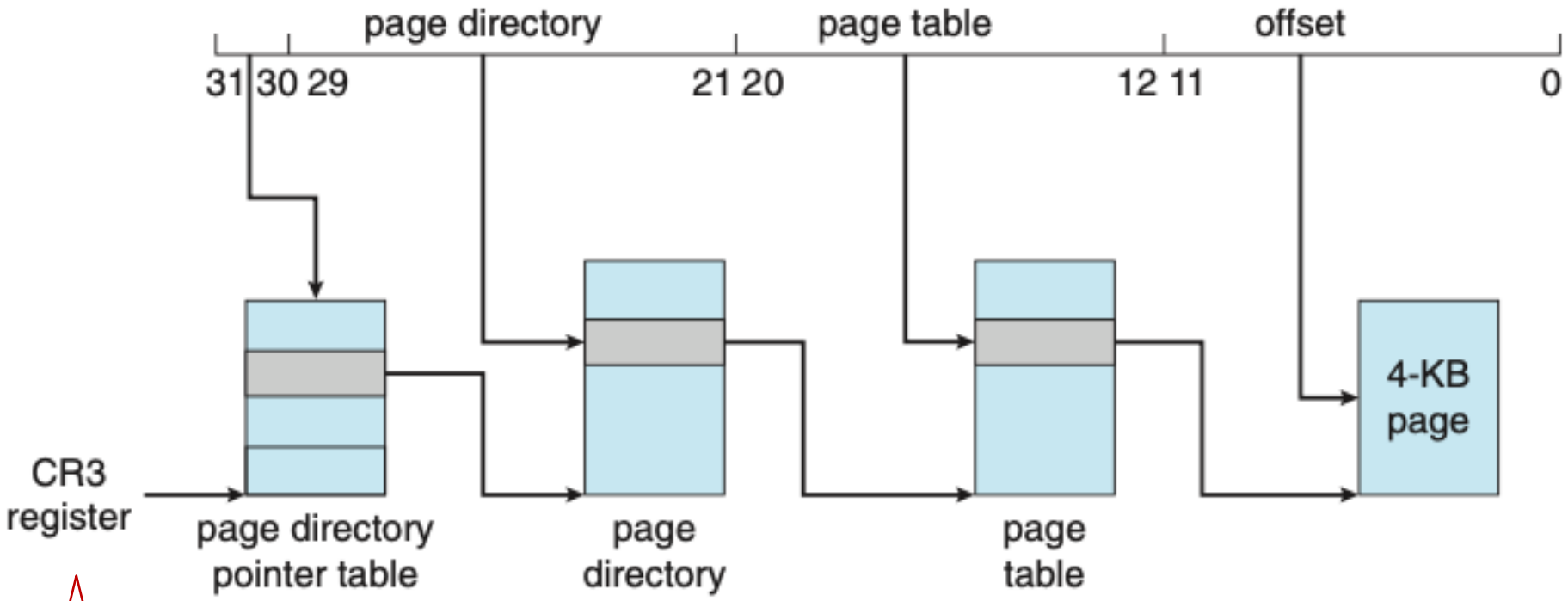


مرور ـ حافظه صفحه‌بندی‌شده ـ دو سطحی (مثال IA-32)



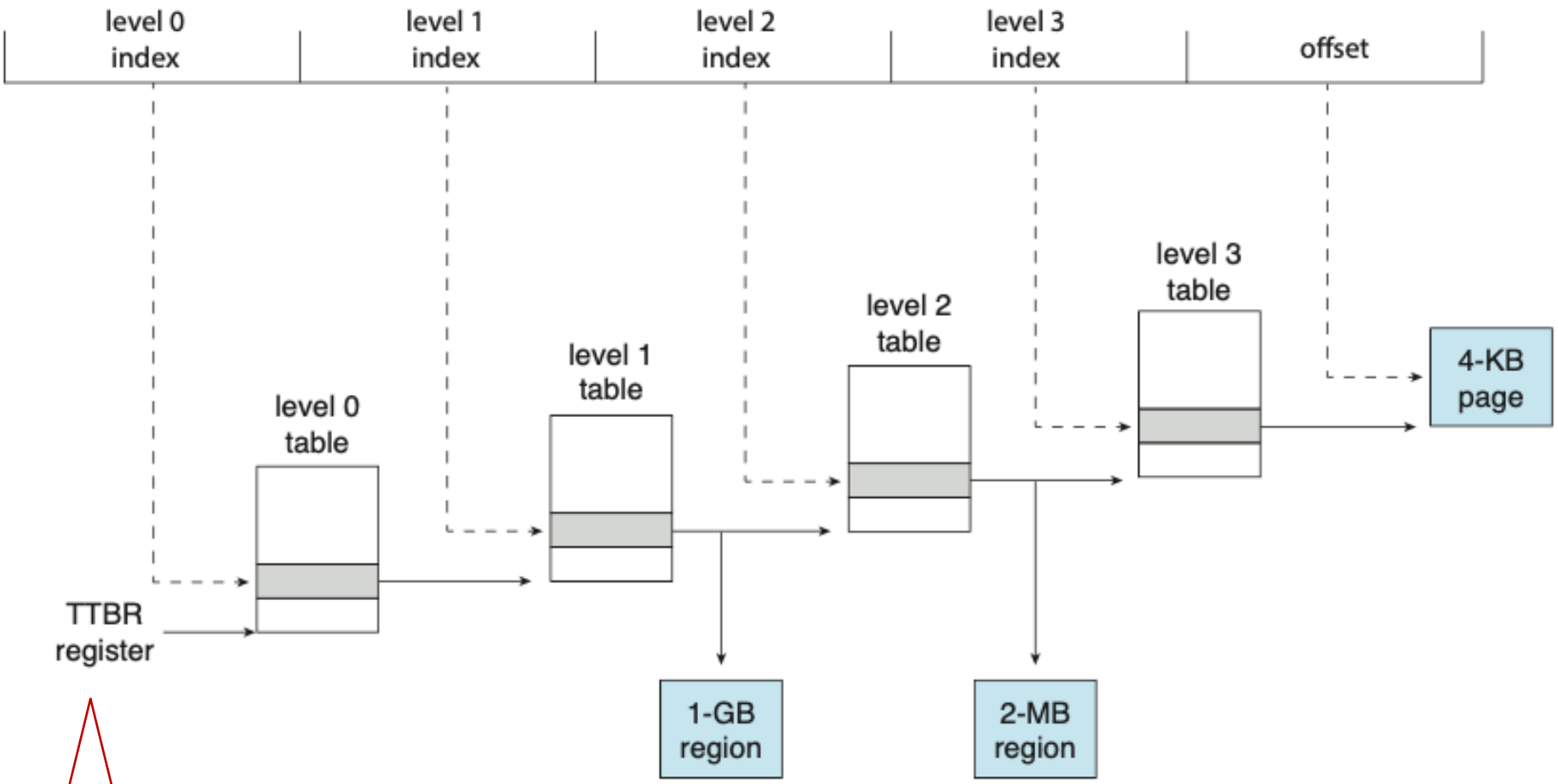
مختص به هر پردازه

مرور ـ حافظه صفحه‌بندی‌شده ـ دو سطحی (مثال PAE)



مختص به هر
پردازه

مرور ـ حافظه صفحه‌بندی‌شده ـ دو سطحی (مثال PAE)



مختص به هر پردازه

Figure 9.27 ARM four-level hierarchical paging.

قطعه‌بندی ...

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?

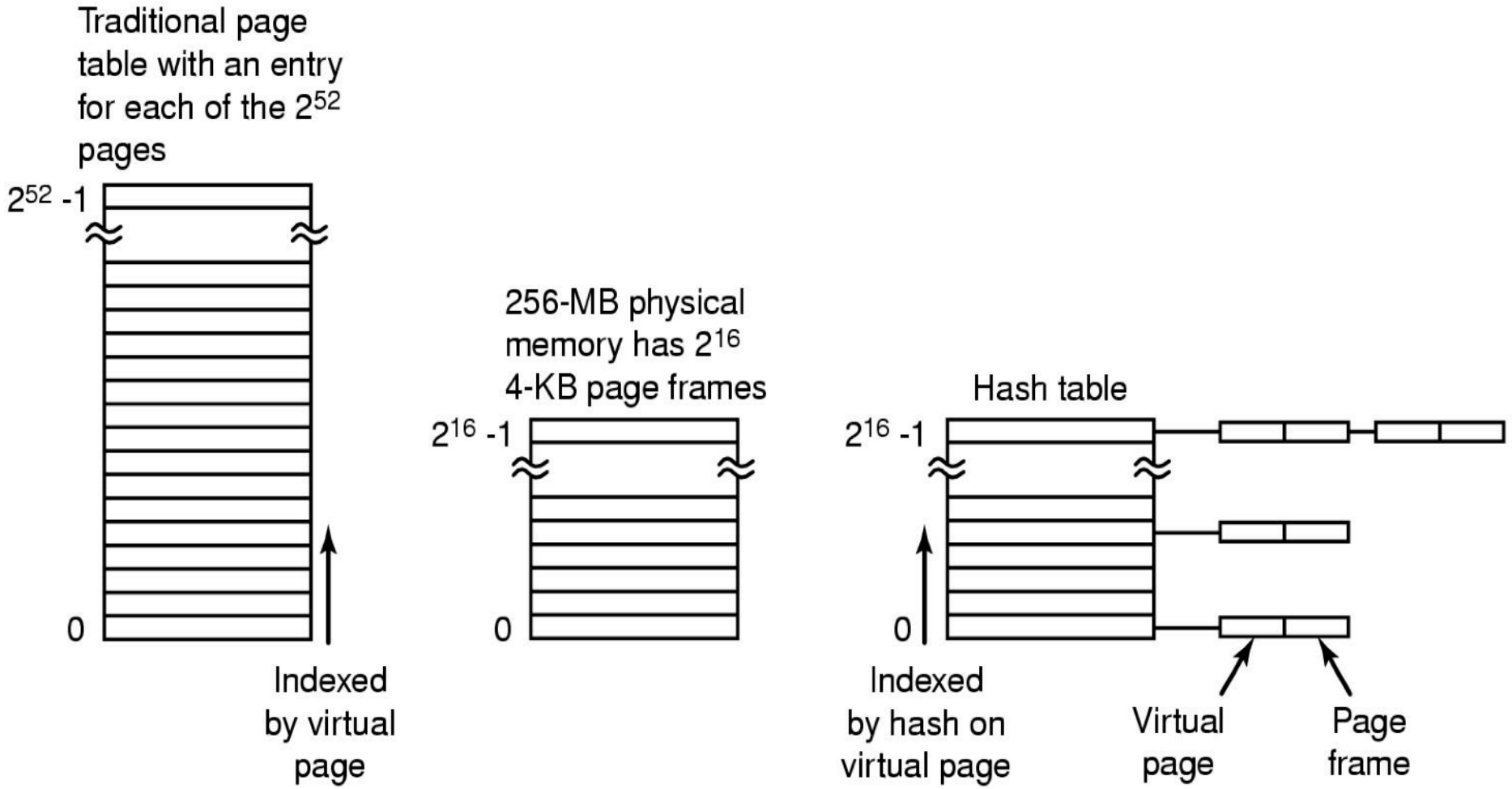
Inverted page tables

- **If we have a page number (from a faulting address) and want to find it 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



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 though 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

