Operating Systems:

Chap3

Page Tables

CS400

Week 6: 20th Feb

Spring 2020

Oliver BONHAM-CARTER

Paging Systems

- A paging system is a table to keep track of where blocks of programs are stored in memory.
  - Page (program in memory) ↔ Frame (physical memory)
- Page – The program in memory
  - Loaded in memory as a continuous stream of bits
  - Logical address space
- Frame – Memory space where program will be loaded
  - physical address space

Paging

Pages: Logical addresses

addresses **Frames**: Physical addresses

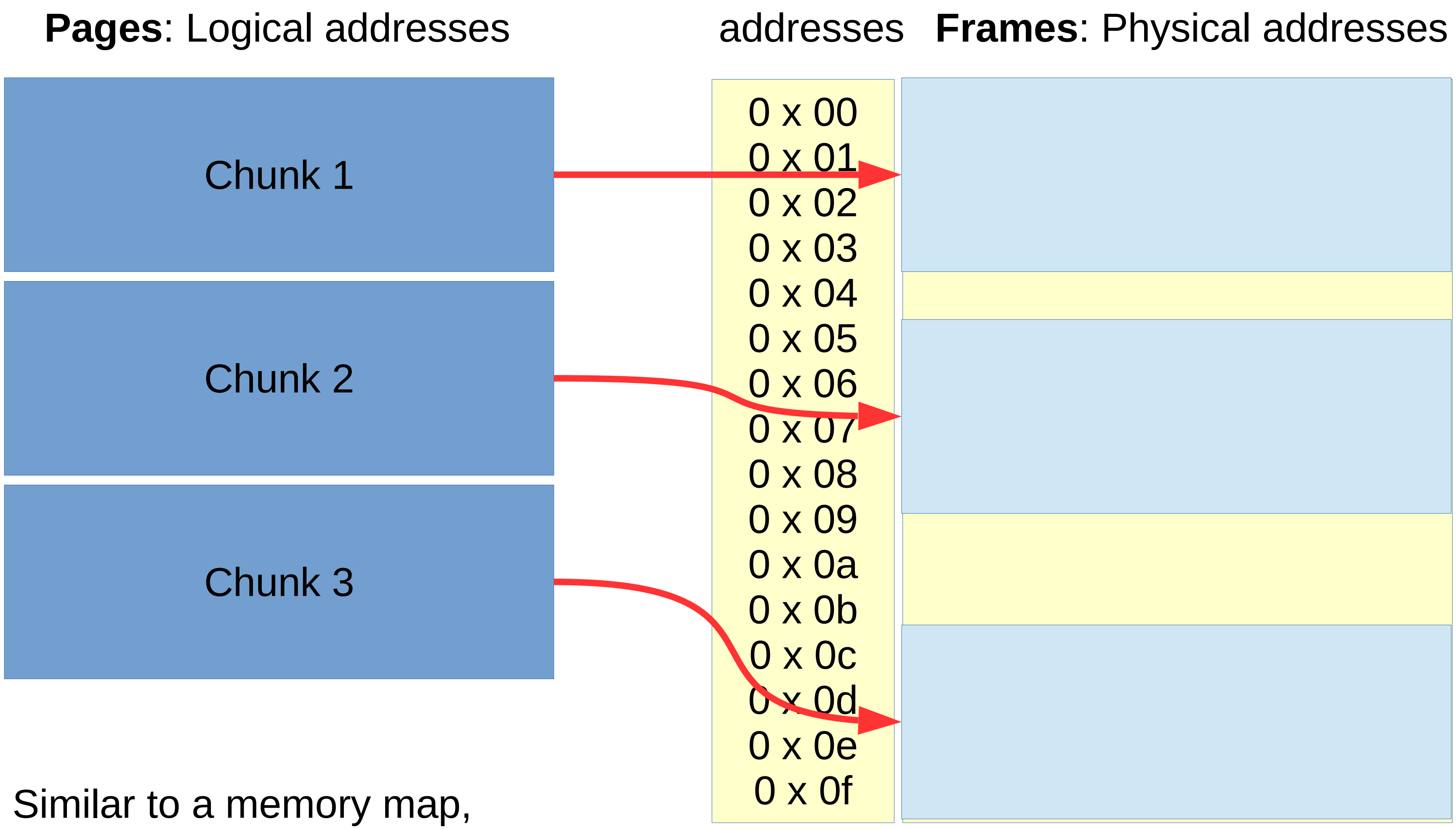
0 x 00
0 x 01
0 x 02
0 x 03
0 x 04
0 x 05
0 x 06
0 x 07
0 x 08
0 x 09
0 x 0a
0 x 0b
0 x 0c
0 x 0d
0 x 0e
0 x 0f

