Caching and Memory Management

OPERATING SYSTEMS COURSE
THE HEBREW UNIVERSITY
SPRING 2018

Caching

OPERATING SYSTEMS COURSE
THE HEBREW UNIVERSITY
SPRING 2017

Address Translation

- CPU understands only main memory addresses.
- Cache size is smaller than main memory size.
- How to map main memory addresses to a locations in the cache?

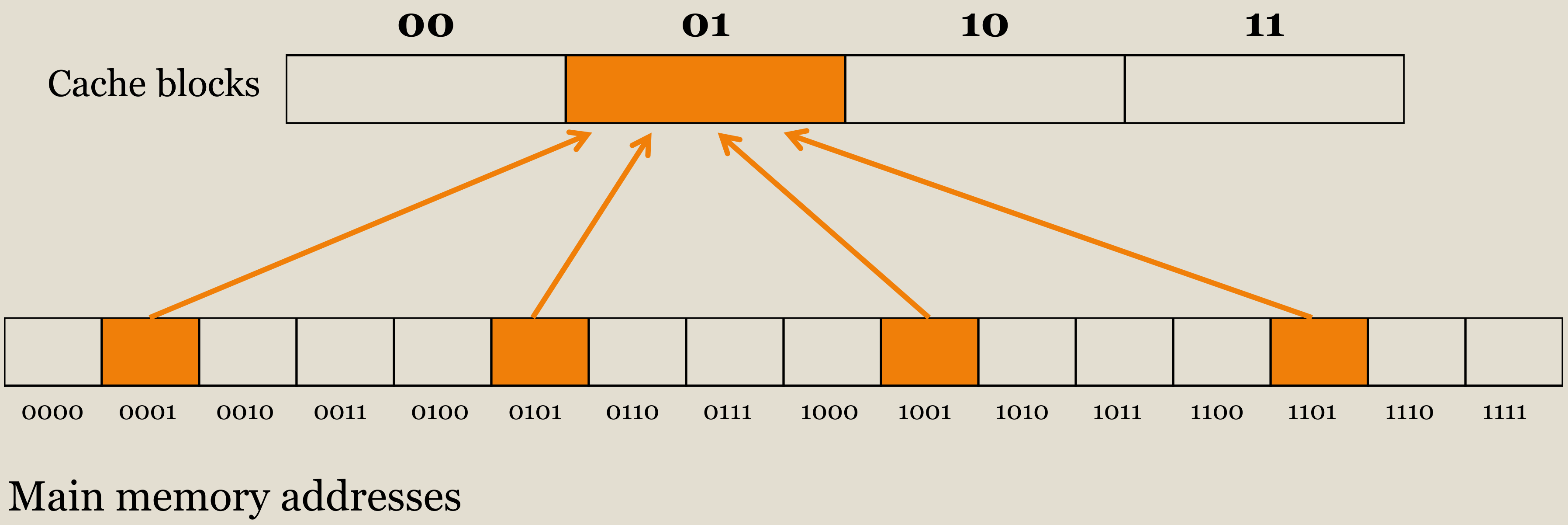
- Each main memory address contains a **word**.
- A cache location may store a **block** (several consecutive words).
  - Might increase hit rate (spatial locality).
  - Reduce tags overhead.

Direct mapping

- Each block is mapped to exactly one cache location:

  Cache location = (block address) MOD (# of blocks in cache)

- An example (cache block size = 1 word):

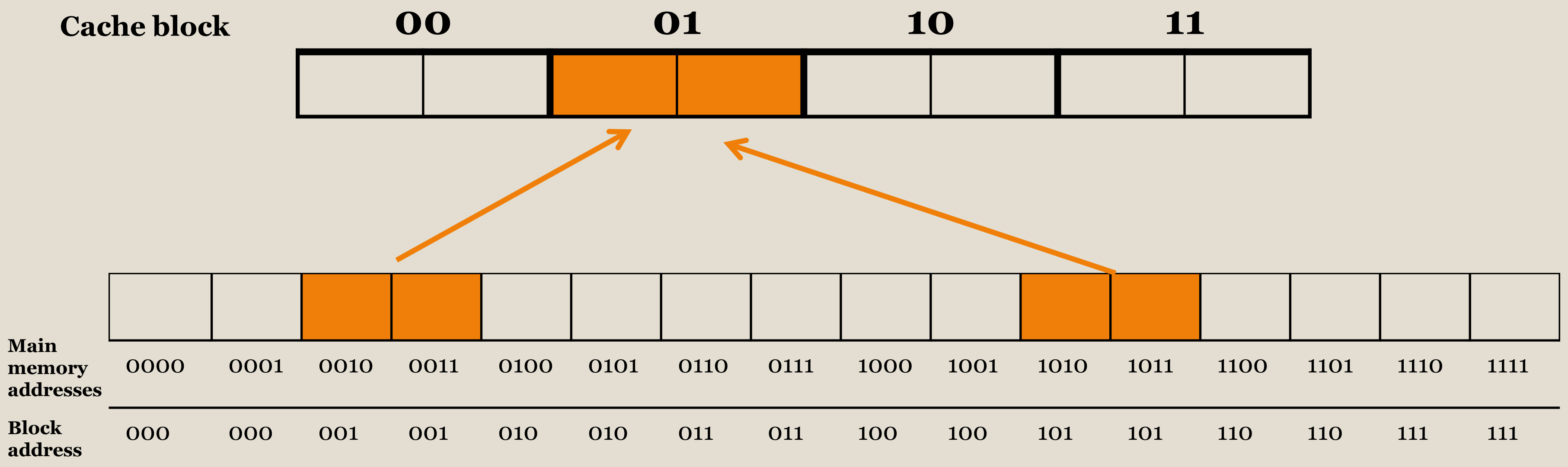


Direct mapping

- When cache block contains more than 1 word:
  - We need to convert the main memory address to block number.
  - then calculate the cache location as before.

  Cache location = (block address) MOD (# of blocks in cache)

- An example (cache block size = 2 words):



Direct mapping

- How do we know whether the block in the cache is the right one?
  - By using a tag.

- After locating the cache block we need to check if the cache block tag is equal to the tag part of the address.

Direct mapping

- RAM size = 4 GB.
- Word = 4 bytes.
- Cache size = 4 MB.
- Block size = 8 words.
- Direct mapping cache.

- Where would memory address 5221 be found?
  - #words in cache = $2^{22}/4 = 2^{20}$.
  - #blocks in cache = $2^{20}/8 = 2^{17}$.
  - Index = $\log_2$(#blocks in cache) = 17 bits.
  - Offset = $\log_2$(block size) = 3 bits.
  - Tag = $\log_2$(#words in RAM) – index – offset = 30-17-3 = 10.

  - 5221 = **0000000000** **00000001010001100** **101**

Direct mapping

- RAM size = 4 GB.
- Word = 4 bytes.
- Cache size = 4 MB.
- Block size = 8 words.
- Direct mapping cache.

- Where would memory address 5221 be found?
  - 5221 = $(\mathbf{000000000000000001010001100101})_2$
  - $(00000001010001100)_2$ = 652.
  - If the tag of cache[652] = $(0000000000)_2$ then fetch the fifth word ($(101)_2$ = 5).

2^x–way Associativity

- The cache is divided into sets.
- Each set contains 2^x blocks.
- Each memory address is mapped to one set (in the same way as direct mapping), but the data could be stored in every block of this set.

- **Fully associativity**: the block can be stored in every location in the cache: if the cache size is 16 blocks, then 16–ways associativity == fully associativity.

2^x–way Associativity

- RAM size = 4 GB.
- Word = 4 bytes.
- Cache size = 4 MB.
- Block size = 8 words.
- 16-way associative cache.

- Where could memory address 5221 be found?
  - #words in cache = $2^{22}/4 = 2^{20}$.
  - #blocks in cache = $2^{20}/8 = 2^{17}$.
  - #sets in cache = $2^{17}/16 = 2^{13}$.
  - Index = $\log_2$(#sets in cache) = 13 bits.
  - Offset = $\log_2$(block size) = 3 bits.
  - Tag = $\log_2$(#words in RAM) – index – offset = 30-13-3 = 14.

  - 5221 = **00000000000000** **0001010001100** **101**

2^x–way Associativity

- RAM size = 4 GB.
- Word = 4 bytes.
- Cache size = 4 MB.
- Block size = 8 words.
- 16-way associative cache.

- Where could memory address 5221 be found?
  - 5221 = $(\mathbf{000000000000000001010001100101})_2$
  - $(0001010001100)_2 = 652$.
  - If one of the 16 blocks in cache[652] has tag $(00000000000000)_2$ then fetch the fifth word ($(101)_2 = 5$) from it.

Assume:

1. A word is 4 bytes.
2. Main memory is of size 2^{20} words.
3. Cache is of size 2^{8} words.
4. Each block is of size 2^{3} words.
5. Cache is 4-way set associative.

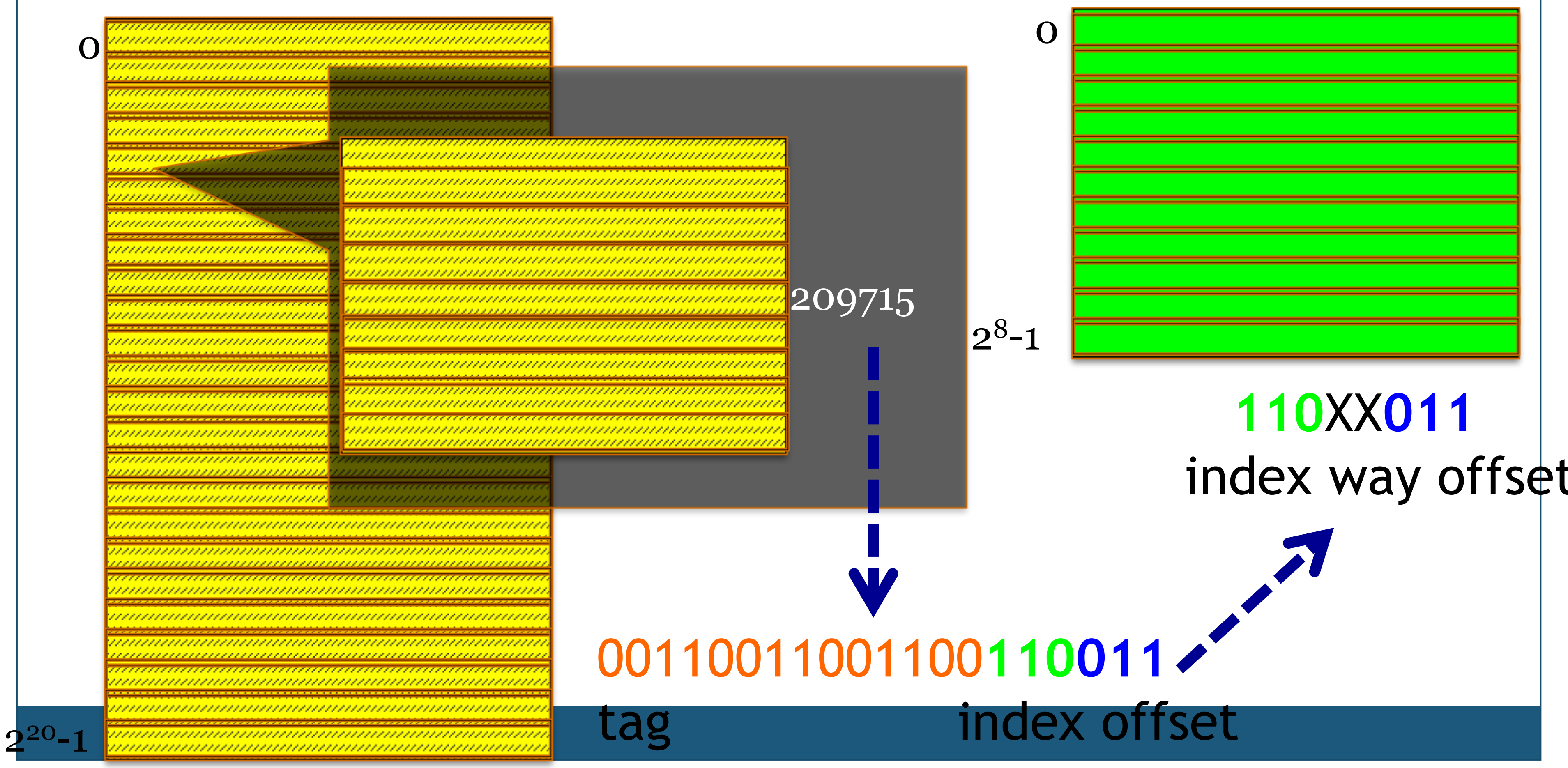
Suppose we want to access physical address 209715
(binary = 00110011001100110011).