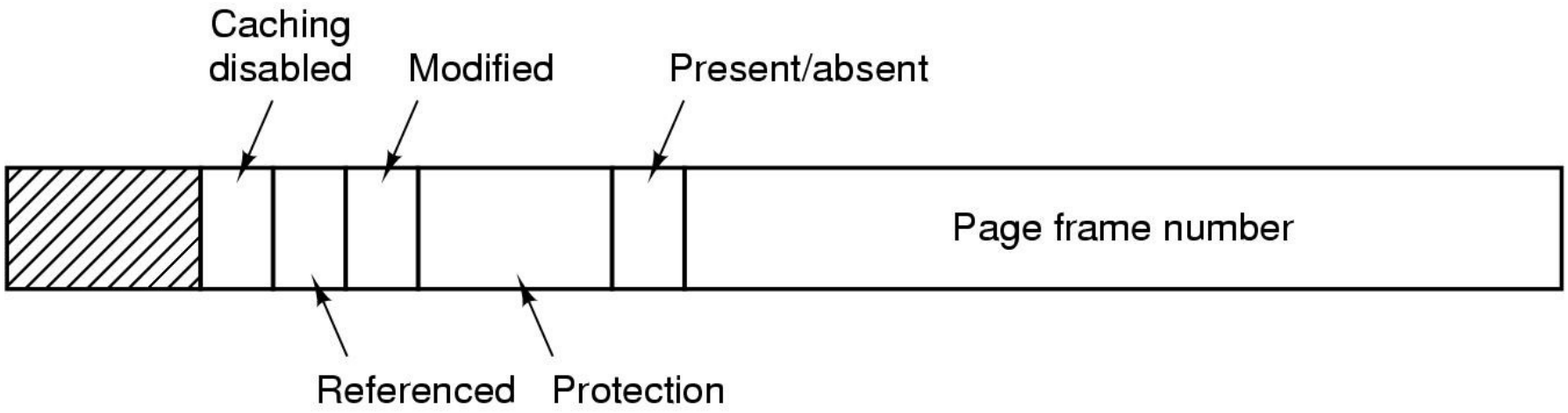
Protection lookaside buffer

- **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

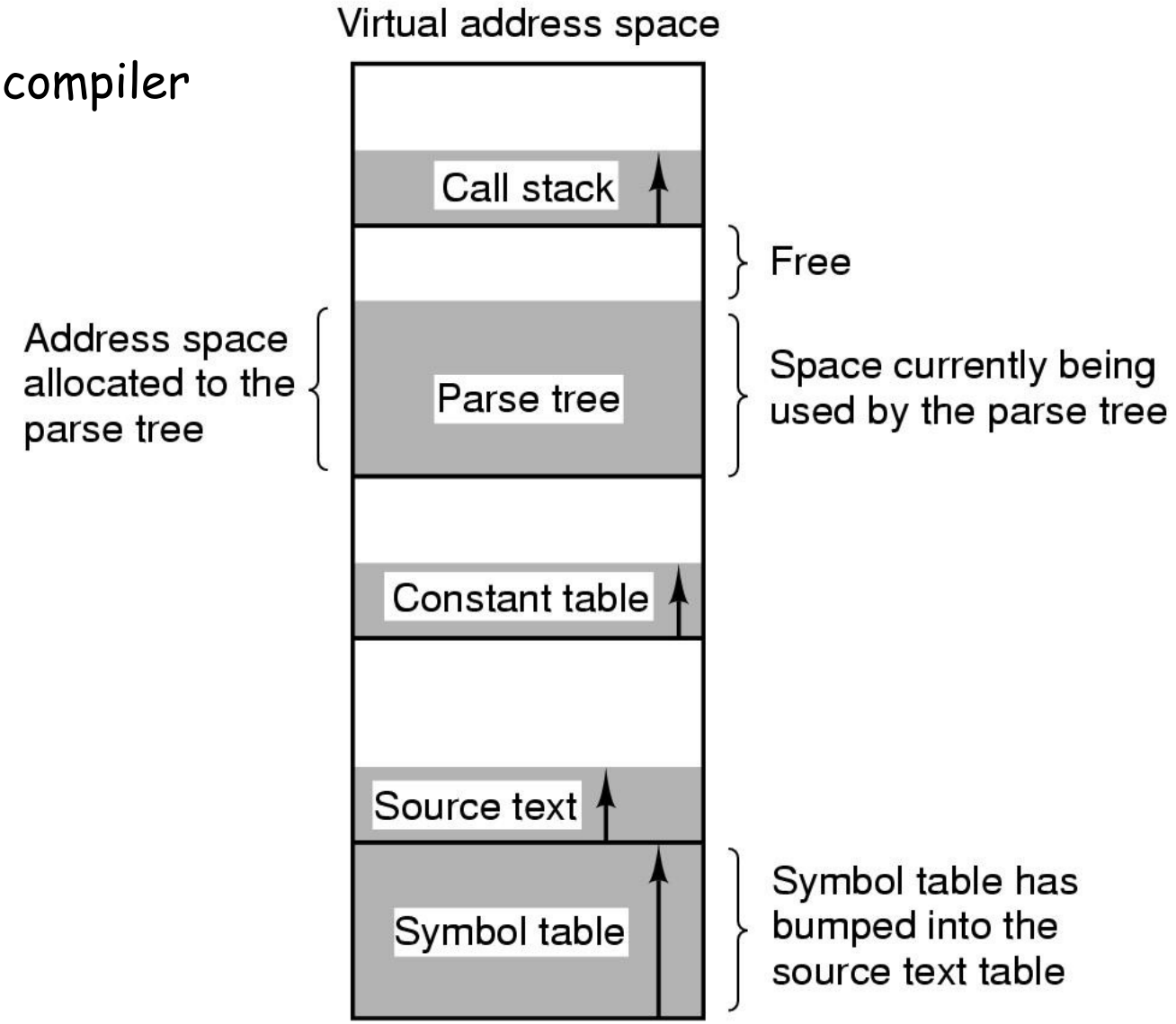
- **Traditional Virtual Address Space**
  - "flat" address space (1 dimensional)

- **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

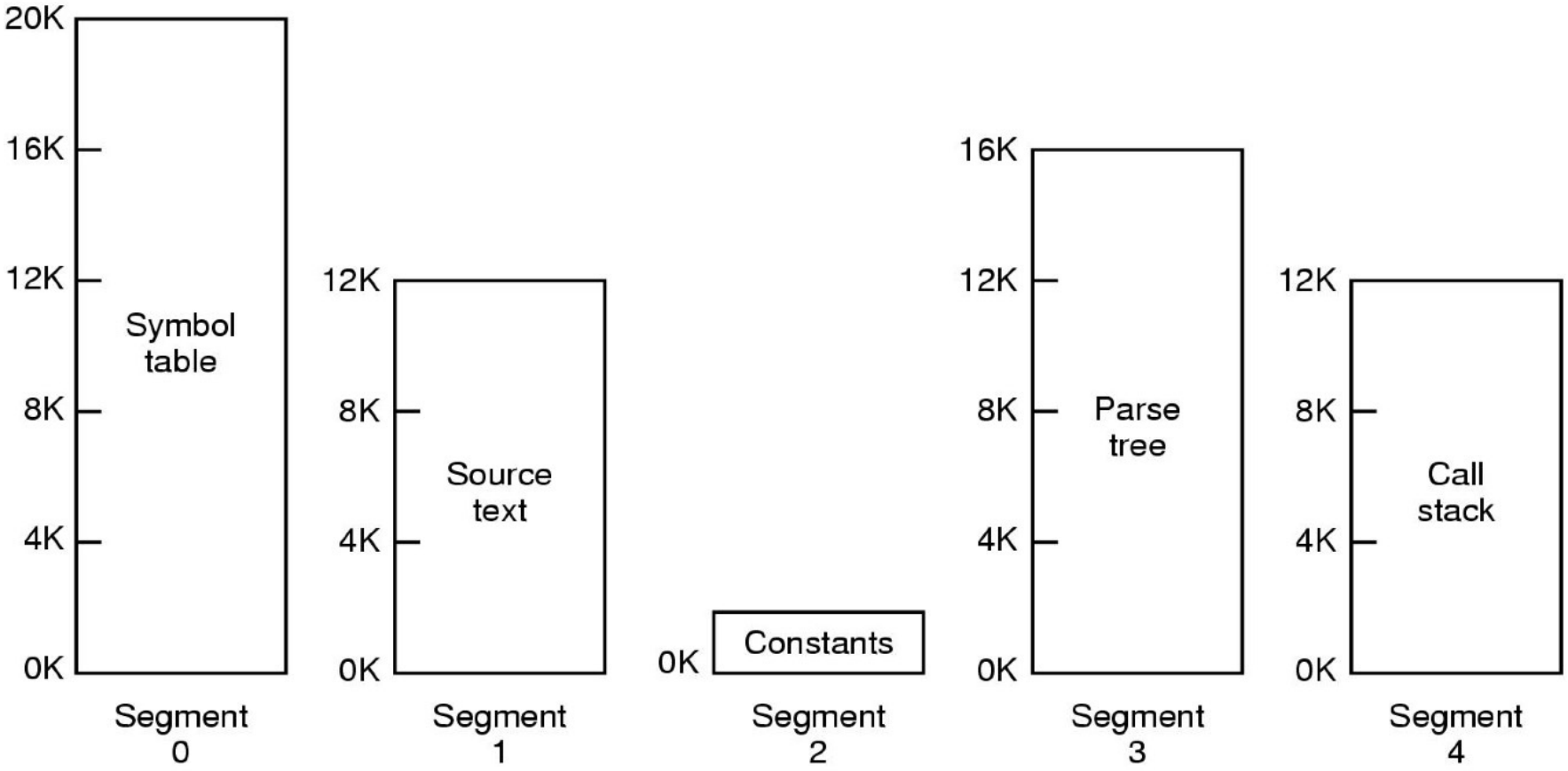
Segmentation in a single address space

Example: A compiler

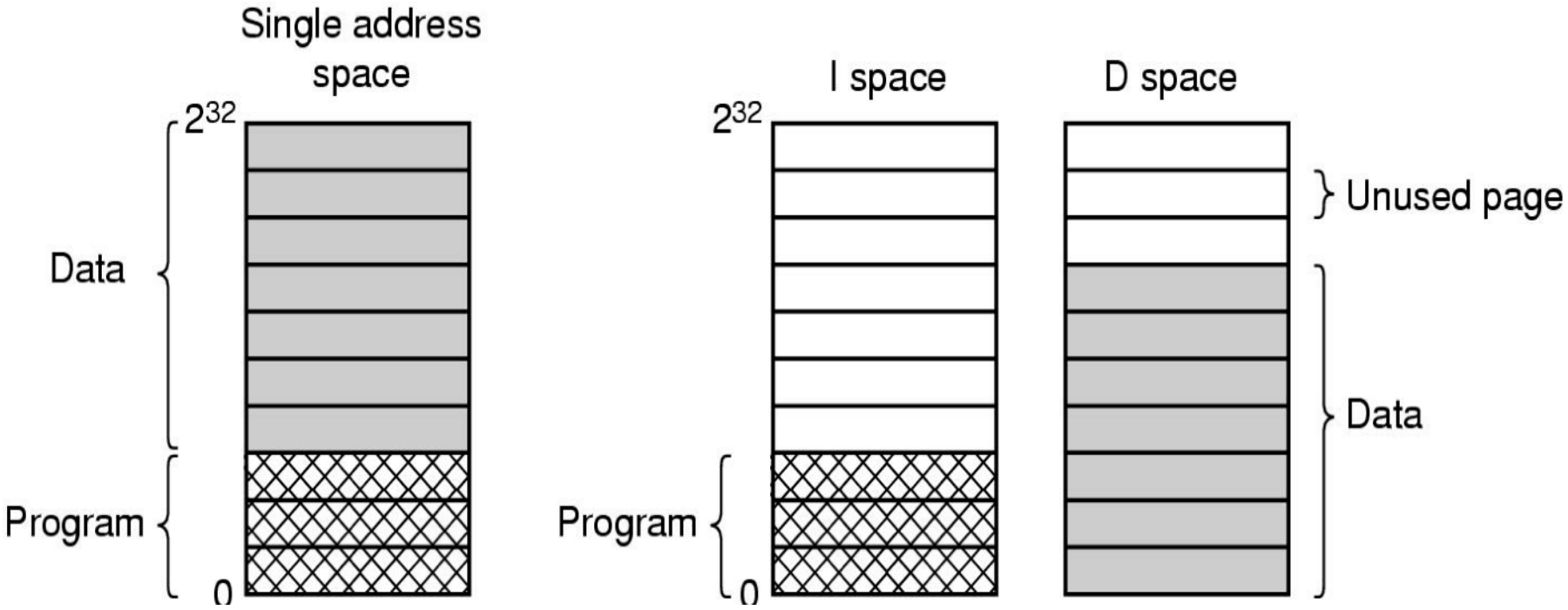


Segmented memory

- Each space grows, shrinks independently!