Similar to a memory map,
the paging system allows physical
addresses of a process to be non-continuous

Loading programs into memory

- The entire program will be loaded into memory in a continuous fashion.
- Logical addresses – addresses used in programs
- Physical addresses – addresses used in RAM memory
- How do we convert logical addresses to physical addresses?
  - Similar to a mapping system that we have studied earlier

Paging:
Placing the logical into the physical



Similar to a memory map,
the paging system allows physical
addresses of a process to be non-continuous

Paging:
Placing the logical into the physical

Pages: Logical addresses

0 : a
1 : h
2 : k
3 : n
4 : l
5 : e
6 : d
7 : j
8 : j
9 : v
10 : x
11 : s
12 : u
13 : y
14 : o
15 : p

Program

Frames: Physical addresses

0 x 00		0 x 10	
0 x 01		0 x 11	
0 x 02		0 x 12	
0 x 03		0 x 13	
0 x 04		0 x 14	
0 x 05		0 x 15	
0 x 06		0 x 16	
0 x 07		0 x 17	
0 x 08		0 x 18	
0 x 09		0 x 19	
0 x 0a		0 x 1a	
0 x 0b		0 x 1b	
0 x 0c		0 x 1c	
0 x 0d		0 x 1d	
0 x 0e		0 x 1e	
0 x 0f		0 x 1f	