What are all the possible cache physical address it can be stored?
(mark all possible location)

a. 10101101
b. 11000011
c. 11011011
d. 00110011

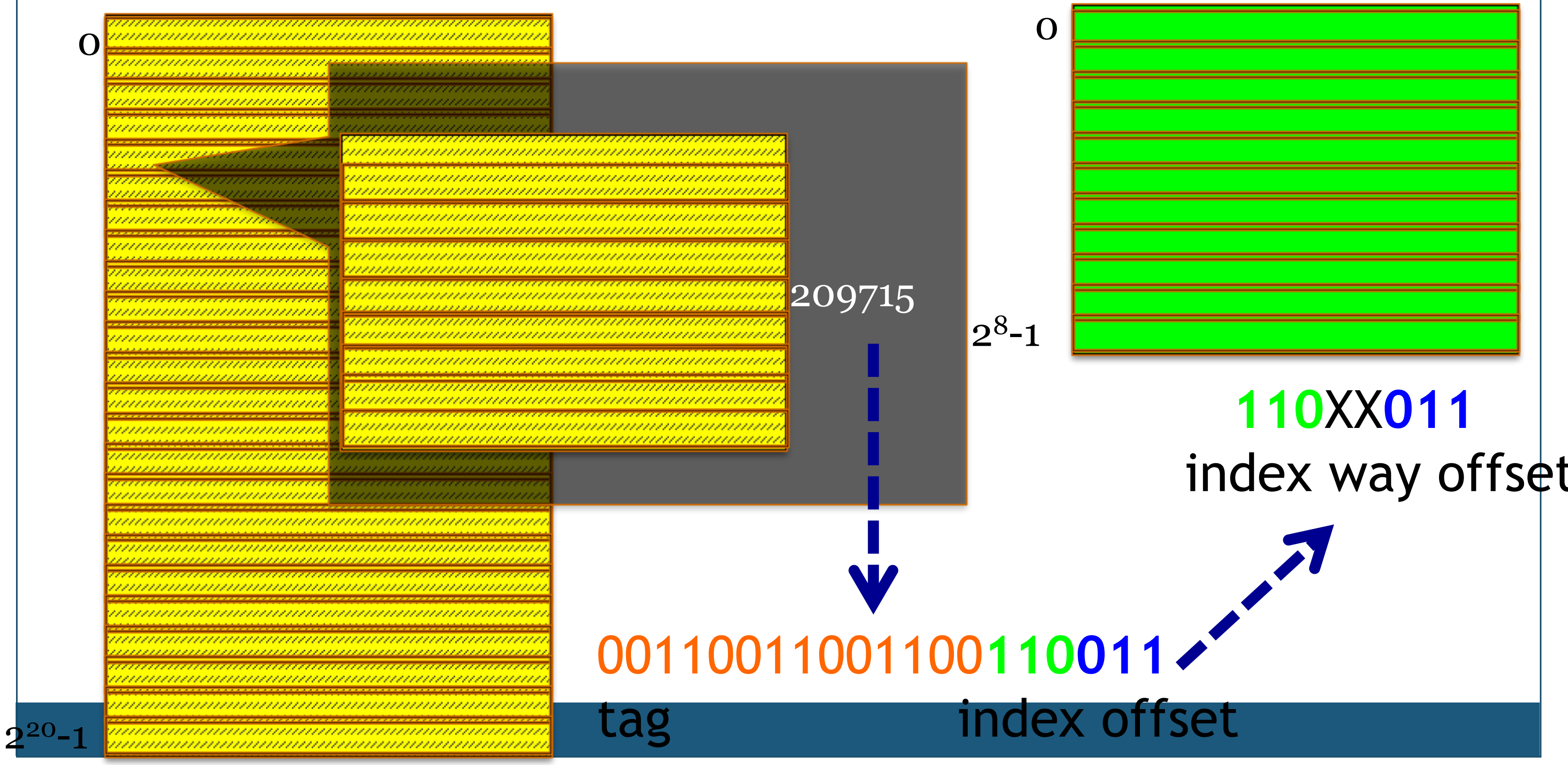
Assume:

1. A word is 4 bytes.
2. Main memory is of size 2^{20} words.
3. Cache is of size 2^8 words.
4. Each block is of size 2^3 words.
5. Cache is 4-way set associative.



Main memory address 209715 can be in cache address:

1. 195 (11000011) – answer b
2. 203 (11001011)
3. 211 (11010011)
4. 219 (11011011) – answer c
5. Not in the cache.



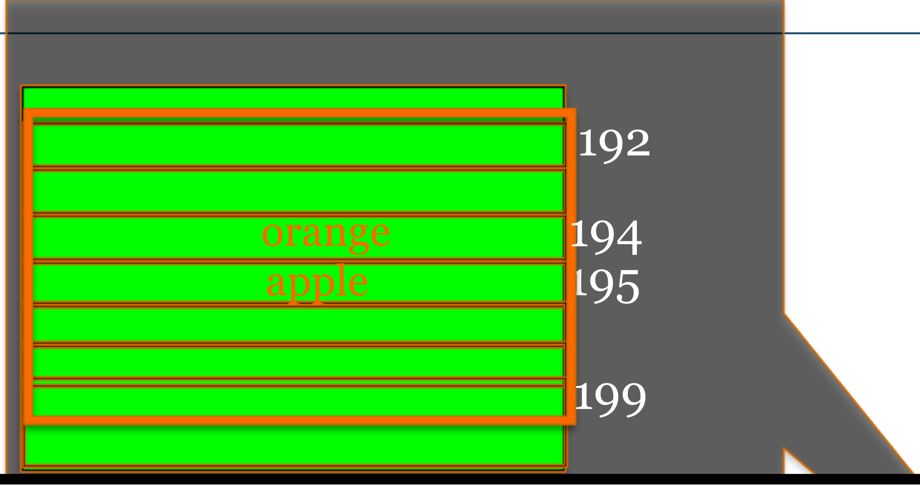
Assume:

1. A word is 4 bytes.
2. Main memory is of size 2^{20} words.
3. Cache is of size 2^8 words.
4. Each block is of size 2^3 words.
5. Cache is 4-way set associative.

Suppose we have a cache hit for physical address 209715 (binary = 00110011001100110011) at cache address 195.

Reading cache address 194 will return:

a. 209715
b. 209714
c. The value at physical address 209714
d. We cannot determine for sure



00110011001100
Tag for block
11000

Read 209714 (00110011001100110010)

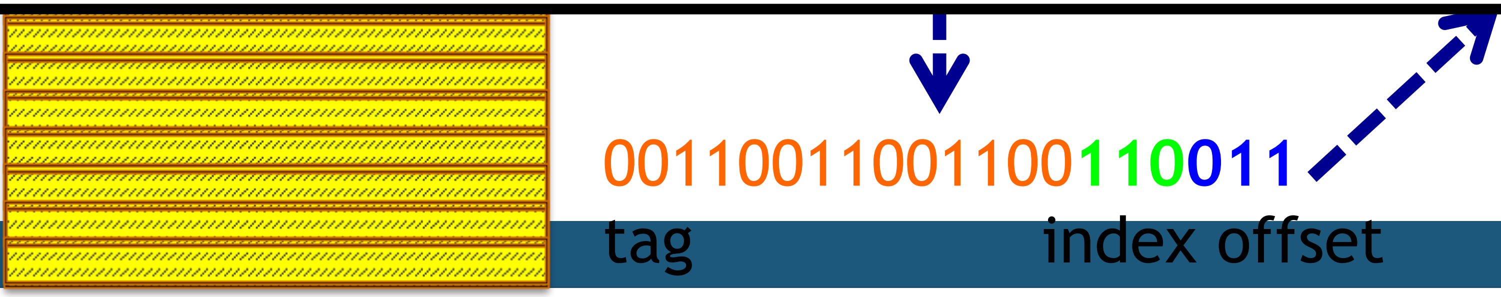
Compare tag 00110011001100 **with tag of cache block:**

11000 → HIT → Location 194 (11000010) → Return orange

11001 → MISS

11010 → MISS

11011 → MISS



Assume:

1. A word is 4 bytes.
2. Main memory is of size 2^{20} words.
3. Cache is of size 2^{8} words.
4. Each block is of size 2^{3} words.
5. Cache is 4-way set associative.

Suppose we have a cache hit for physical address 209715 (binary = 00110011001100110011) at cache address 195.

Reading cache address 194 will return:

a. 209715
b. 209714
c. The value at physical address 209714
d. We cannot determine for sure

Memory Management

OPERATING SYSTEMS COURSE
THE HEBREW UNIVERSITY
SPRING 2017

CPU

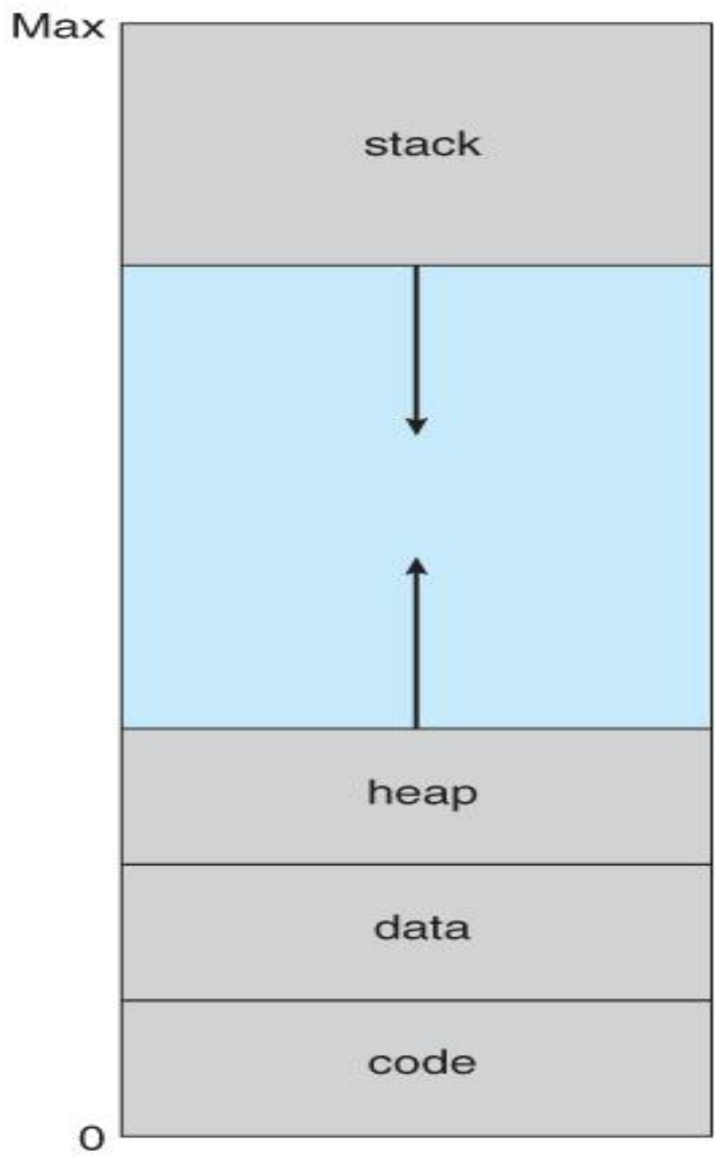
- Can only use data that is stored in registers and memory.
- Executes a set of instructions:
  - Data handling – set a register, store (register in memory), load (from memory to register)
  - Arithmetic operations on registers - Bitwise operations, compare, and basic mathematics operations.
  - Control flow using registers– branch, conditional branch
- Therefore, programs must be brought (from disk) into memory for them to be run

Virtual Address Space (VAS)

- The set of ranges of virtual addresses that an operating system makes available to a process
- Also called logical address space.
- In 32-bit operating system, the virtual address space size if 2^32 = 4GB, in 64-bit it's 2^64 (= a lot)
- For example, a pointer to a function or variable is actually a virtual address.

Virtual Address Space

Data contains static variables and globals.



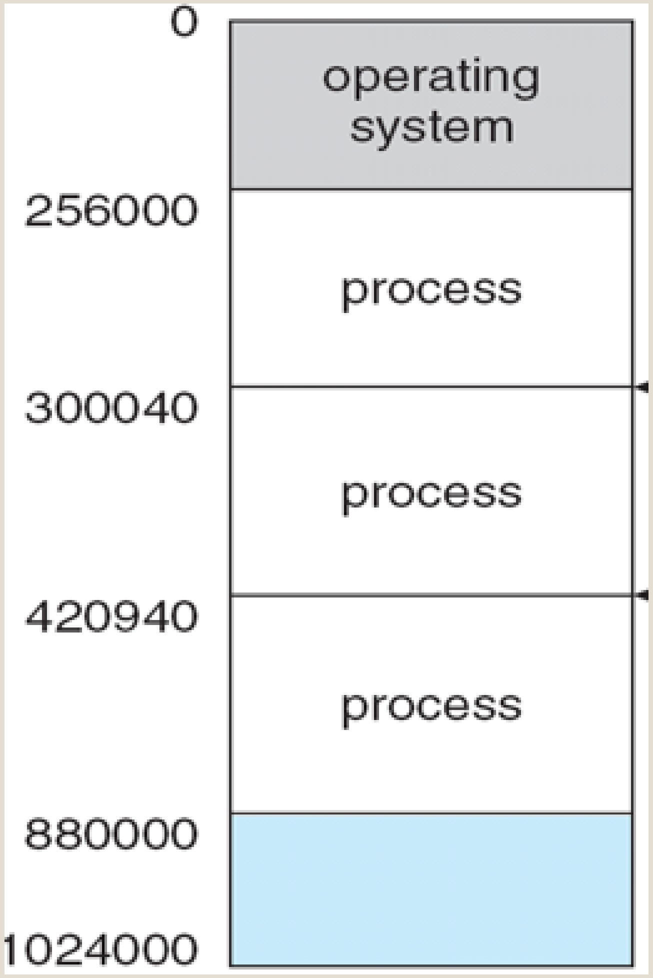
Physical Address

- The address in the particular storage cell of main memory.

- The physical address space depends on the memory size
  - If we have 8GB of RAM, the address length is 33 (log2(8GB)]).