Separate instruction and data spaces



* One address space

* Separate I and D spaces

Comparison of paging vs. segmentation

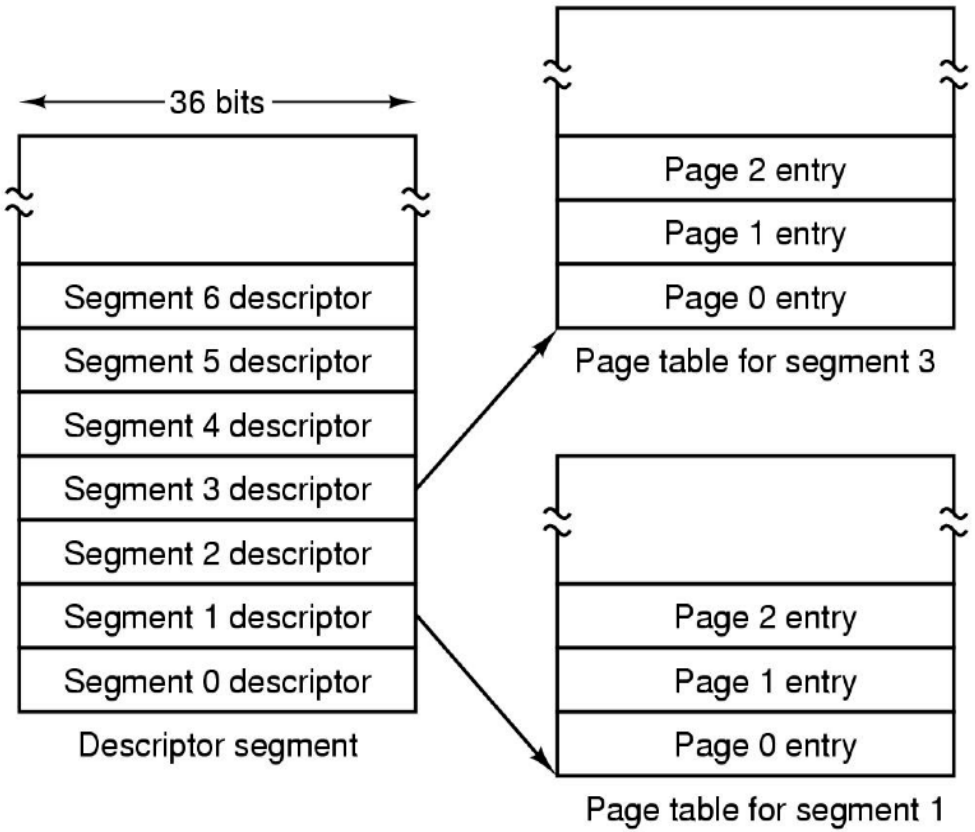
Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can 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

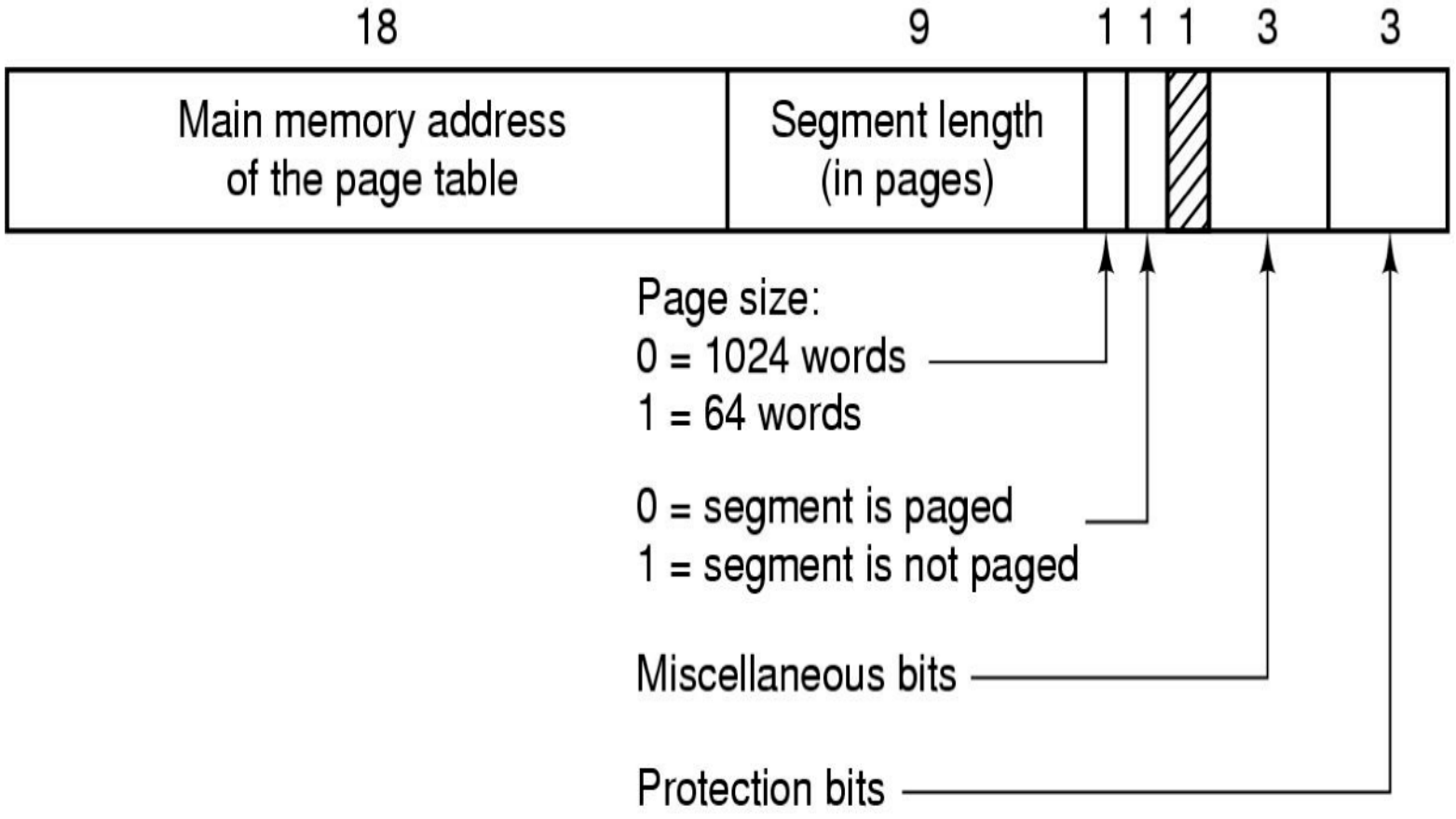
Segmentation with paging (MULTICS)

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

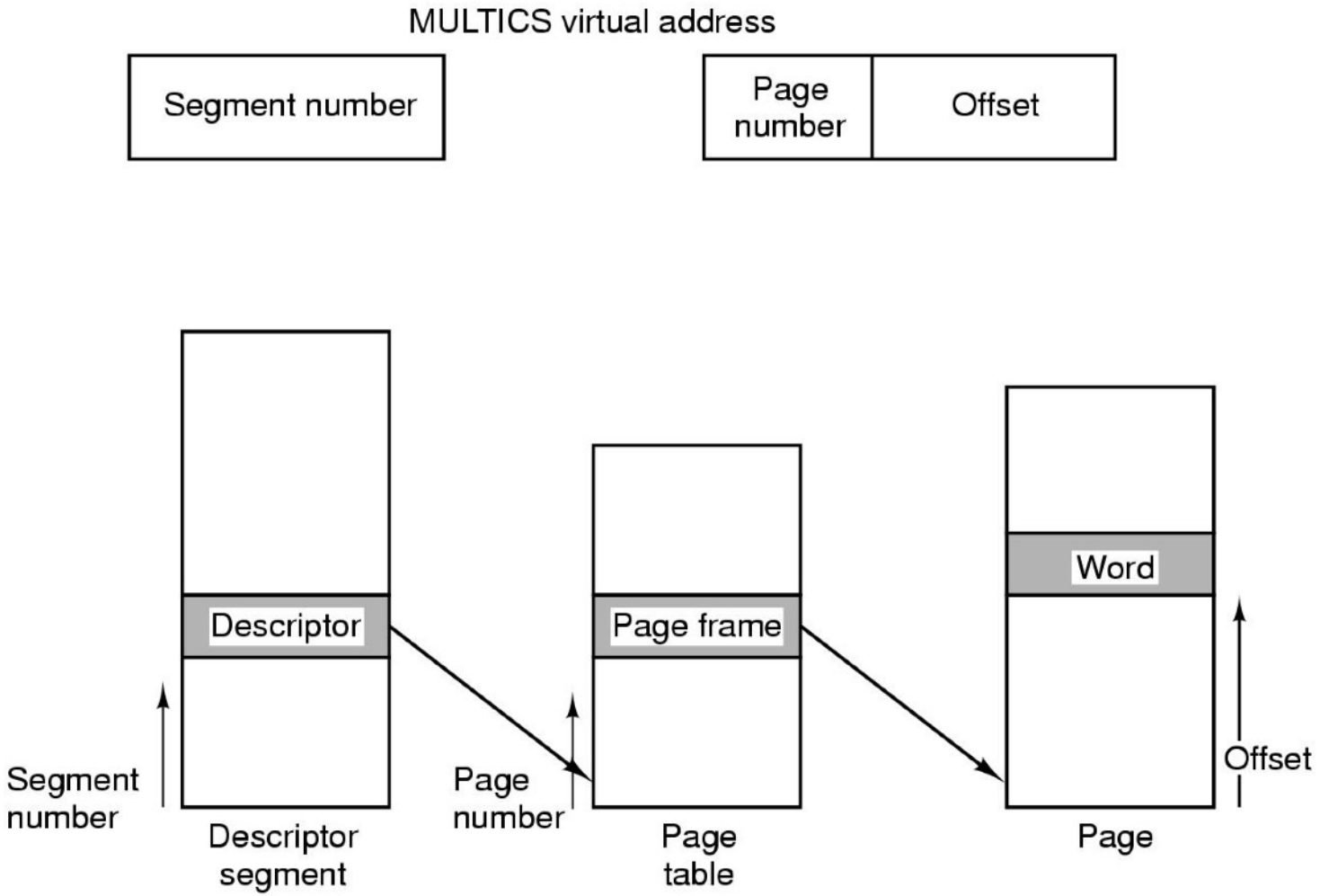


Segmentation with paging (MULTICS)

- Each entry in segment table...