Memory space

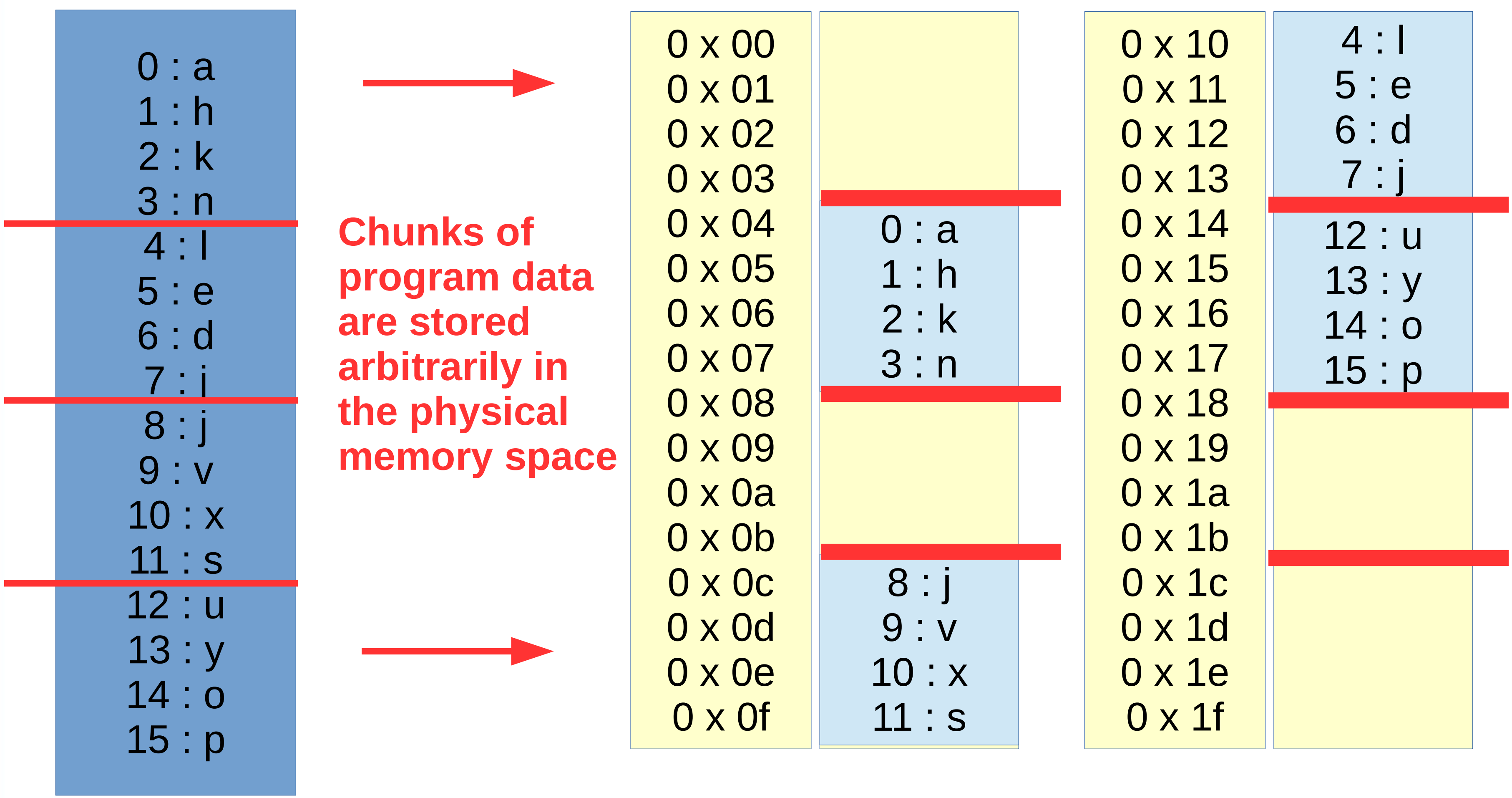
Loading Programs

- The program (left) is loaded into memory
- Pages can be stored in any frame
- We need some method to map the logical memory (the program’s instructions) to the frame (RAM)
- **How do we map which logical addresses to physical addresses?**