- Also called real address, or binary address

Physical Address Basic Example



MMU

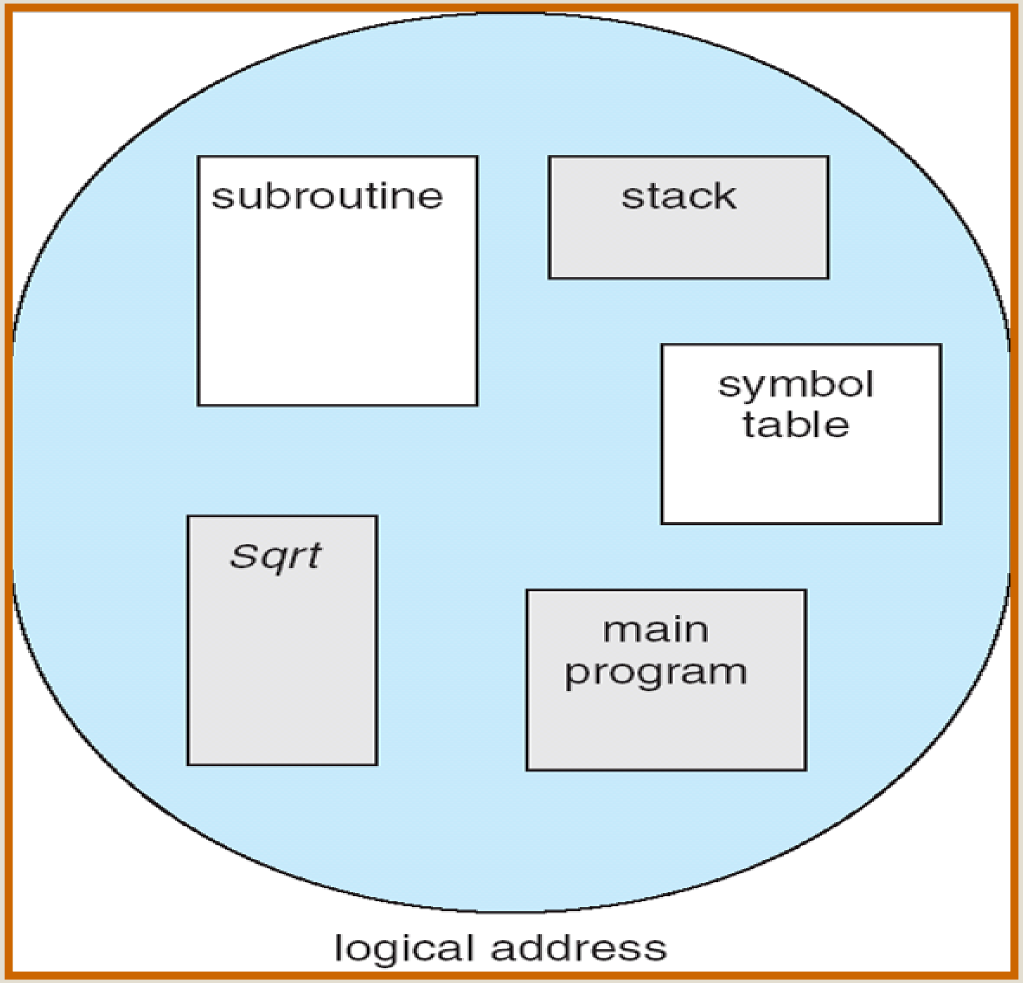
- The user program deals with *logical* addresses - it never sees the *real* physical addresses

- Memory-Management Unit (MMU) is the hardware device that maps virtual to physical address

Segmentation

- Memory segmentation is the division of a computer's primary memory into segments
- A program is a collection of segments. A segment is a logical unit such as:
  - main program
  - function
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays

User's View of a Program

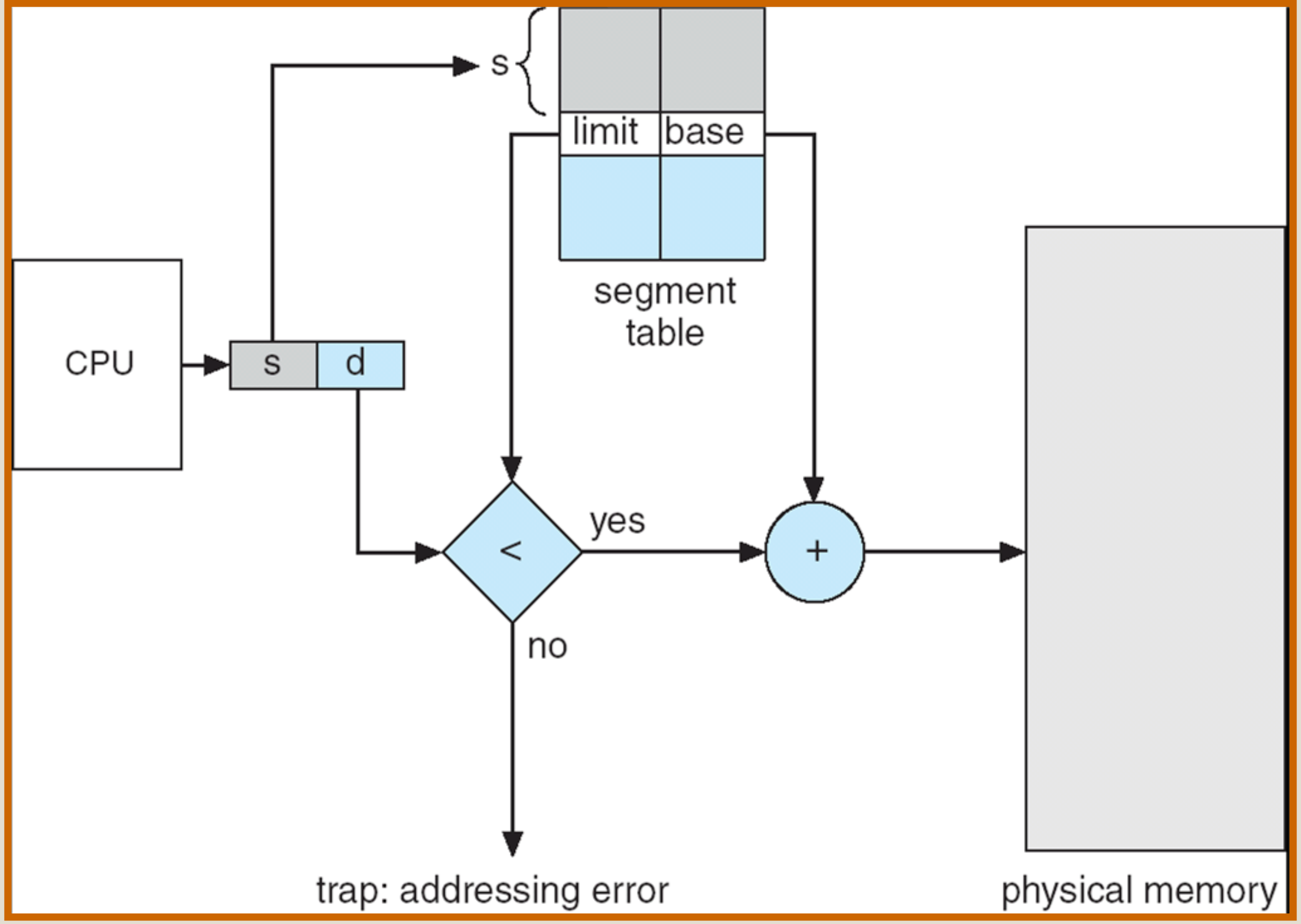


Segmentation Architecture

- A Logical address consists of a two tuple:

  <segment-number, offset>
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segment resides in memory
  - **limit** – specifies the length of the segment. Offset o is valid if o < limit
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program; segment number *s* is legal if s < STLR

Segmentation Hardware

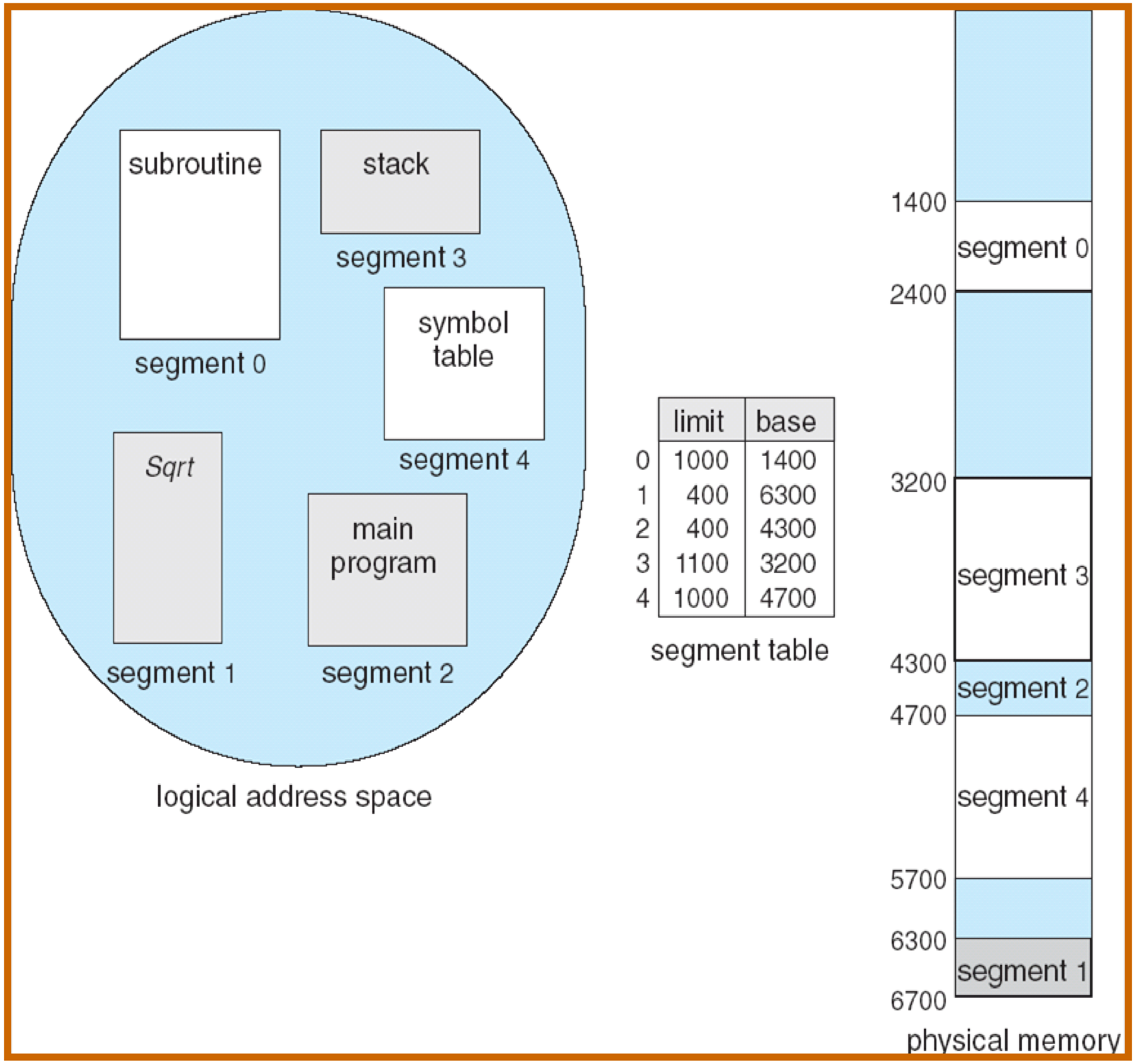


Segmentation Architecture - Cont.

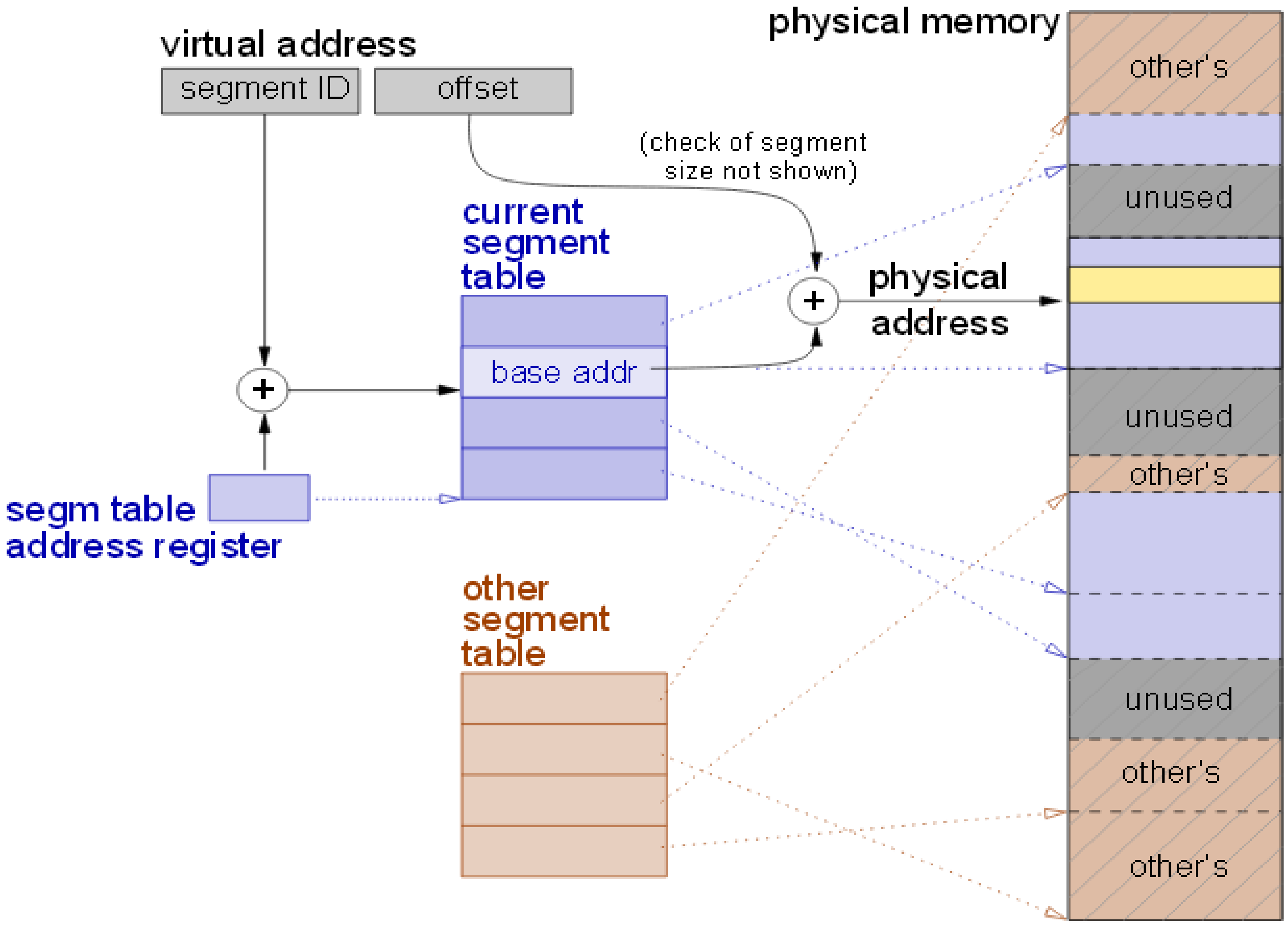
- Additional bits in each entry in the segment table:
  - validation bit (also called present-bit)
  - read/write/execute privileges

- Using the validation bit, when there is no space in the RAM, we can store segments in the disk and mark the validation bit 0.