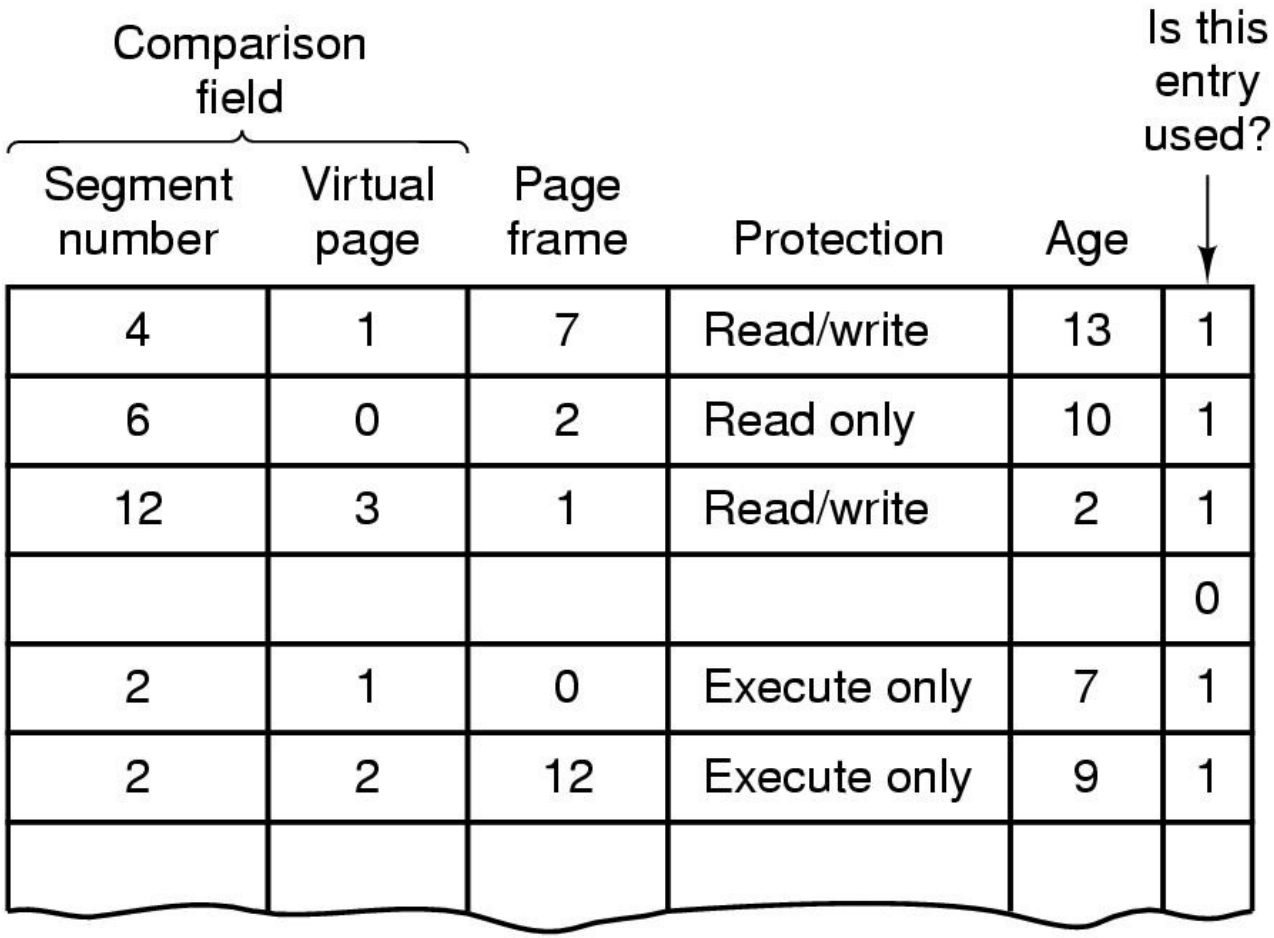


Segmentation with paging: MULTICS



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

Segmentation with Paging: TLB operation

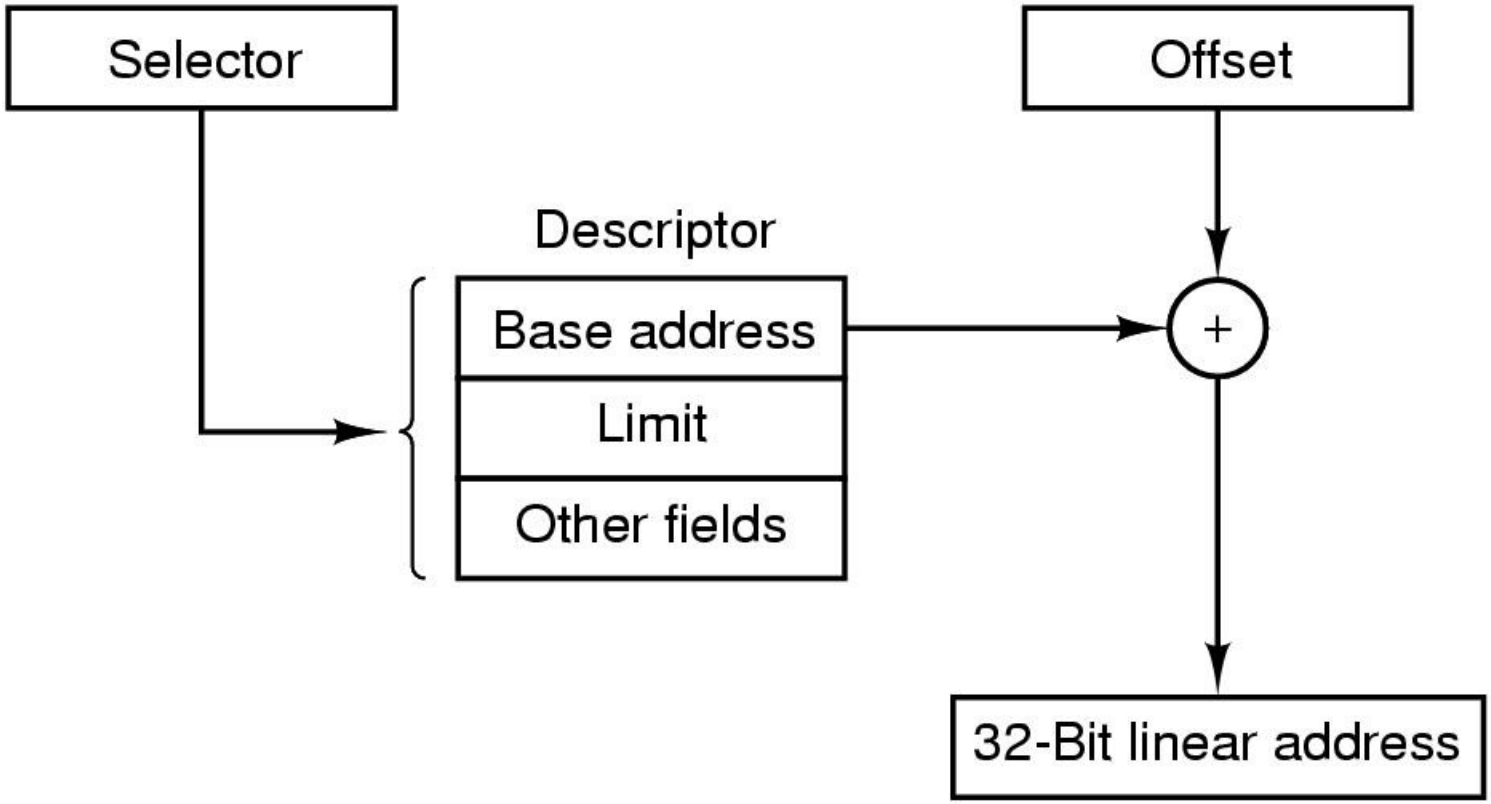


Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

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

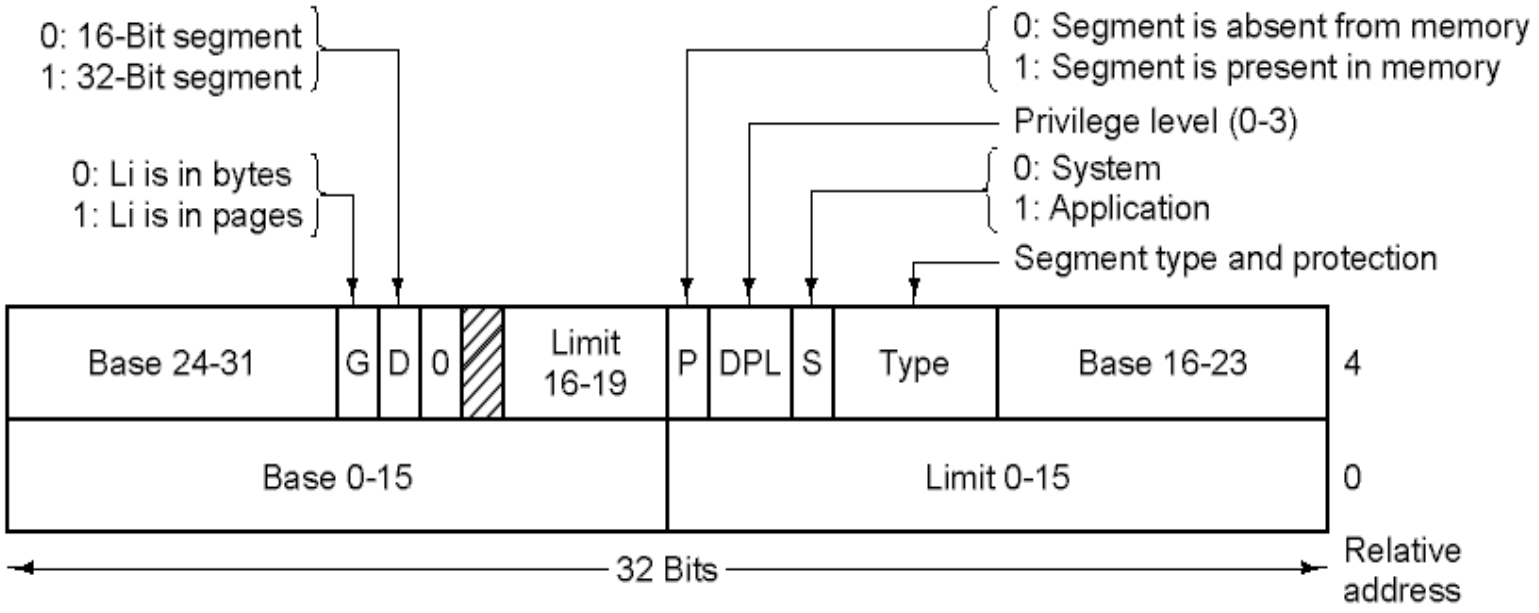
Spare slides

Segmentation & paging in the Pentium



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

Segmentation & paging in the Pentium



- Pentium segment descriptor

بسم الله الرحمن الرحیم

۱۳۷

«سیستم عامل»

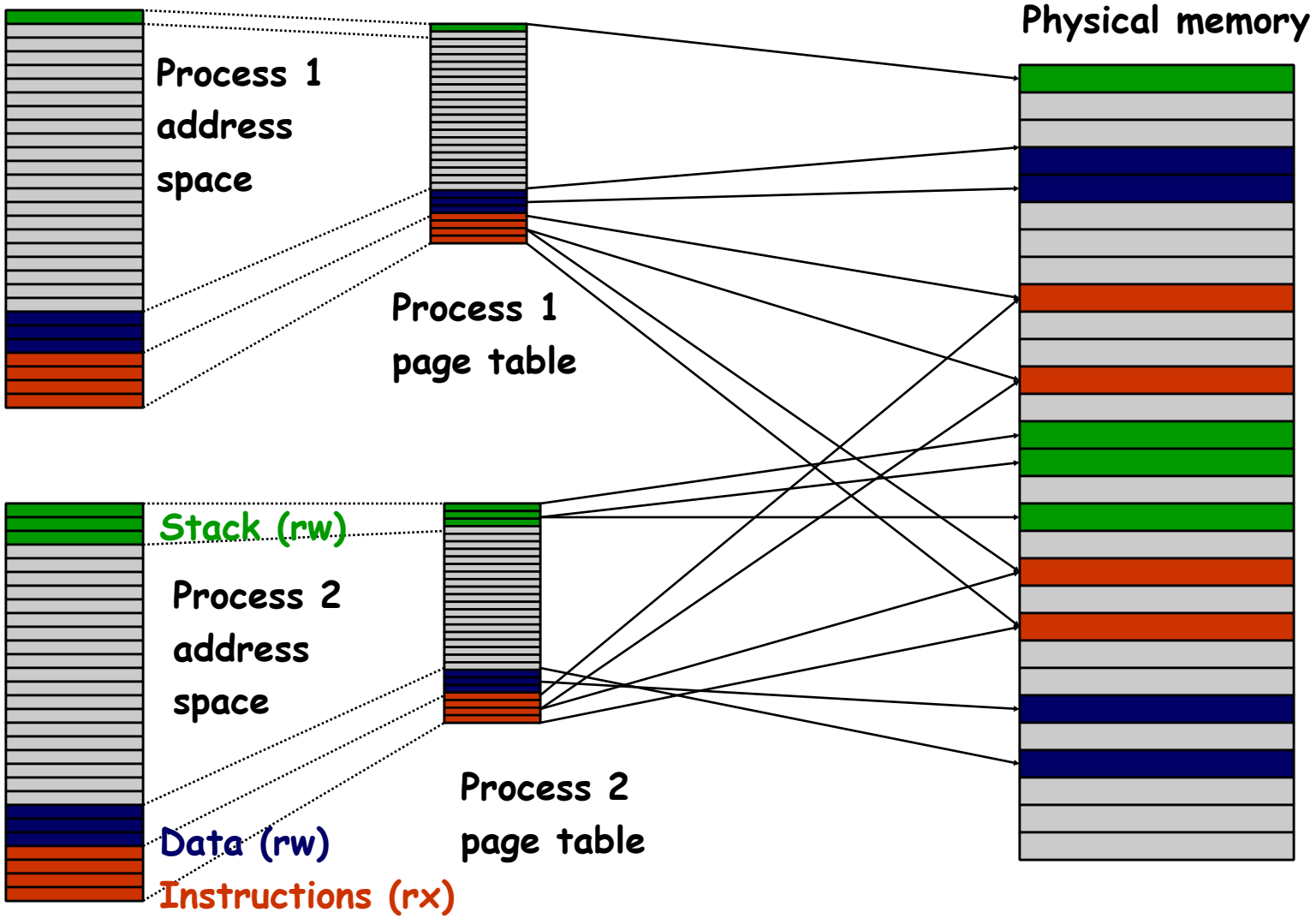
جلسه ۱۷: مدیریت حافظه (۵)