Paging:
Placing the Logical into the Physical

Pages: Logical addresses

Frames: Physical addresses

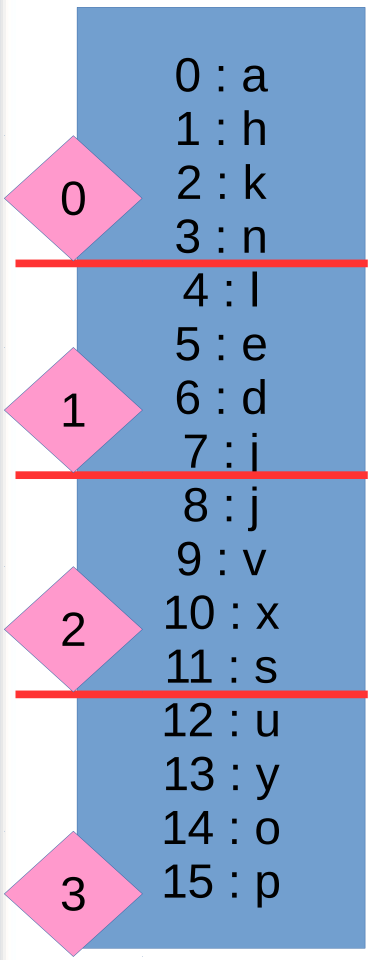


Program

Memory space

Paging:
Address Numbering System

Pages: Logical addresses

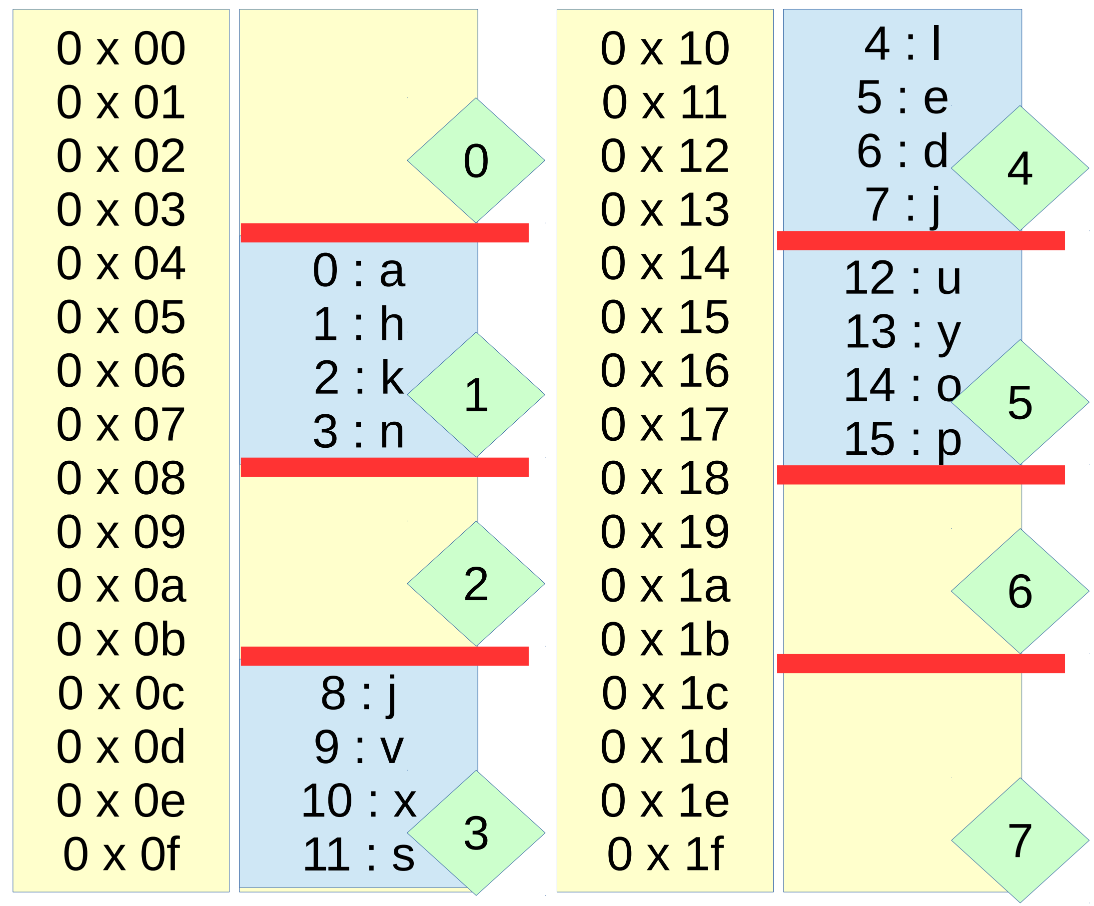


Program

Cut up regions Of the memory spaces: pages and frames.