Example of Segmentation



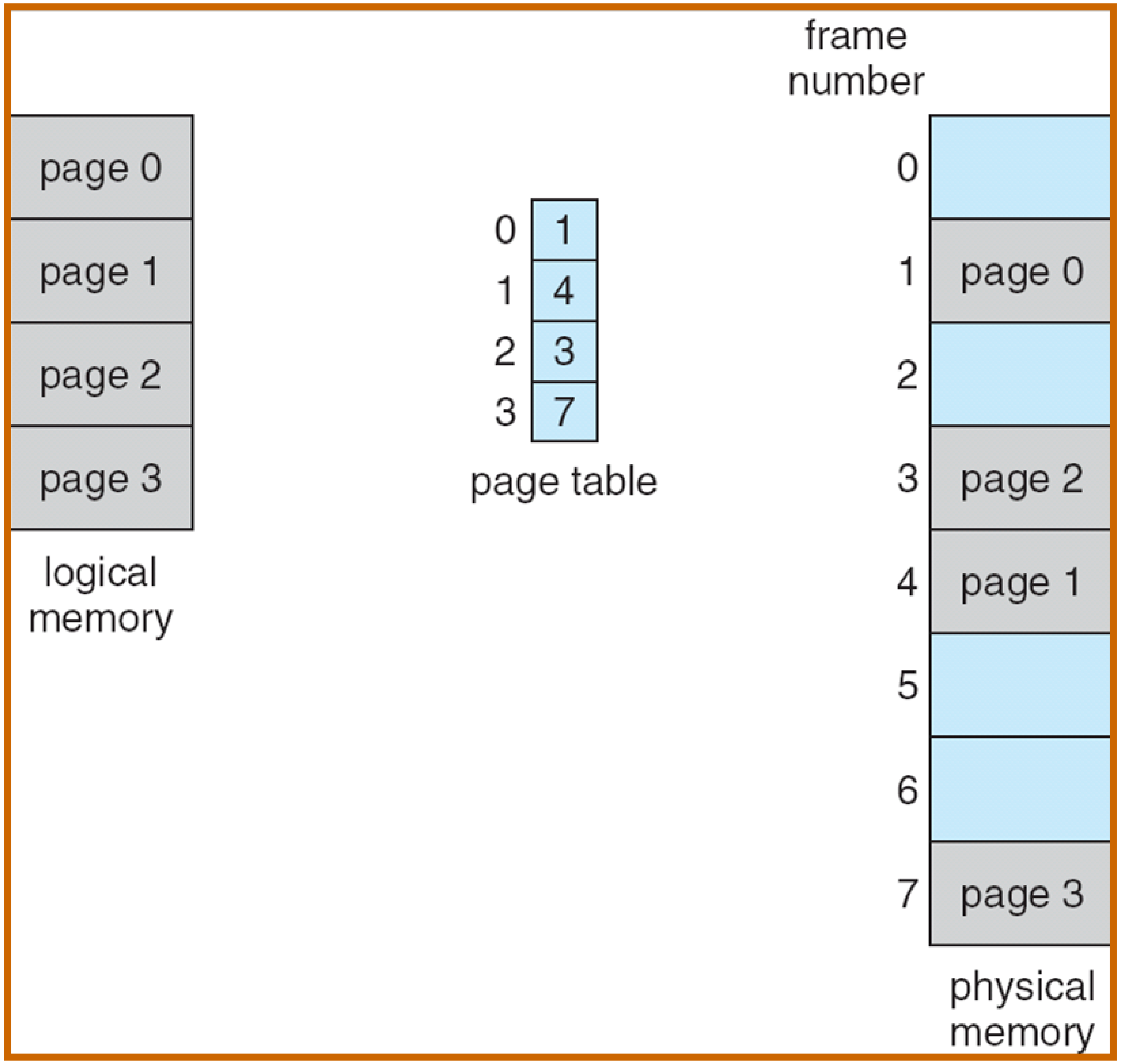
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700


virtual address
segment ID
offset
physical memory
other's
(check of segment size not shown)
unused
current segment table
physical address
+
base addr
+
unused
other's
segm table address register
other segment table
unused
other's
other's

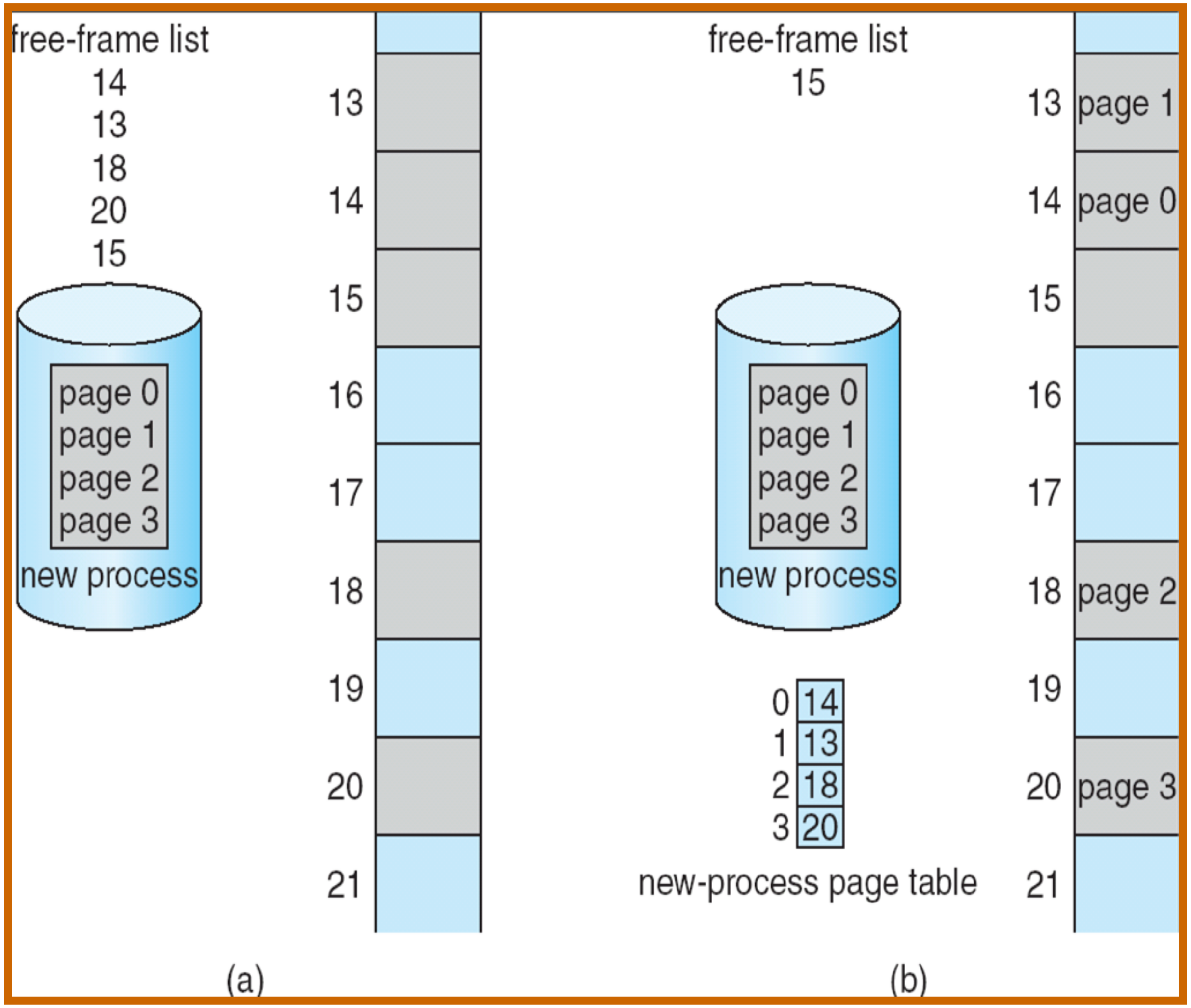
Paging

- The logical address space of process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Physical memory is partitioned into fixed-sized blocks called **frames** (size is power of 2).
- logical memory is partitioned into blocks of the same size called **pages**.
- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- No external fragmentation but there is bounded internal fragmentation.

Paging Model of Logical and Physical Memory

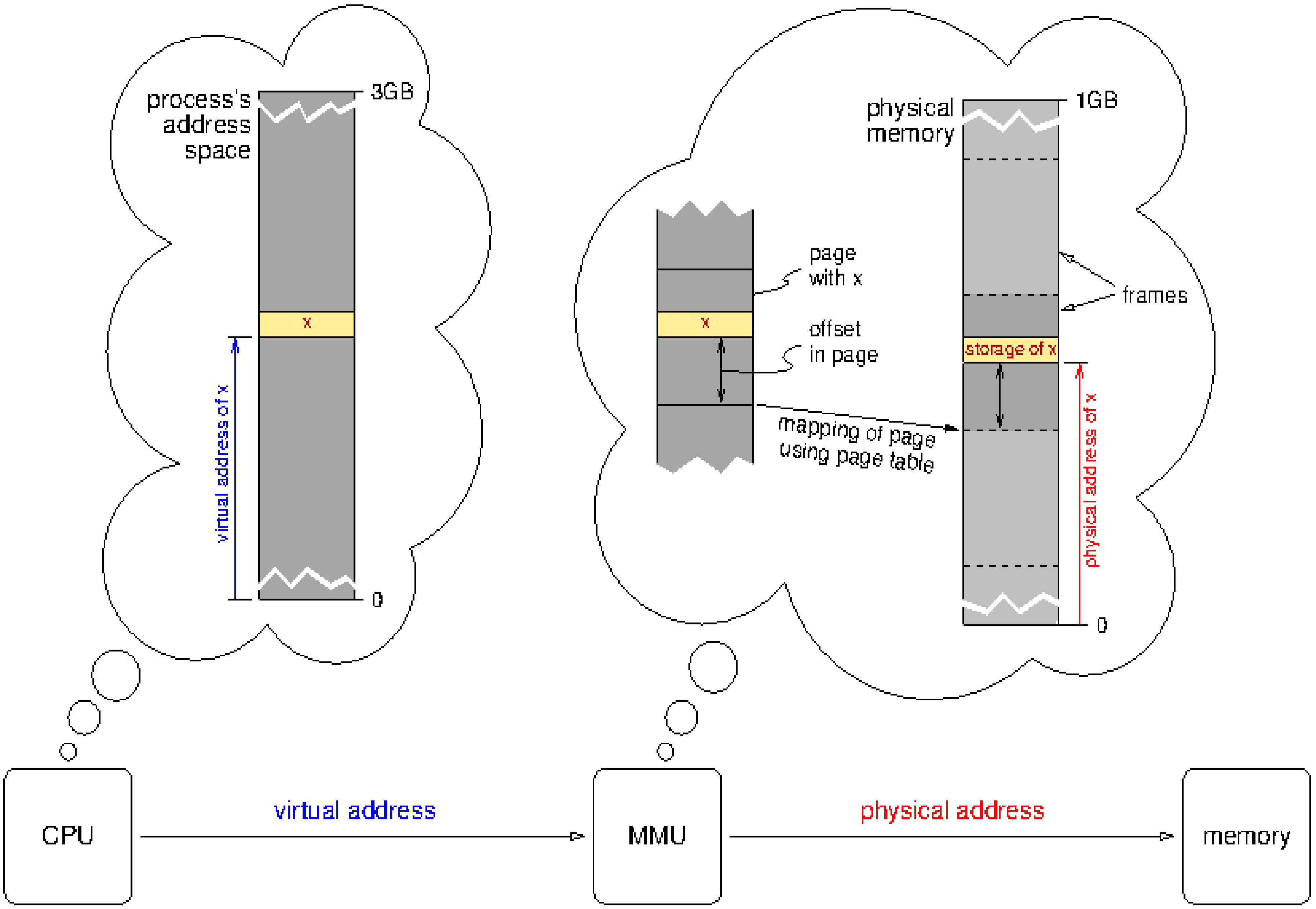


Frames Allocation Example



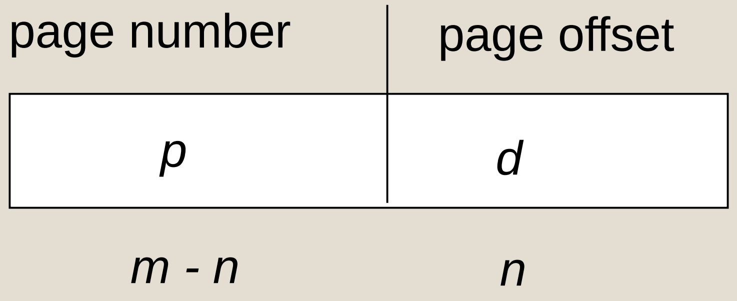
Before allocation

After allocation


process's address space
3GB
x
virtual address of x
0
page with x
x
offset in page
mapping of page using page table
physical memory
1GB
frames
storage of x
physical address of x
0
CPU
virtual address
MMU
physical address
memory