Page sharing

- **In a large multiprogramming system...**
  - Some users run the same program at the same time
    - **Why have more than one copy of the same page in memory???**

- **Goal:**
  - Share pages among "processes" (not just threads!)
    - **Cannot share writable pages**
    - **If writable pages were shared processes would notice each other's effects**
    - **Text segment can be shared**

Page sharing



Page sharing

- **"Fork" system call**
  - Copy the parent's virtual address space
    - **... and immediately do an "Exec" system call**
    - **Exec overwrites the calling address space with the contents of an executable file (ie a new program)**
  - Desired Semantics:
    - **pages are copied, not shared**
  - Observations
    - **Copying every page in an address space is expensive!**
    - **processes can't notice the difference between copying and sharing unless pages are modified!**

Page sharing

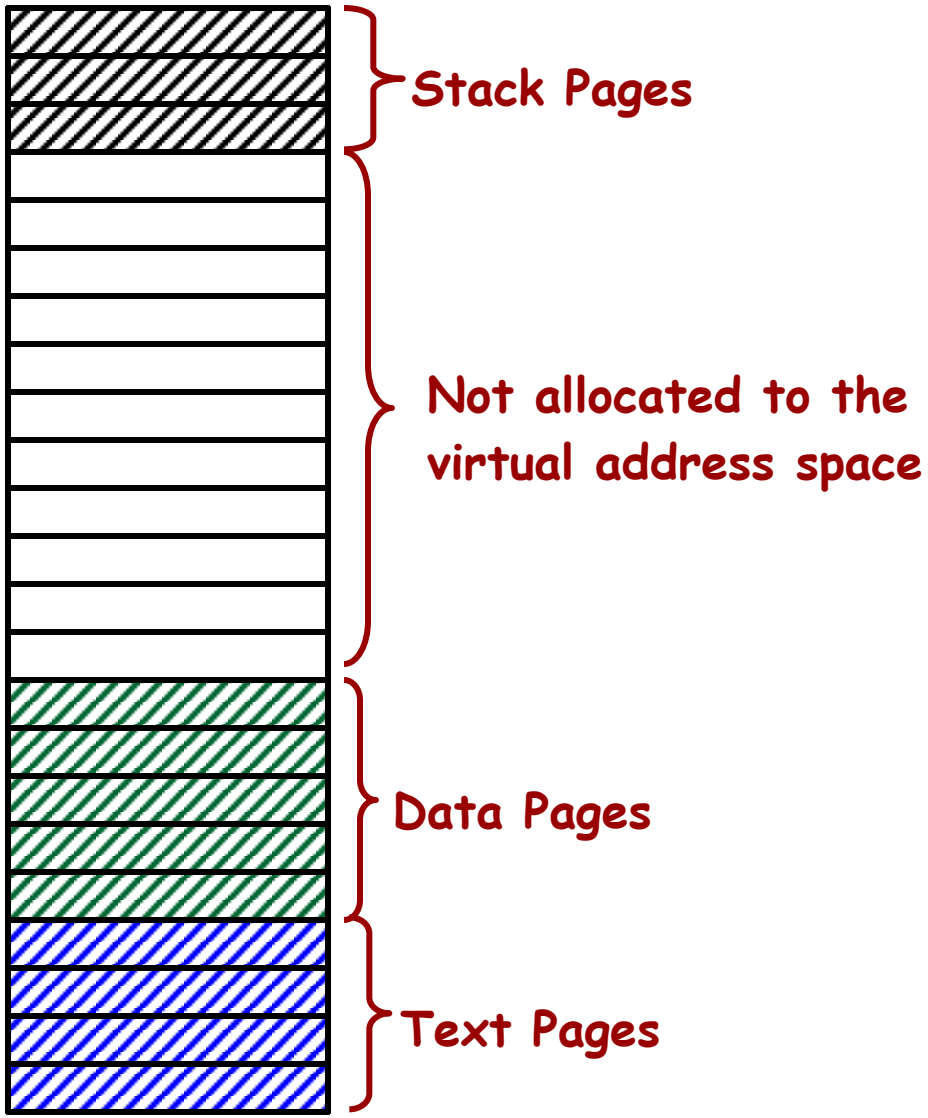
- **Idea:** **Copy-On-Write**
  - Initialize new page table, but point entries to existing page frames of parent
    - **Share pages**
  - Temporarily mark all pages "read-only"
    - **Share all pages until a protection fault occurs**
  - Protection fault (copy-on-write fault):
    - **Is this page really read only or is it writable but temporarily protected for copy-on-write?**
    - **If it is writable**
      - **copy the page**
      - **mark both copies "writable"**
      - **resume execution as if no fault occurred**

New System Calls for Page Management

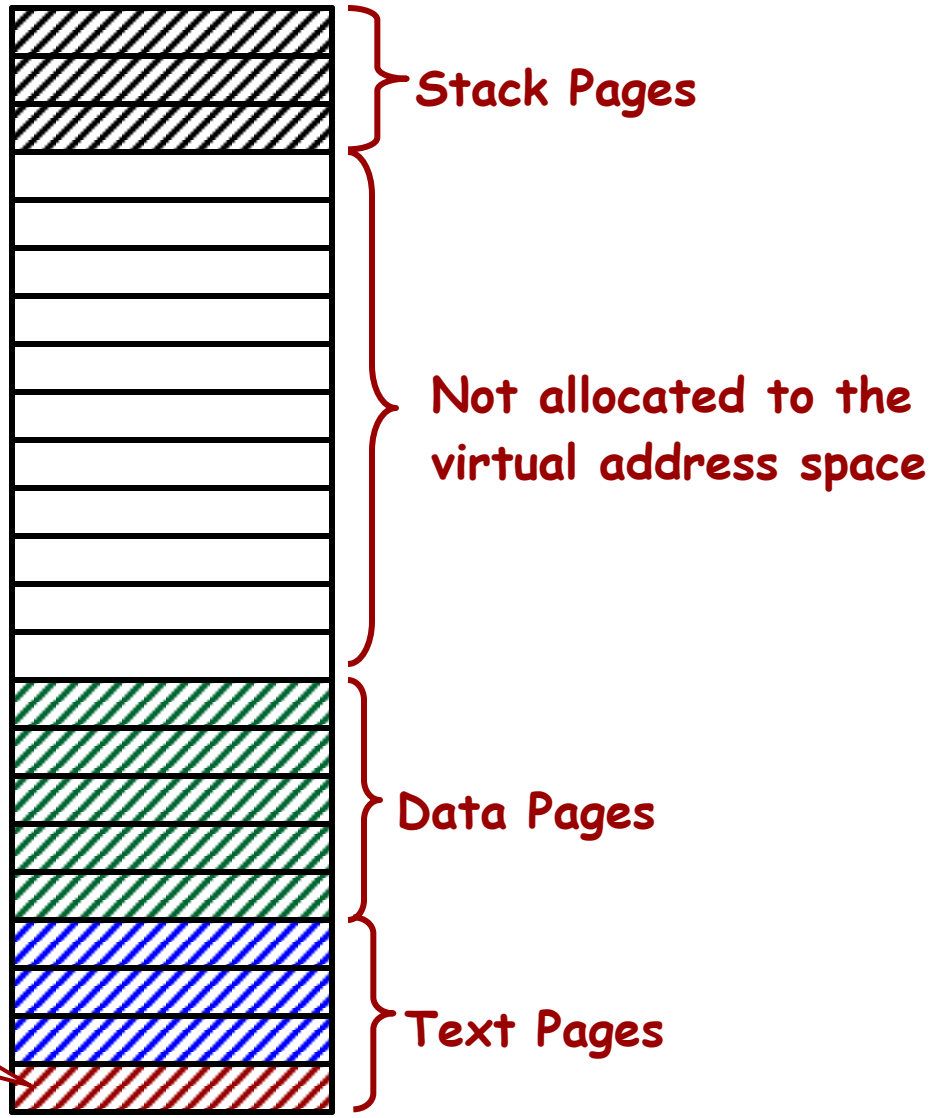
- **Goal:**
  - Allow some processes more control over paging!

- **System calls added to the kernel**
  - A process can request a page before it is needed
    - **Allows processes to grow (heap, stack etc)**
  - Processes can share pages
    - **Allows fast communication of data between processes**
    - **Similar to how threads share memory**
      - **… so what is the difference?**

Unix processes



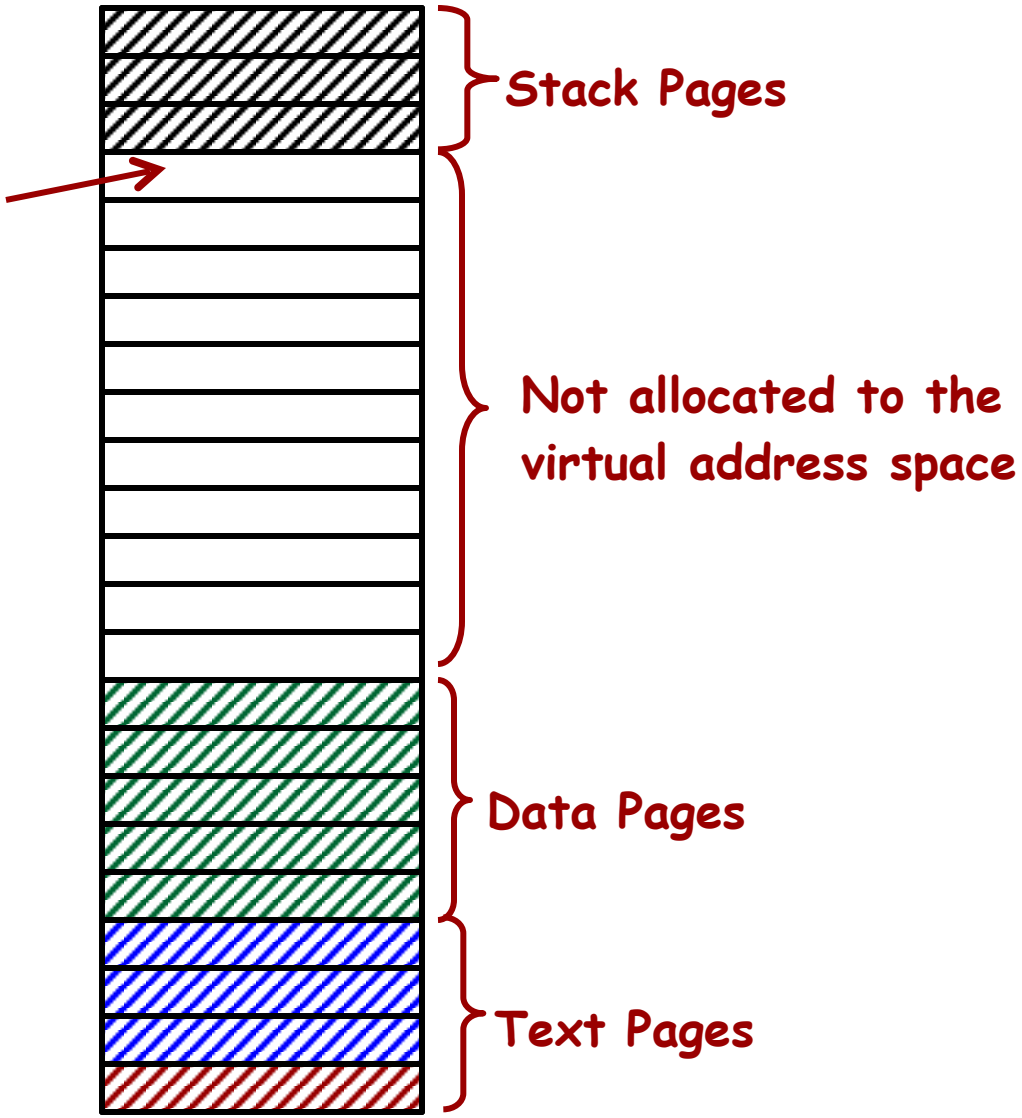
Unix processes



Unix processes

The stack grows;

Page requested here



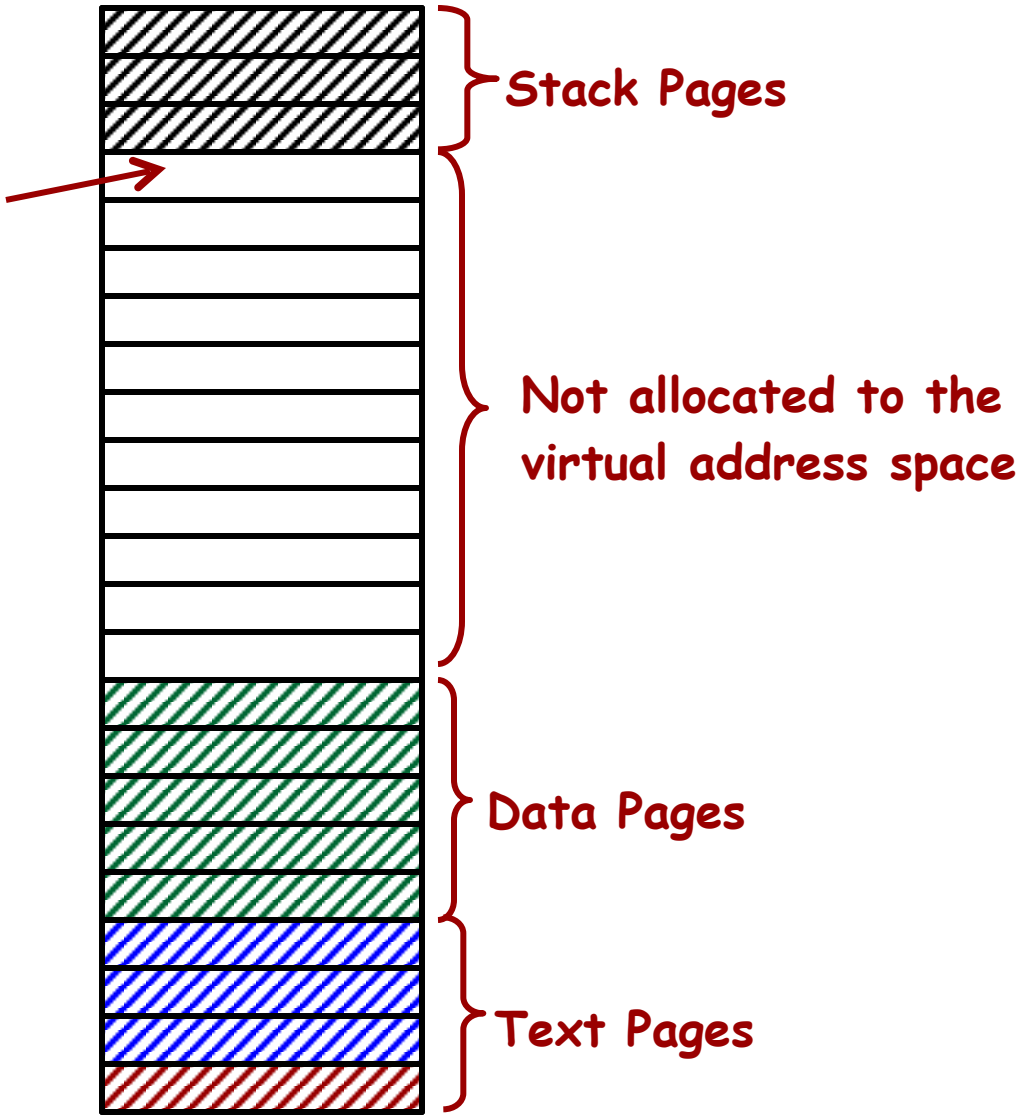
Unix processes

The stack grows;

Page requested here

A new page is allocated

and process continues



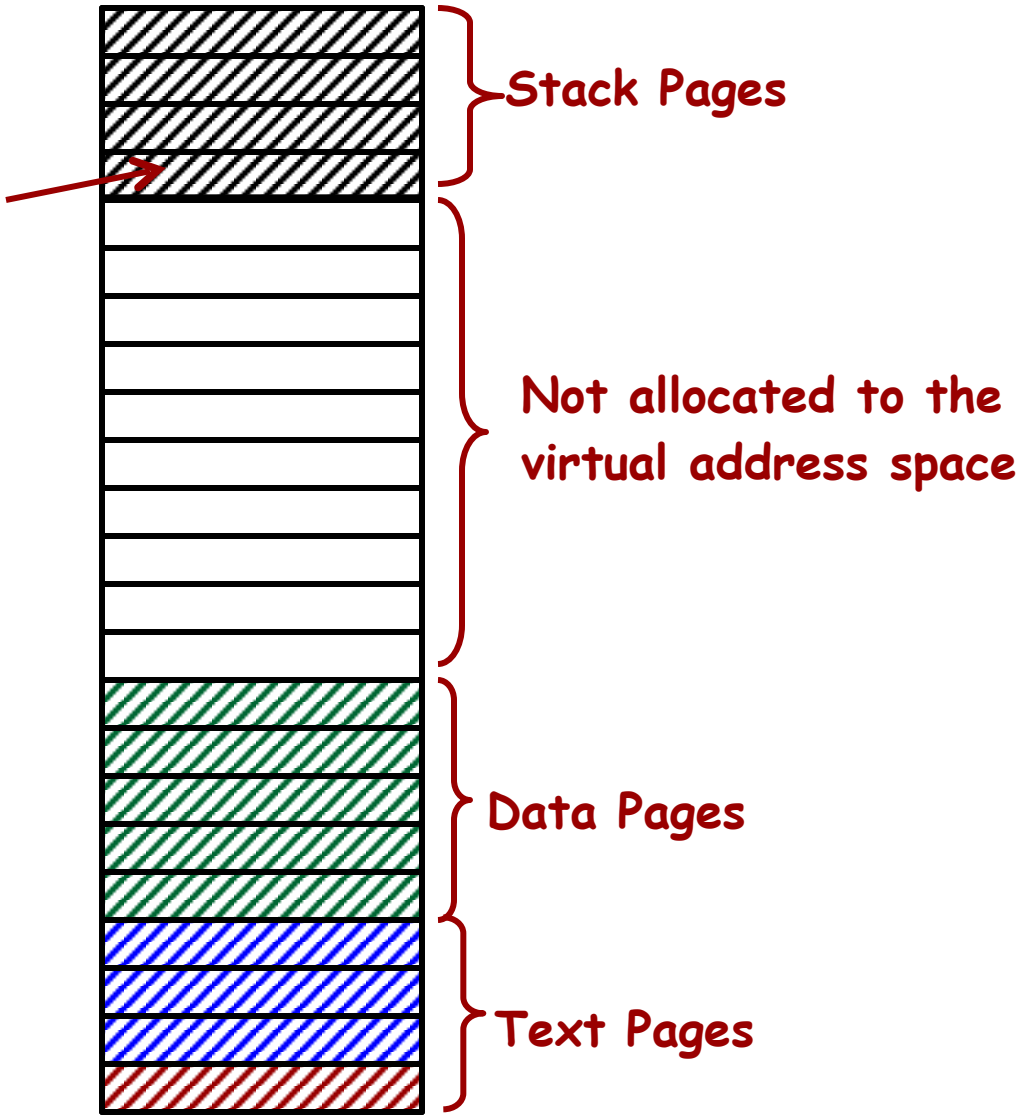
Unix processes

The stack grows;