Address Translation Scheme

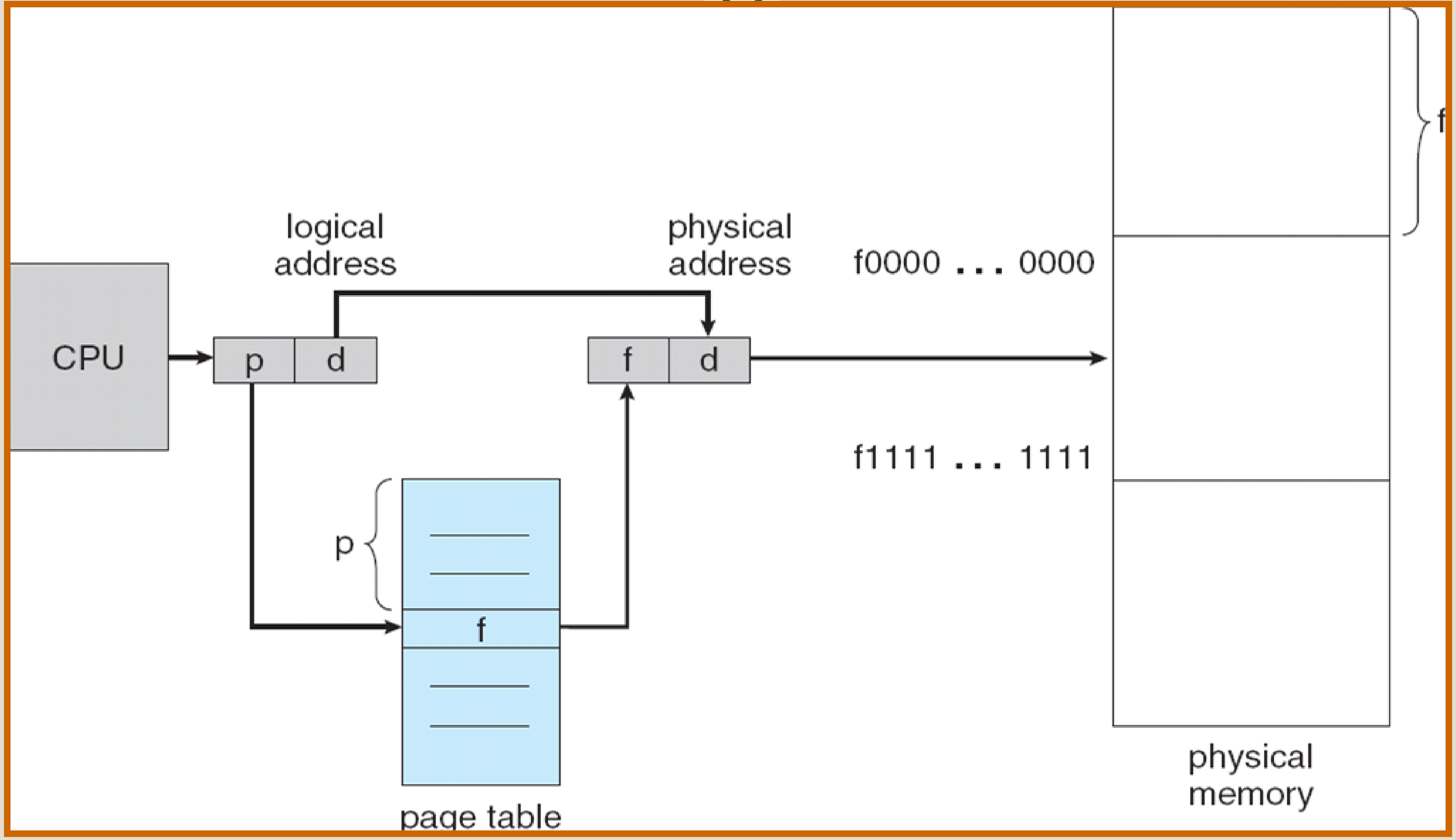
- Address generated by CPU is divided into:
- **Page number (p)** – used as an index into a ***page table*** which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit
- For a given logical address space of size 2^m words and **page size** 2^n words:



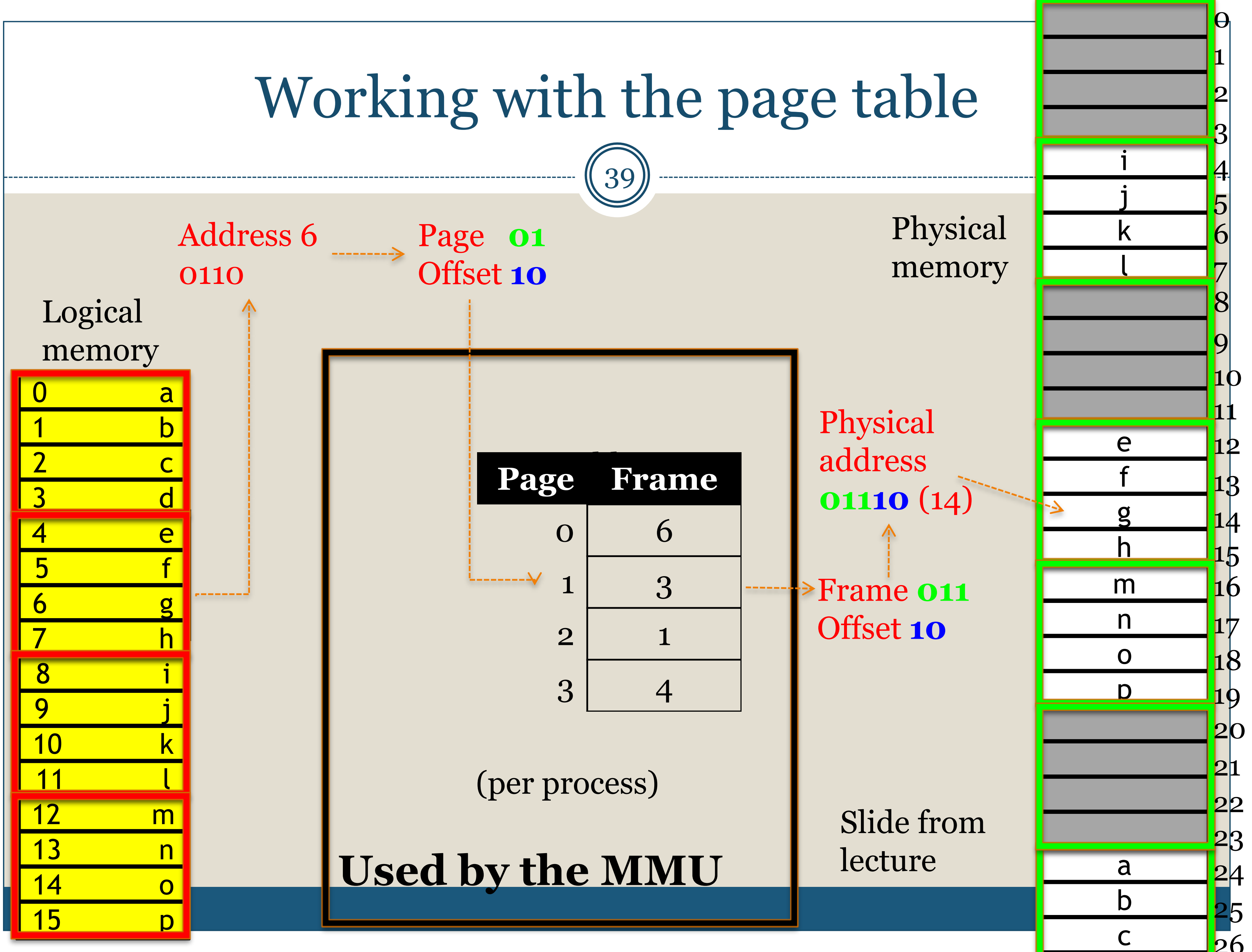
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PRLR**) indicates size of the page table

Paging Hardware



Working with the page table



Page	Frame
0	6
1	3
2	1
3	4

(per process)

Used by the MMU

Slide from lecture

Page Table Additional Info

- ***Valid bit*** - indicates whether the page is assigned to a frame.
- **Modified** - **Dirty bit** - indicates whether the page was modified.
- **Used bit** - when was the last accesses to page, i.e. timestamp.
  - E.g. for the clock algorithm for page replacement
- **Access permissions** - read-only, read-write

Page Replacement

- If the valid bit is off, the page is not mapped to a frame
- This caused an exception named page fault
- The OS searches for an available frame
  - If there is one – the page is loaded from the disk to this frame
  - Otherwise, the OS pages out (swaps out) a page to the disk, brings the requested frame to this location in the memory, and updates the page table accordingly.
- To choose the evicted page, the OS uses Page Replacement Algorithms, such as the clock algorithm (using used bit)

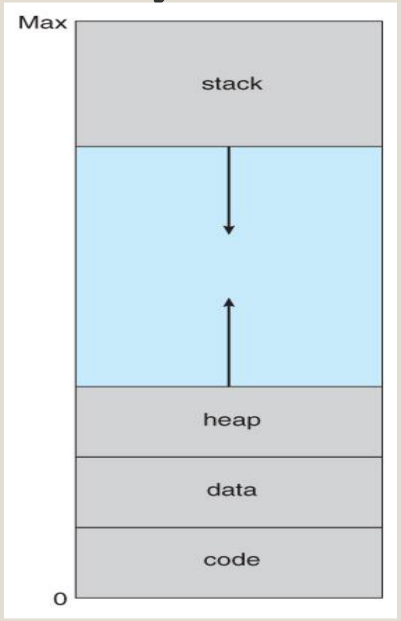
Structure of the Page Table

- As the number of processes increases, the percentage of memory devoted to page tables also increases.

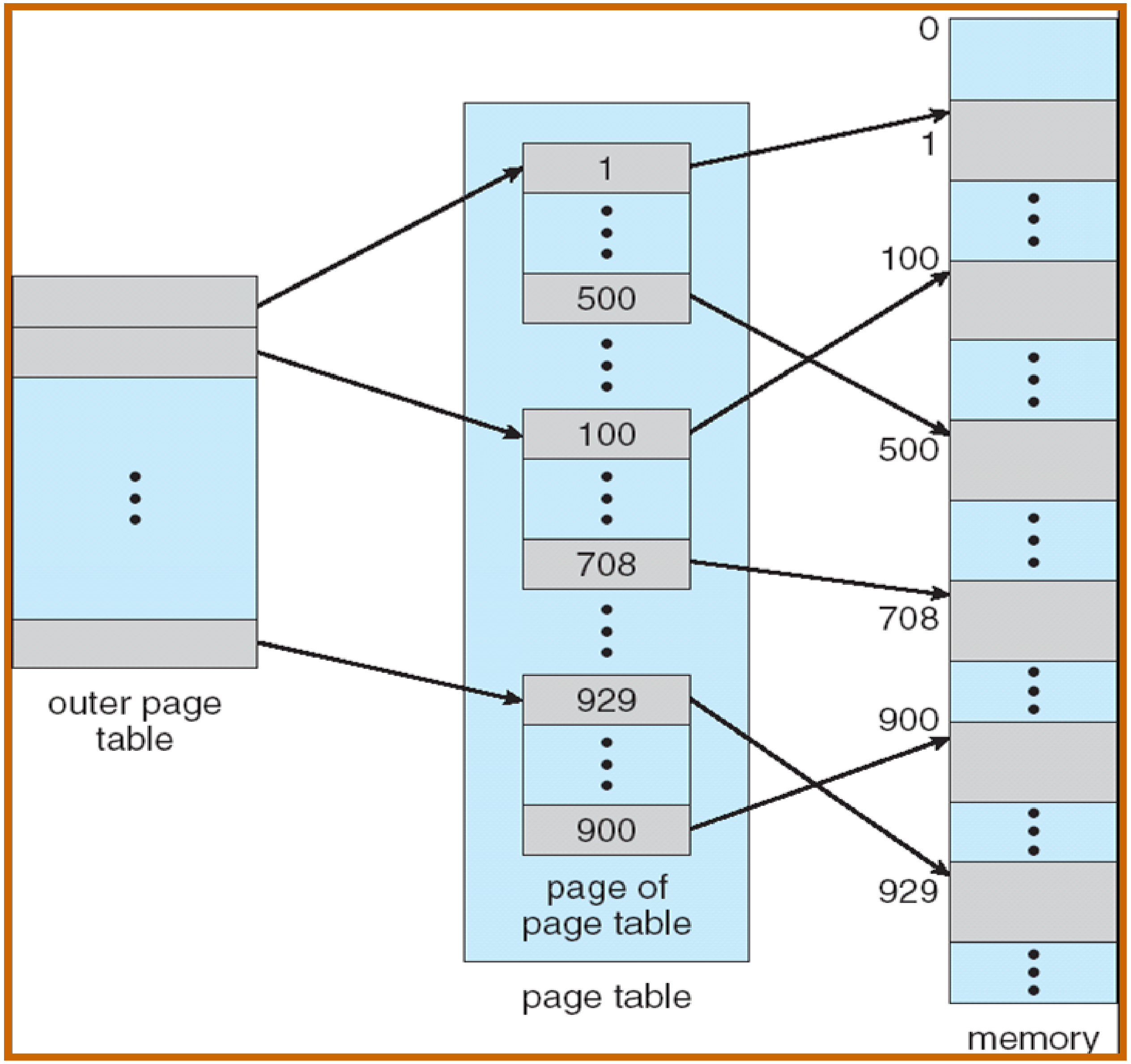
- The following structures solved this problem:
  - Hierarchical Paging
  - Inverted Page Tables

Hierarchical Page Tables

- Also called multilevel page table
- Break up the logical address space into multiple page tables
  - We don't need to preserve all the page tables, but only those that we use.
  - This works well because many times programs don't use the middle addresses ->
- A simple technique is a two-level page table (the page table is paged).

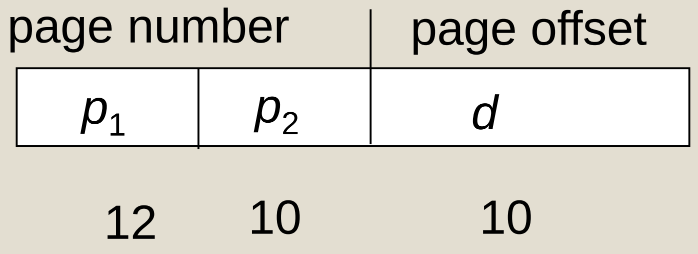


Two-Level Page-Table Scheme



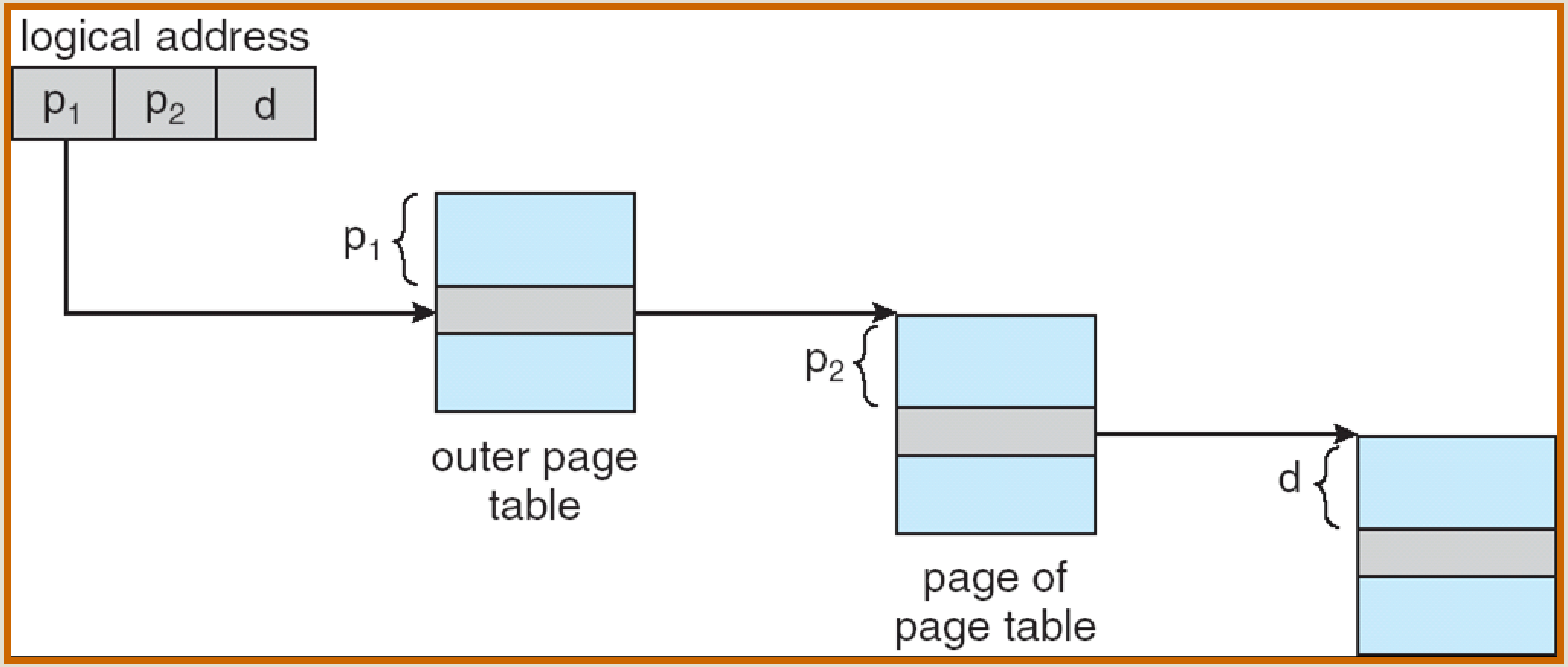
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page offset consisting of 10 bits.
  - a page number consisting of 22 bits (32-10).
- Since the page table is paged, the page number is divided into:
  - a 10-bit page offset.
  - a 12-bit page number (22-10).
- Thus, a logical address is as follows:



- where p1 is an index into the outer page table, and p2 is the displacement within the page of the outer page table.

Address-Translation Scheme



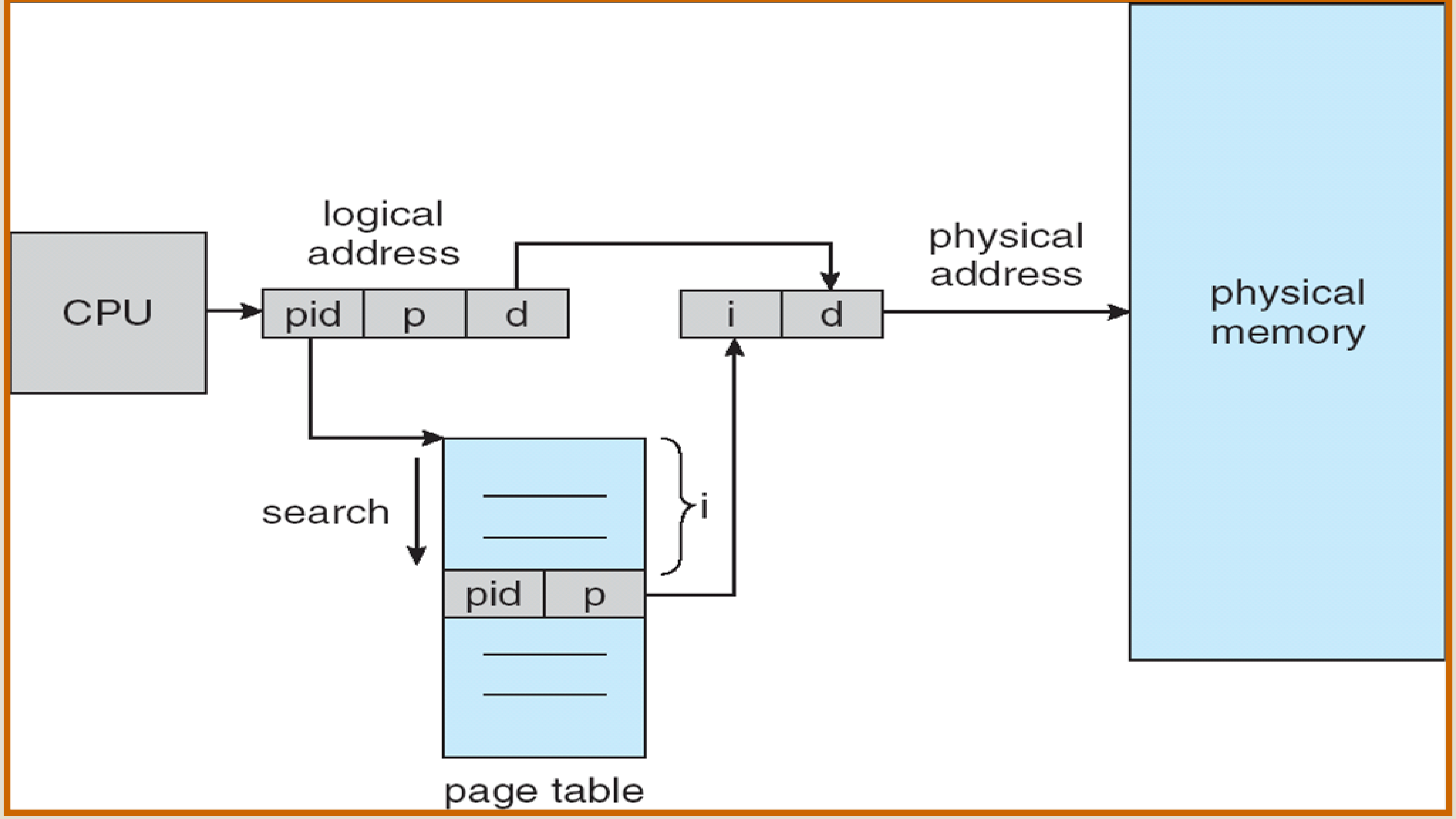
Inverted Page Table

- There is one table with the size of the physical memory <u>shared for all processes.</u>

- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

Inverted Page Table Architecture



TLB

- In 1-level paging, every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
  - Usually it's more than two accesses (multi-level paging).

- This problem can be solved by storing pages-to-frames maps in a (HW) cache

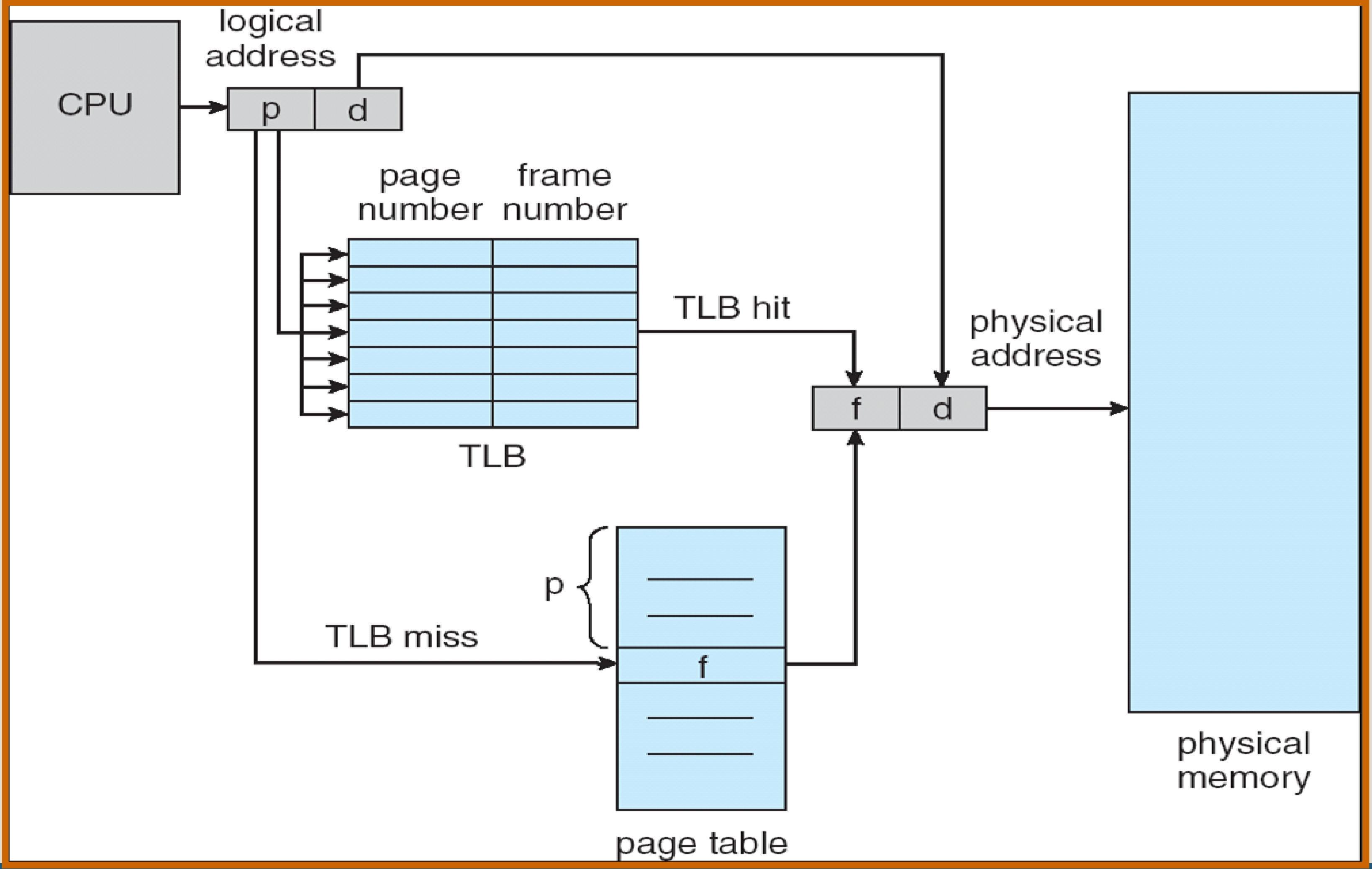
- **Translation look-aside buffers (TLBs) is a** special fast-lookup hardware cache of the page table