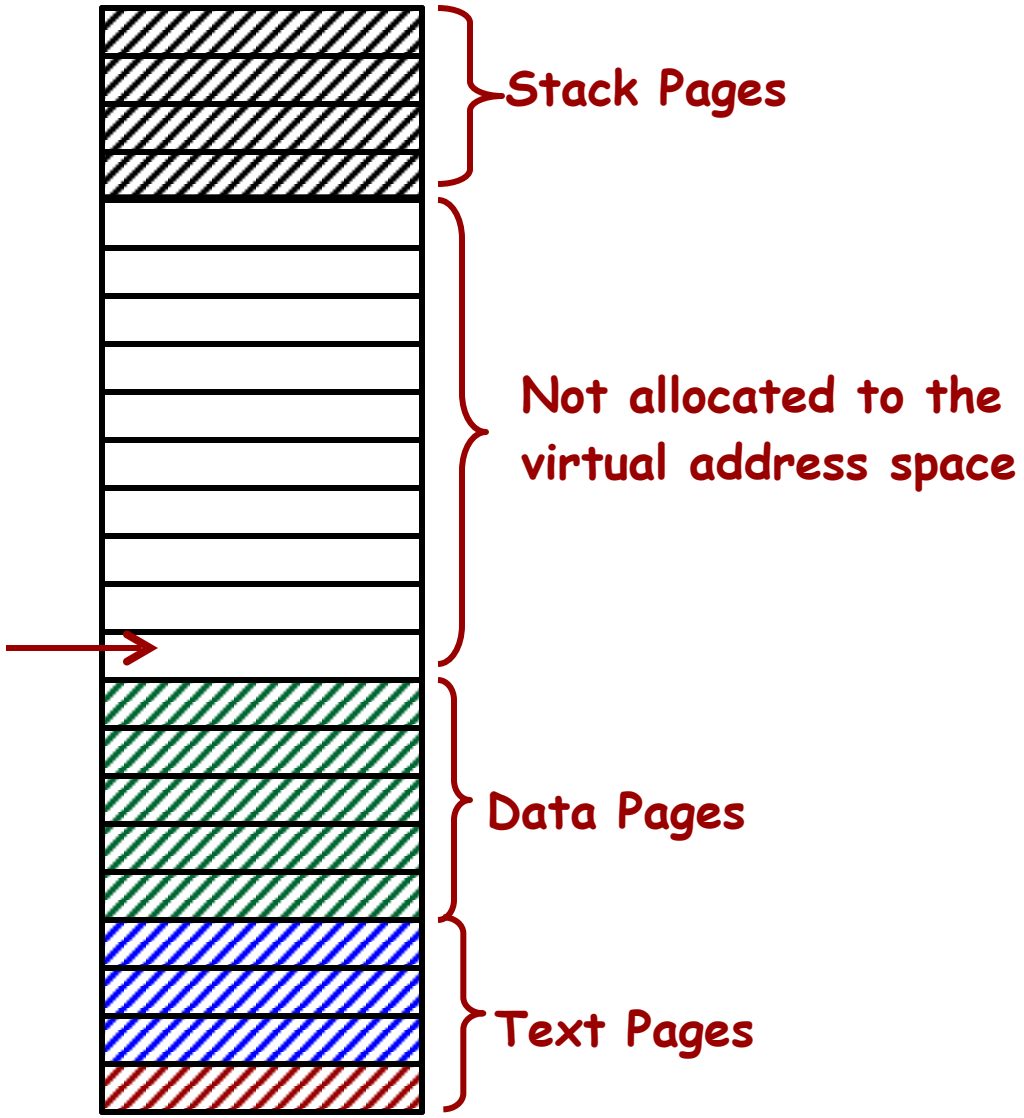
Page requested here

A new page is allocated

and process continues



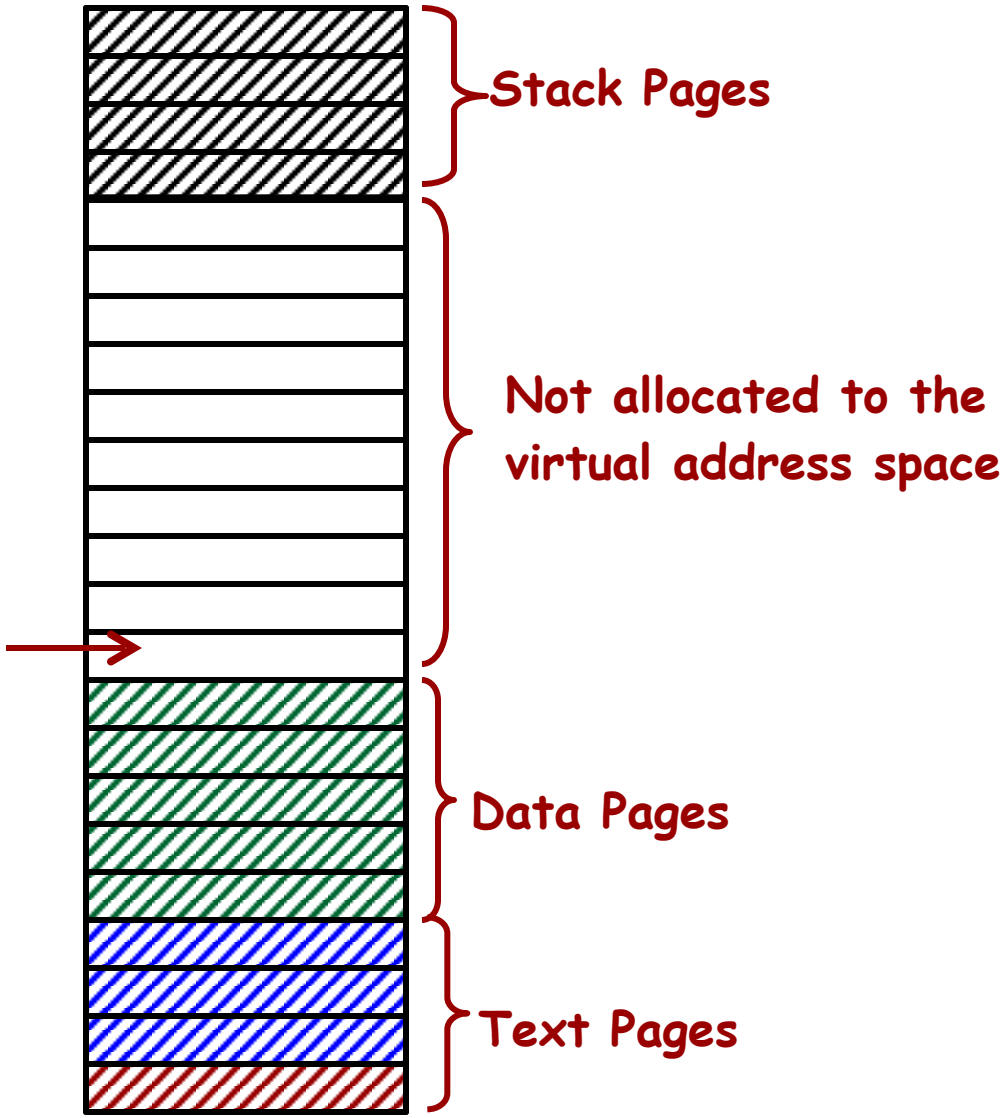
Unix processes



The heap grows;

Page requested here

Unix processes



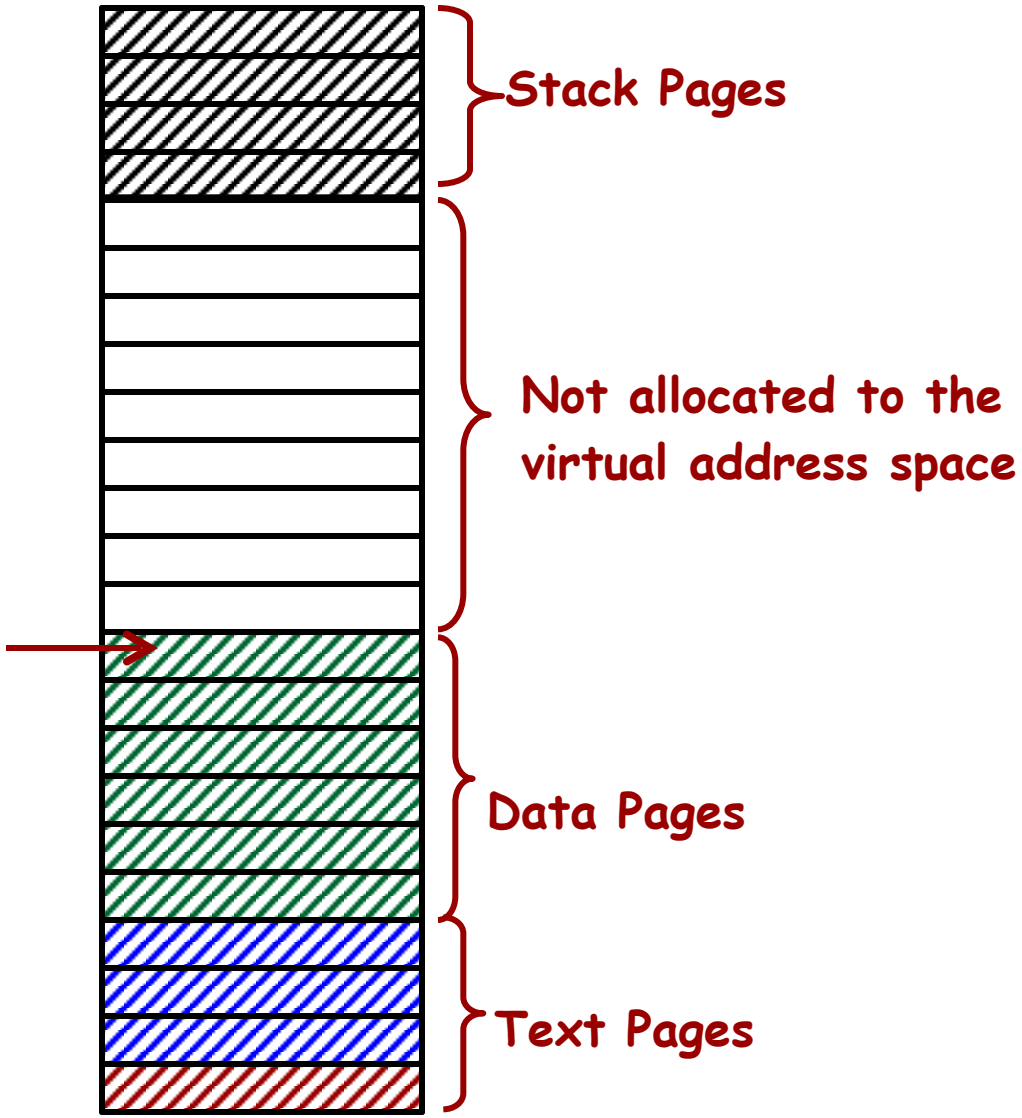
The heap grows;

Page requested here

A new page is allocated

and process continues

Unix processes



The heap grows;

Page requested here

A new page is allocated

and process continues

Virtual memory implementation

- When is the kernel involved?

Virtual memory implementation

- **When is the kernel involved?**
  - Process creation
  - Process is scheduled to run
  - A fault occurs
  - Process termination

Virtual memory implementation

- **Process creation**
  - Determine the process size
  - Create new page table

Virtual memory implementation

- **Process is scheduled to run**
  - MMU is initialized to point to new page table
  - TLB is flushed (unless it's a tagged TLB)

Virtual memory implementation

- **A fault occurs**
  - Could be a TLB-miss fault, segmentation fault, protection fault, copy-on-write fault …
  - Determine the virtual address causing the problem
  - Determine whether access is allowed, if not terminate the process
  - Refill TLB (TLB-miss fault)
  - Copy page and reset protections (copy-on-write fault)
  - Swap an evicted page out & read in the desired page (page fault)

Virtual memory implementation

- **Process termination**
  - Release / free all frames (if reference count is zero)
  - Release / free the page table

Handling a page fault

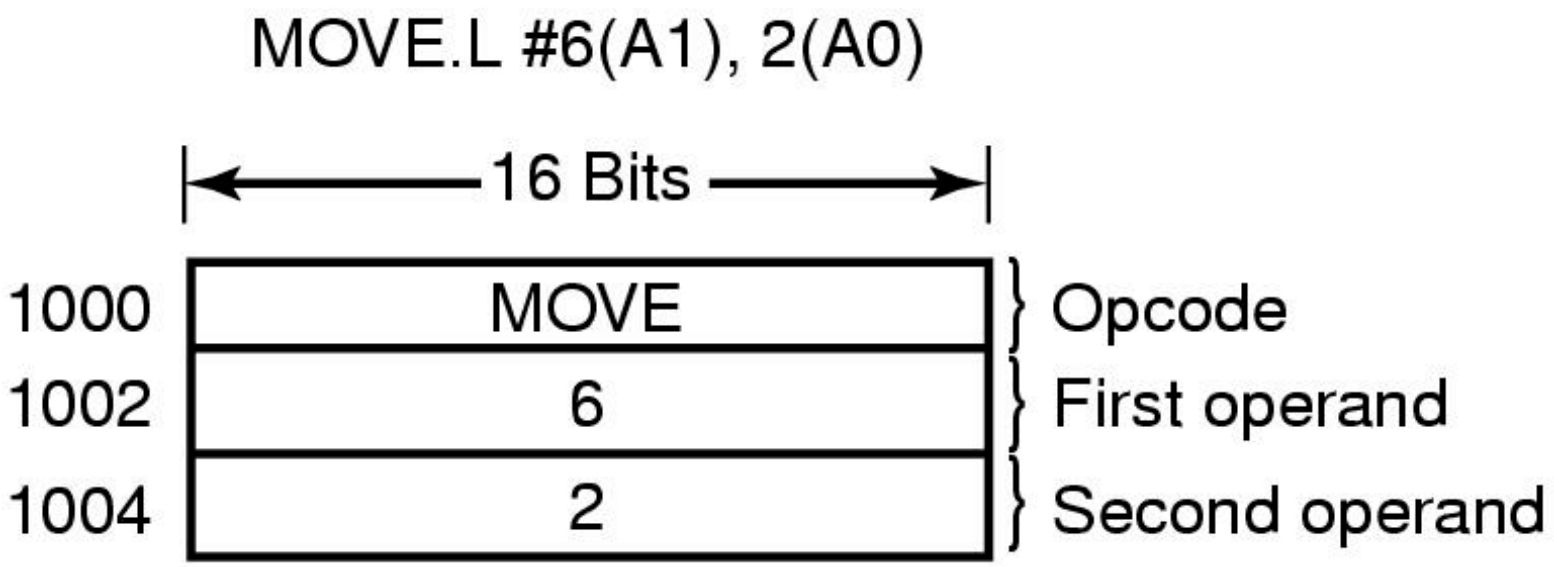
- **Hardware traps to kernel**
  - PC and SR are saved on stack
- **Save the other registers**
- **Determine the virtual address causing the problem**
- **Check validity of the address**
  - determine which page is needed
  - may need to kill the process if address is invalid
- **Find the frame to use (page replacement algorithm)**
- **Is the page in the target frame dirty?**
  - If so, write it out (& schedule other processes)
- **Read in the desired frame from swapping file**
- **Update the page tables**
- **(continued)**

Handling a page fault

- **Back up the current instruction**
  - The "faulting instruction"
- **Schedule the faulting process to run again**
- **Return to scheduler**
- **...**
- **Reload registers**
- **Resume execution**

Backing the PC up to restart an instruction

- Consider a multi-word instruction.
- The instruction makes several memory accesses.
- One of them faults.
- The value of the PC depends on when the fault occurred.
- How can you know what instruction was executing???