TLB

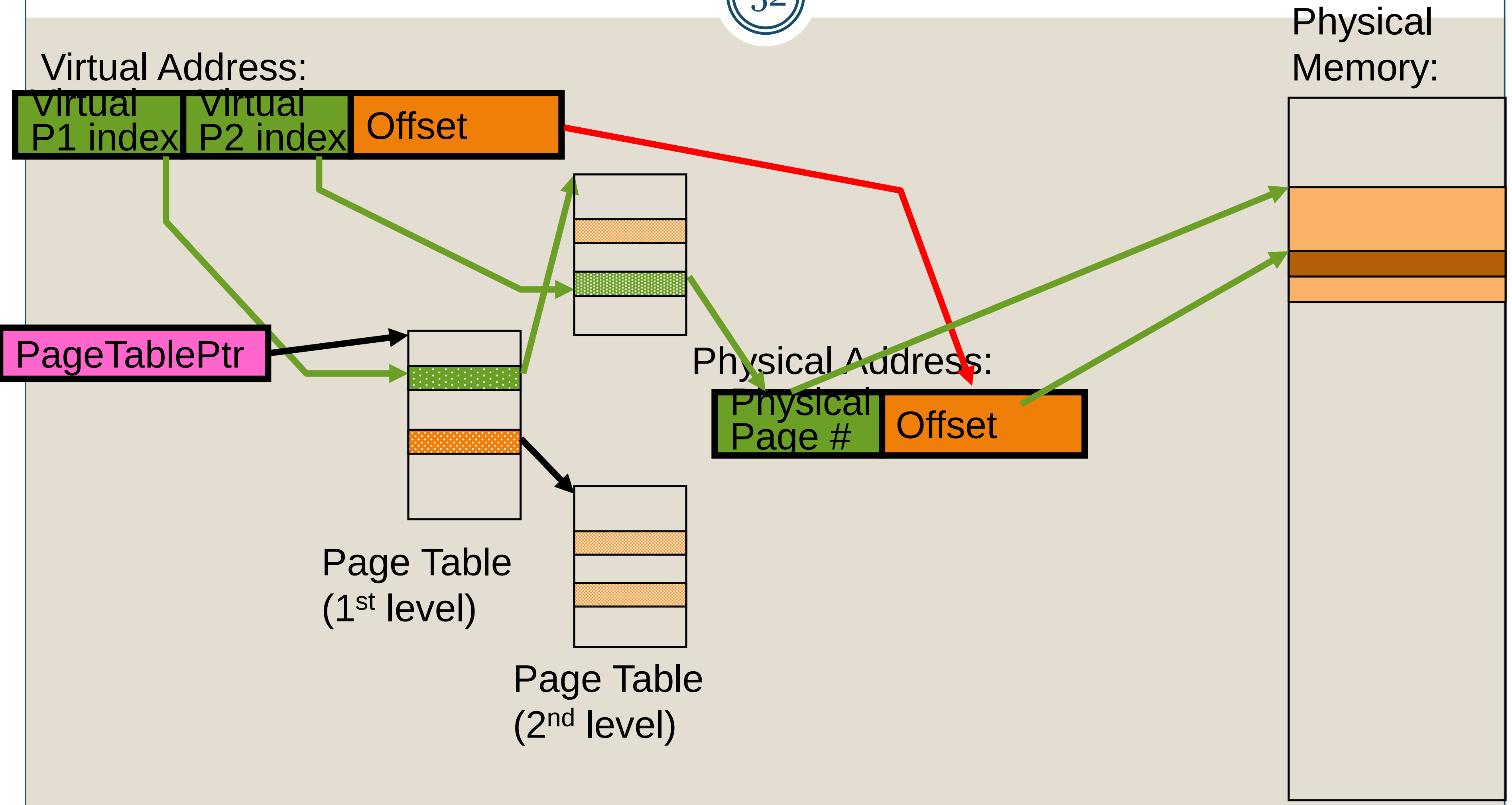
- Usually it's fully associative with 64 entries.
  - Fully associative ➔ Index size = 0 ➔ tag = page number (only tag and offset parts).

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process.

Paging Hardware With TLB

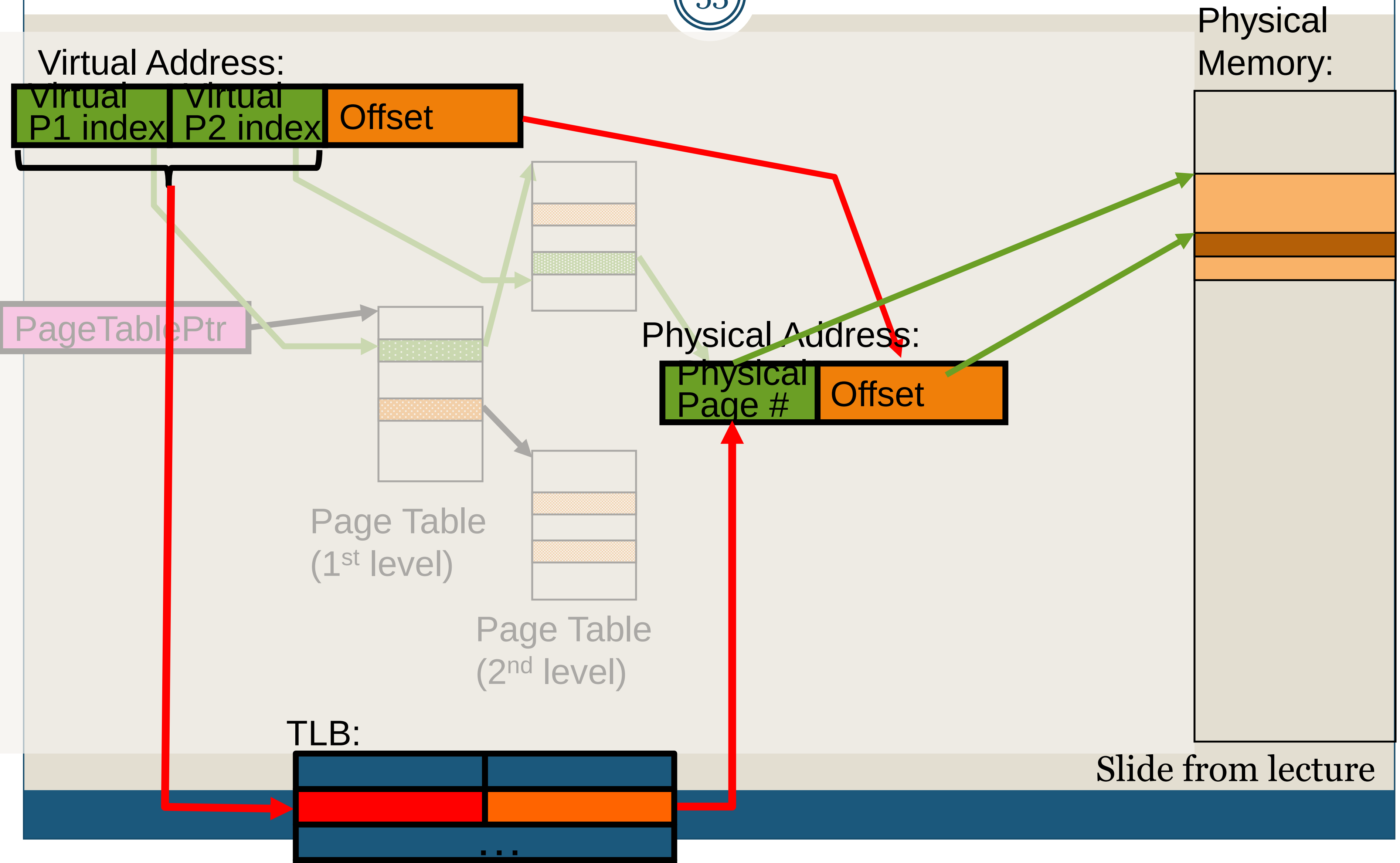


Putting Everything Together: Address Translation



Slide from lecture

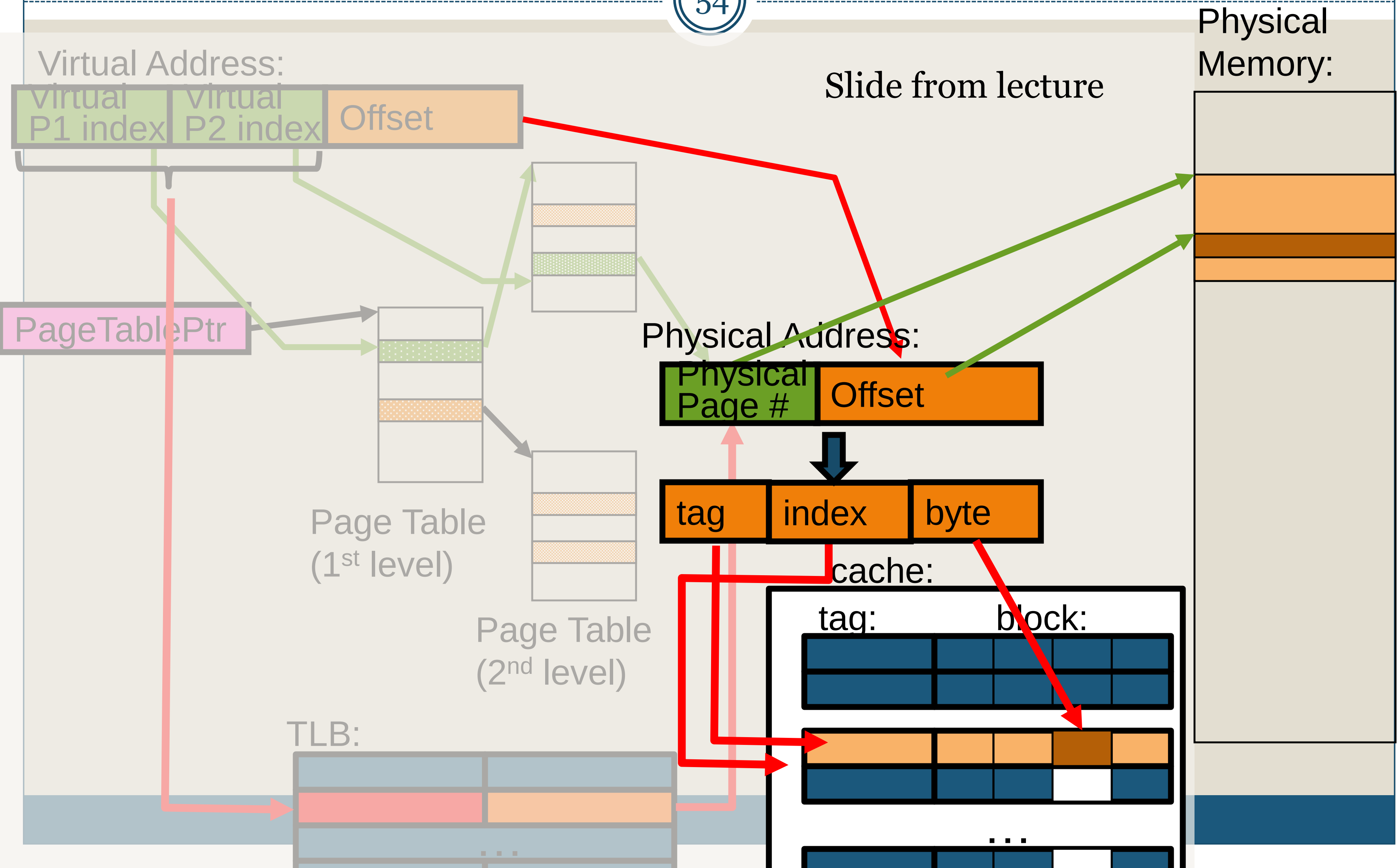
Putting Everything Together: TLB



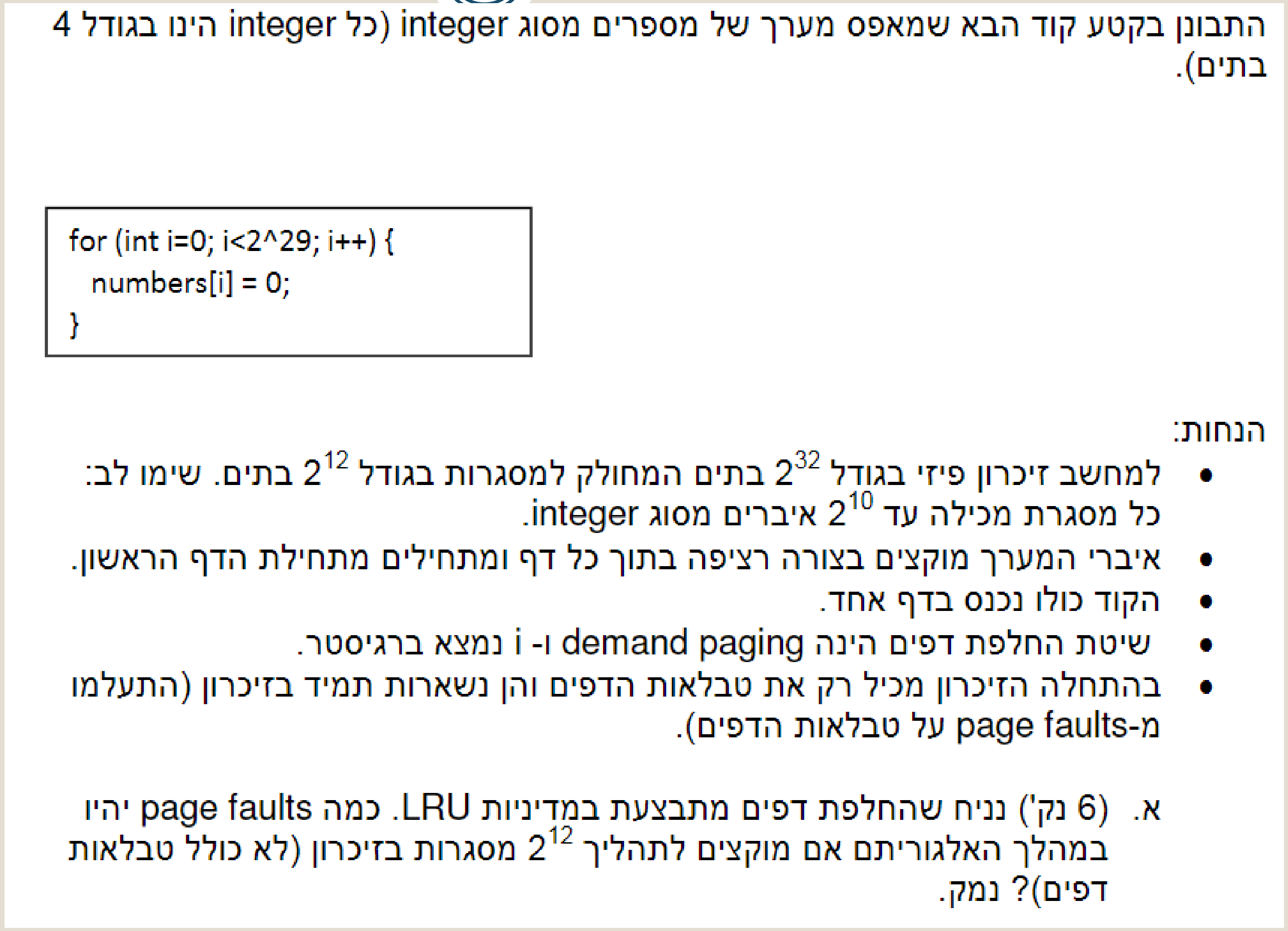
Slide from lecture

Putting Everything Together: Cache

Slide from lecture



Last Year Exam



התבונן בקטע קוד הבא שמאפס מערך של מספרים מסוג integer (כל integer הינו בגודל 4 בתים).

```
for (int i=0; i<2^29; i++) {
  numbers[i] = 0;
}
```

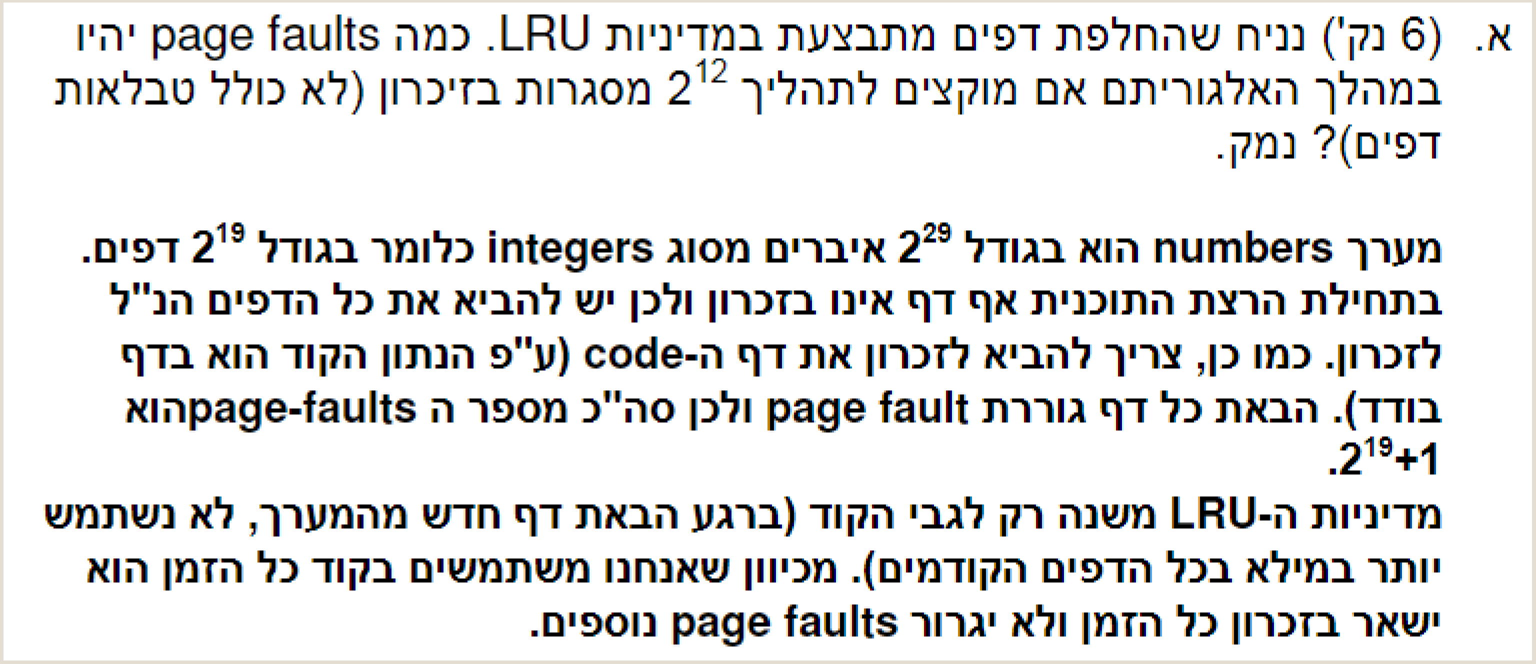
הנחות:

- למחשב זיכרון פיזי בגודל 2^{32} בתים המחולק למסגרות בגודל 2^{12} בתים. שימו לב: כל מסגרת מכילה עד 2^{10} איברים מסוג integer.
- איברי המערך מוקצים בצורה רציפה בתוך כל דף ומתחילים מתחילת הדף הראשון.
- הקוד כולו נכנס בדף אחד.
- שיטת החלפת דפים הינה demand paging ו- i נמצא ברגיסטר.
- בהתחלה הזיכרון מכיל רק את טבלאות הדפים והן נשארות תמיד בזיכרון (התעלמו מ-page faults על טבלאות הדפים).

א. (6 נק') נניח שהחלפת דפים מתבצעת במדיניות LRU. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהליך 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

Taken from: http://www4.huji.ac.il/exams/67808_2016_2_1_1.pdf

Last Year Exam

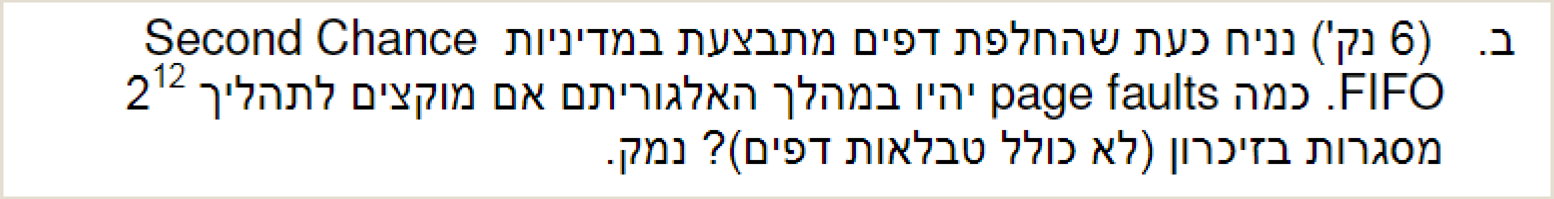


א. (6 נק') נניח שהחלפת דפים מתבצעת במדיניות LRU. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהליך 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

מערך numbers הוא בגודל 2^{29} איברים מסוג integers כלומר בגודל 2^{19} דפים. בתחילת הרצת התוכנית אף דף אינו בזכרון ולכן יש להביא את כל הדפים הנ"ל לזכרון. כמו כן, צריך להביא לזכרון את דף ה-code (ע"פ הנתון הקוד הוא בדף בודד). הבאת כל דף גוררת page fault ולכן סה"כ מספר ה page-faultsהוא $2^{19}+1$.
מדיניות ה-LRU משנה רק לגבי הקוד (ברגע הבאת דף חדש מהמערך, לא נשתמש יותר במילא בכל הדפים הקודמים). מכיוון שאנחנו משתמשים בקוד כל הזמן הוא ישאר בזכרון כל הזמן ולא יגרור page faults נוספים.

Taken from: http://www4.huji.ac.il/exams/67808_2016_2_1_1.pdf

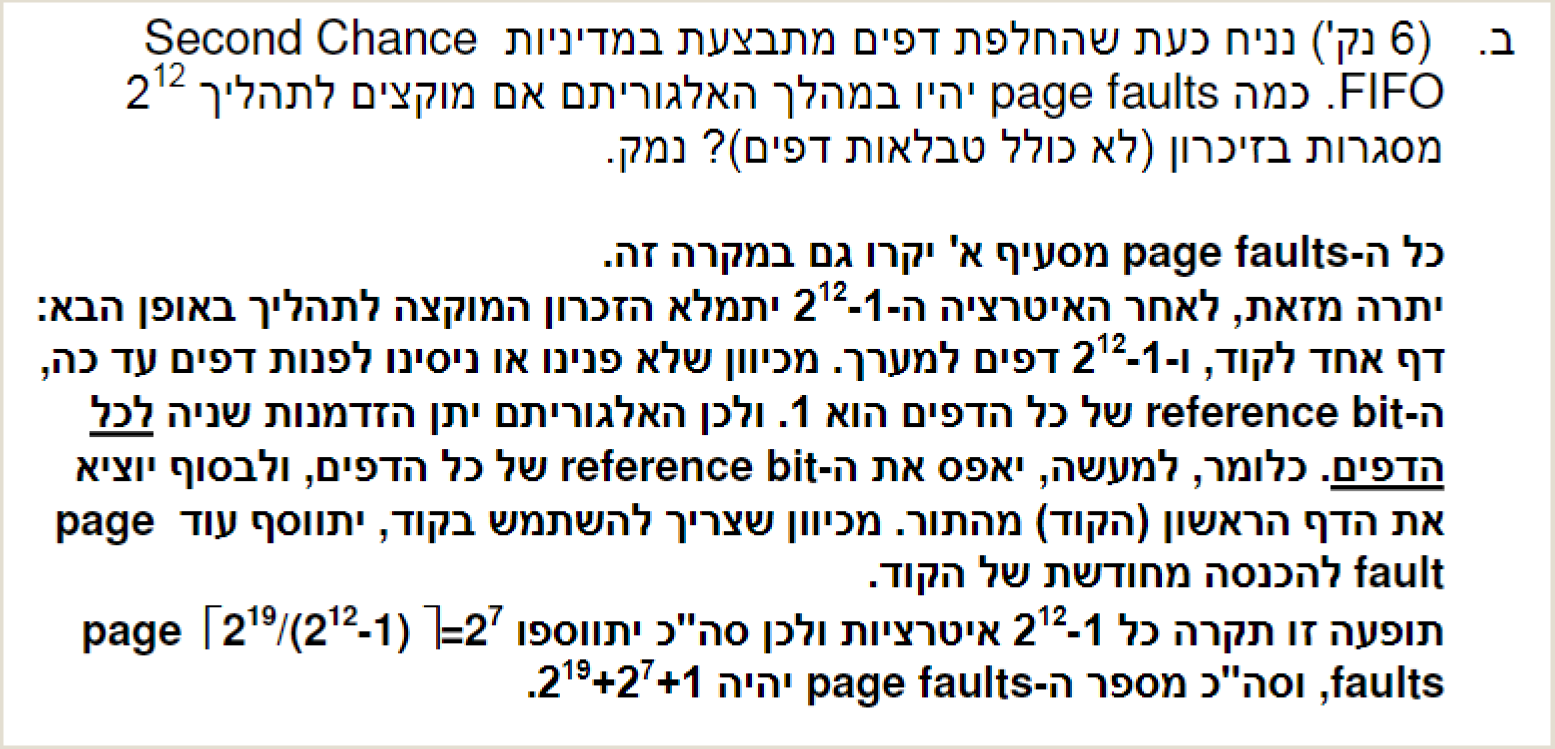
Last Year Exam



ב. (6 נק') נניח כעת שהחלפת דפים מתבצעת במדיניות Second Chance FIFO. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהליך 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

Taken from: http://www4.huji.ac.il/exams/67808_2016_2_1_1.pdf

Last Year Exam



ב. (6 נק') נניח כעת שהחלפת דפים מתבצעת במדיניות Second Chance FIFO. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהליך 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

כל ה-page faults מסעיף א' יקרו גם במקרה זה.
יתרה מזאת, לאחר האיטרציה ה-$2^{12}-1$ יתמלא הזכרון המוקצה לתהליך באופן הבא:
דף אחד לקוד, ו-$2^{12}-1$ דפים למערך. מכיוון שלא פנינו או ניסינו לפנות דפים עד כה,
ה-reference bit של כל הדפים הוא 1. ולכן האלגוריתם יתן הזדמנות שניה לכל
הדפים. כלומר, למעשה, יאפס את ה-reference bit של כל הדפים, ולבסוף יוציא
את הדף הראשון (הקוד) מהתור. מכיוון שצריך להשתמש בקוד, יתווסף עוד page
fault להכנסה מחודשת של הקוד.
תופעה זו תקרה כל $2^{12}-1$ איטרציות ולכן סה"כ יתווספו $\lceil 2^{19}/(2^{12}-1) \rceil=2^7$ page
faults, וסה"כ מספר ה-page faults יהיה $2^{19}+2^7+1$.

Taken from: http://www4.huji.ac.il/exams/67808_2016_2_1_1.pdf