Solutions

- **Lot's of clever code in the kernel**

- **Hardware support (precise interrupts)**
  - Dump internal CPU state into special registers
  - Make "hidden" registers accessible to kernel

- **What if you swapped out the page containing the first operand in order to bring in the second?**

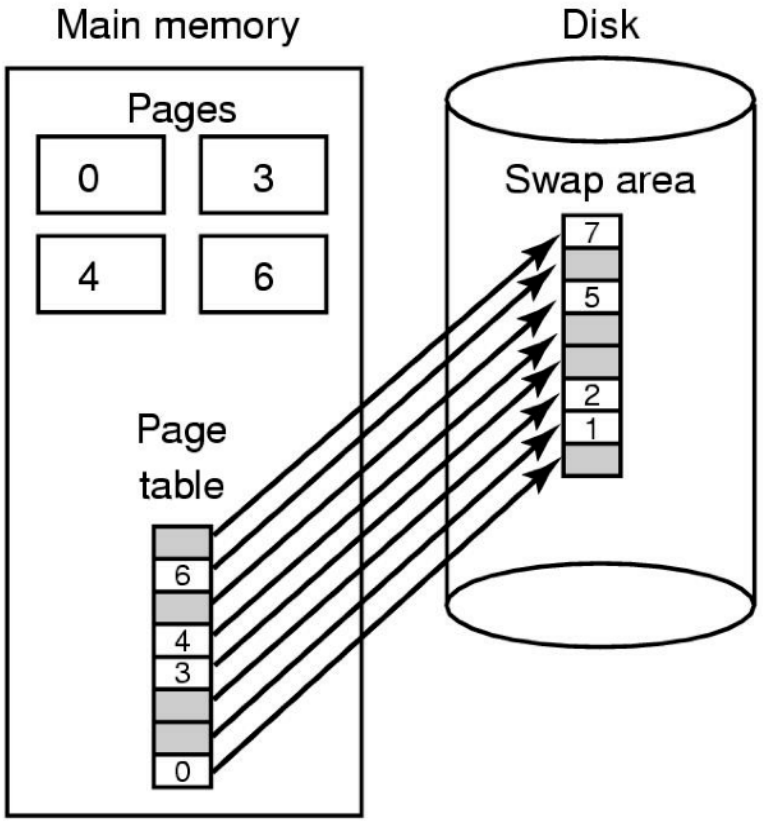
Locking pages in memory

- **Virtual memory and I/O interact**
  - Requires "Pinning" pages
- **Example:**
  - One process does a read system call
    - **(This process suspends during I/O)**
  - Another process runs
    - **It has a page fault**
    - **Some page is selected for eviction**
    - **The frame selected contains the page involved in the read**
- **Solution:**
  - Each frame has a flag: "Do not evict me".
  - Must always remember to un-pin the page!

Managing the swap area on disk

- **Approach #1:**
  - A process starts up
    - **Assume it has N pages in its virtual address space**
  - A region of the swap area is set aside for the pages
  - There are N pages in the swap region
  - The pages are kept in order
  - For each process, we need to know:
    - **Disk address of page 0**
    - **Number of pages in address space**
  - Each page is either...
    - **In a memory frame**
    - **Stored on disk**

Approach #1



Problem

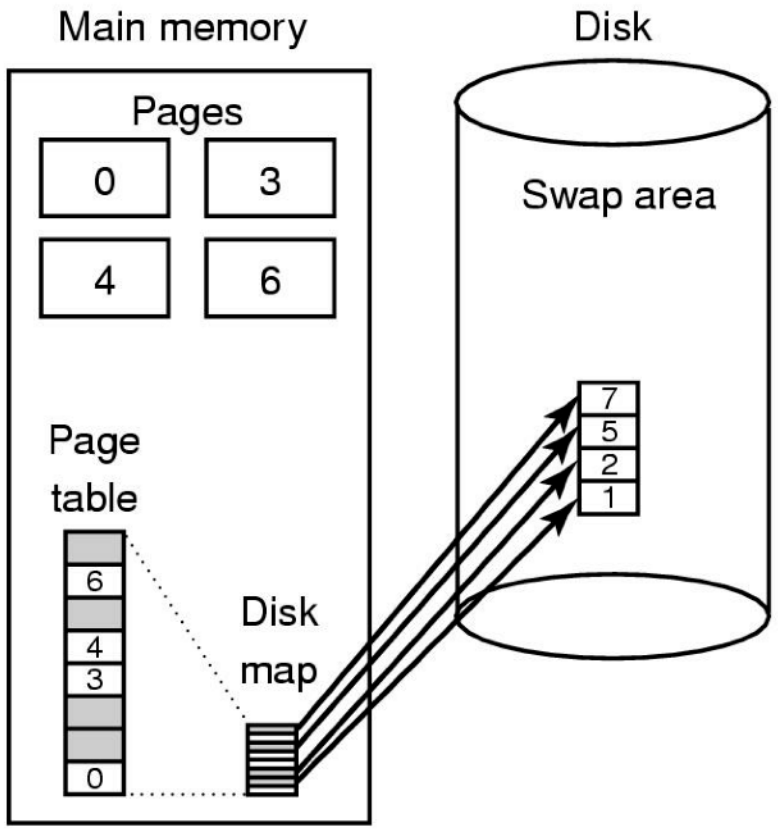
- **What if the virtual address space grows during execution? i.e. more pages are allocated.**

- **Approach #2**
  - Store the pages in the swap in a random order.
  - View the swap file as a collection of free "swap frames".
  - Need to evict a frame from memory?
    - **Find a free "swap frame".**
    - **Write the page to this place on the disk.**
    - **Make a note of where the page is.**
    - **Use the page table entry.**
      - **Just make sure the valid bit is still zero!**
  - Next time the page is swapped out, it may be written somewhere else.

Approach #2

This picture uses a separate data structure to tell where pages are stored on disk rather than using the page table

Some information, such as protection status, could be stored at segment granularity



Approach #3

- **Swap to a file**
  - Each process has its own swap file
  - File system manages disk layout of files

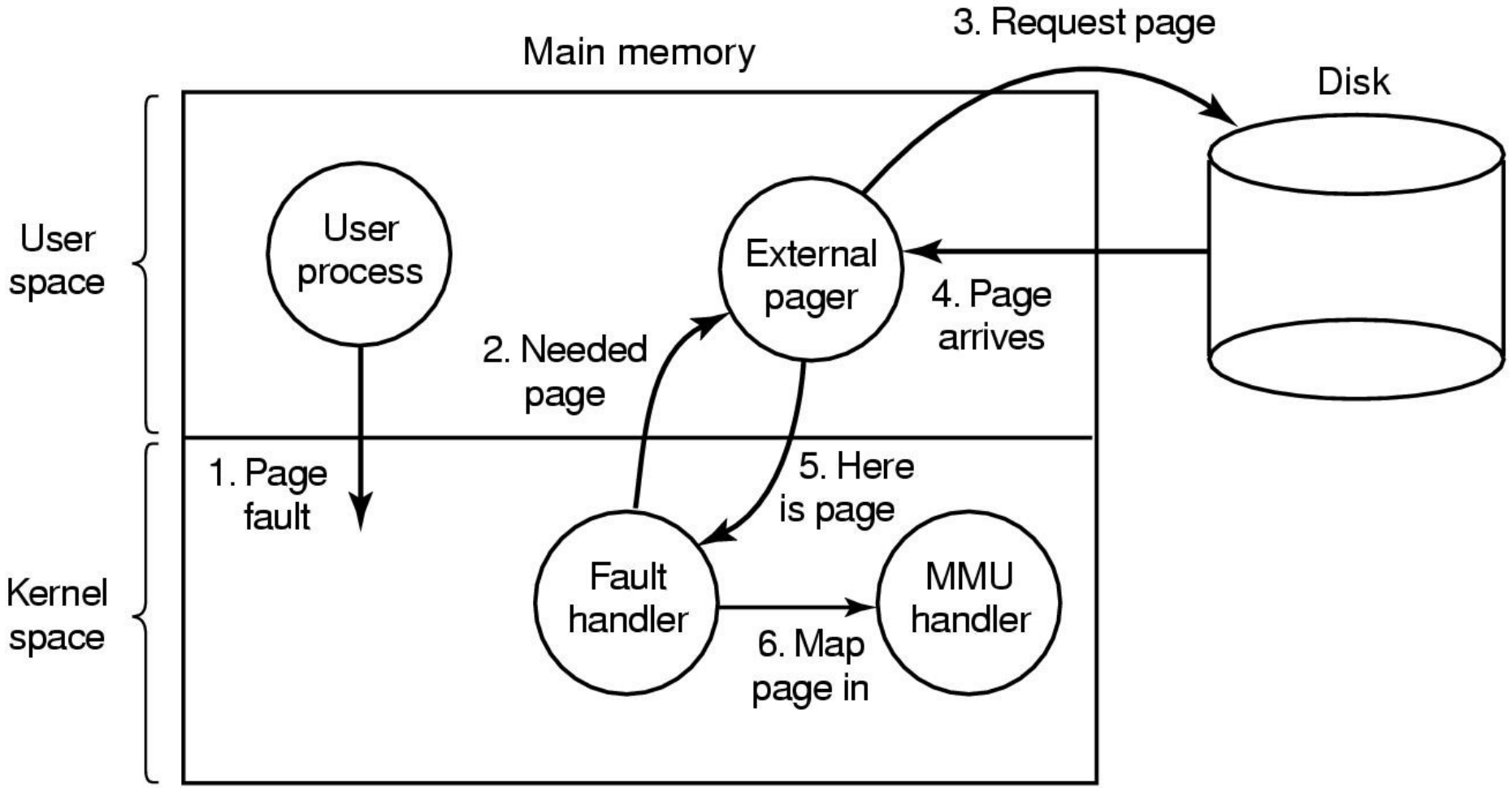
Approach #4

- **Swap to an external pager process (object)**
- **A user-level "External Pager" process can determine policy**
  - Which page to evict
  - When to perform disk I/O
  - How to manage the swap file
- **When the OS needs to read in or write out a page it sends a message to the external pager**
  - Which may even reside on a different machine
- **Examples: Mach, Minix**

Separation of Policy and Mechanism

- **Kernel contains**
  - Code to interact with the MMU
    - **This code tends to be machine dependent**
  - Code to handle page faults
    - **This code tends to be machine independent**

Separation of Policy and Mechanism



Paging performance

- **Paging works best if there are plenty of free frames.**
- **If all pages are full of dirty pages...**
  - Must perform 2 disk operations for each page fault
- **It's a good idea to periodically write out dirty pages in order to speed up page fault handling delay**

Paging daemon

- **Page Daemon**
  - A kernel process
  - Wakes up periodically
  - Counts the number of free page frames
  - If too few, run the page replacement algorithm...
    - **Select a page & write it to disk**
    - **Mark the page as clean**
  - If this page is needed later... then it is still there.
  - If an empty frame is needed later... this page is evicted.