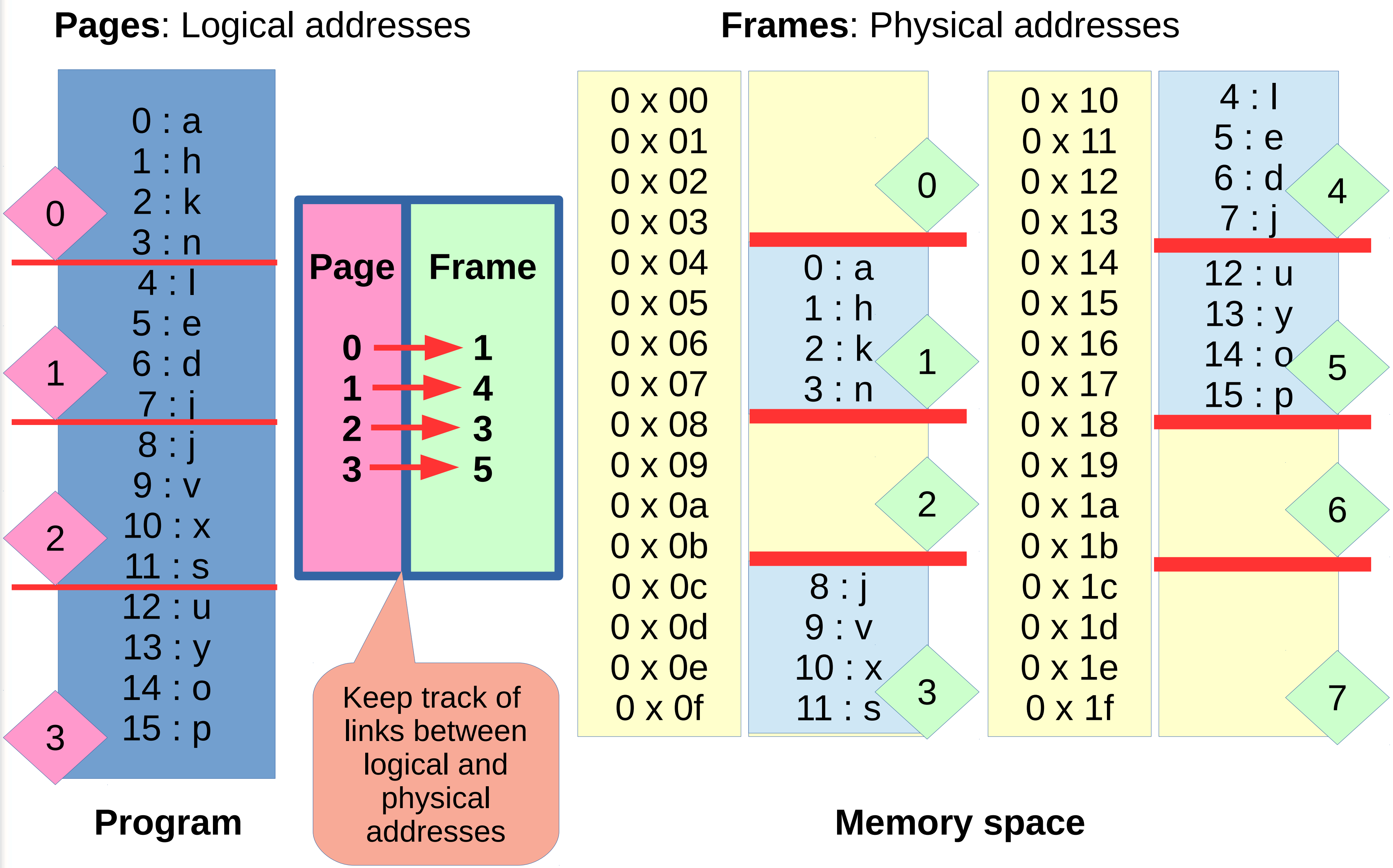
Use a table to connect them

Frames: Physical addresses



Memory space

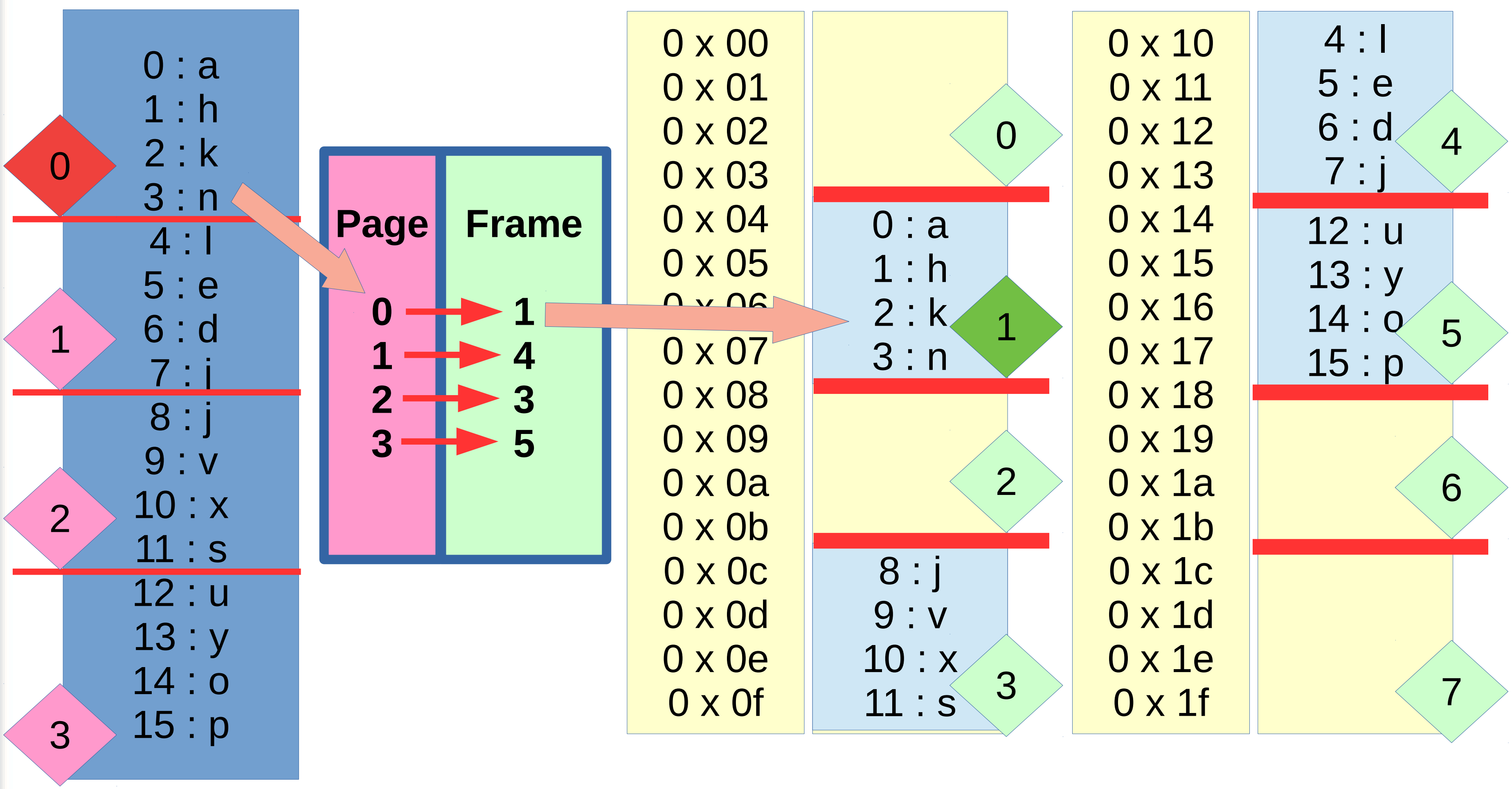
Paging:
Build a *Page Table*



Paging:
Build a *Page Table*

Pages: Logical addresses

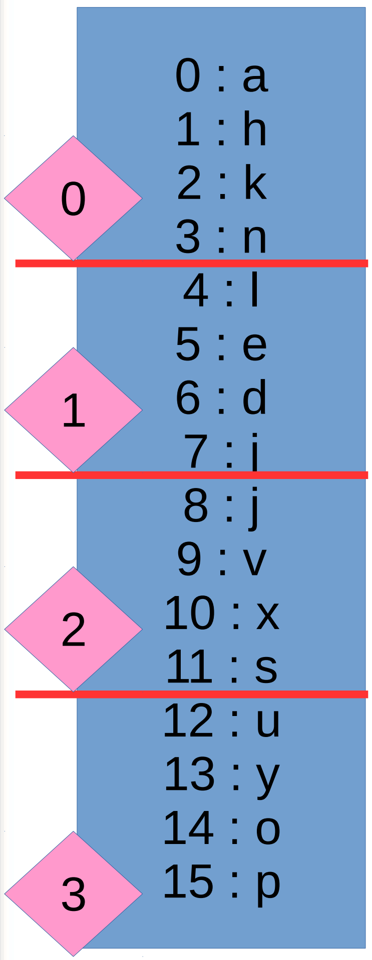
Frames: Physical addresses



Program

Memory space

Calculating Logical Addresses



- How do we represent the physical memory in binary addresses??
- There are 4 pages
- Each page contains 4 bytes of data:
  - Size of each page: 4 bytes
  - We need four different address to store 4 bytes

Back to the Paging Table

Page	Entries	Binary
0	0 : a 1 : h 2 : k 3 : n	00
1	4 : l 5 : e 6 : d 7 : i	01
2	8 : j 9 : v 10 : x 11 : s	10
3	12 : u 13 : y 14 : o 15 : p	11

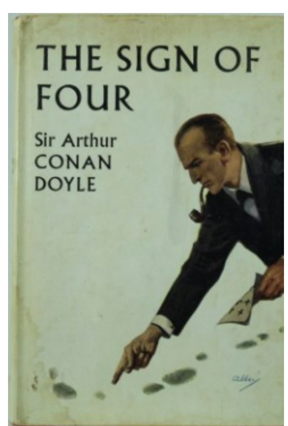
- Number of pages: 4
- Each page may be represented in binary:
  - **(0):** 00
  - **(1):** 01
  - **(2):** 10
  - **(3):** 11

The Sign of Four, *in Binary*

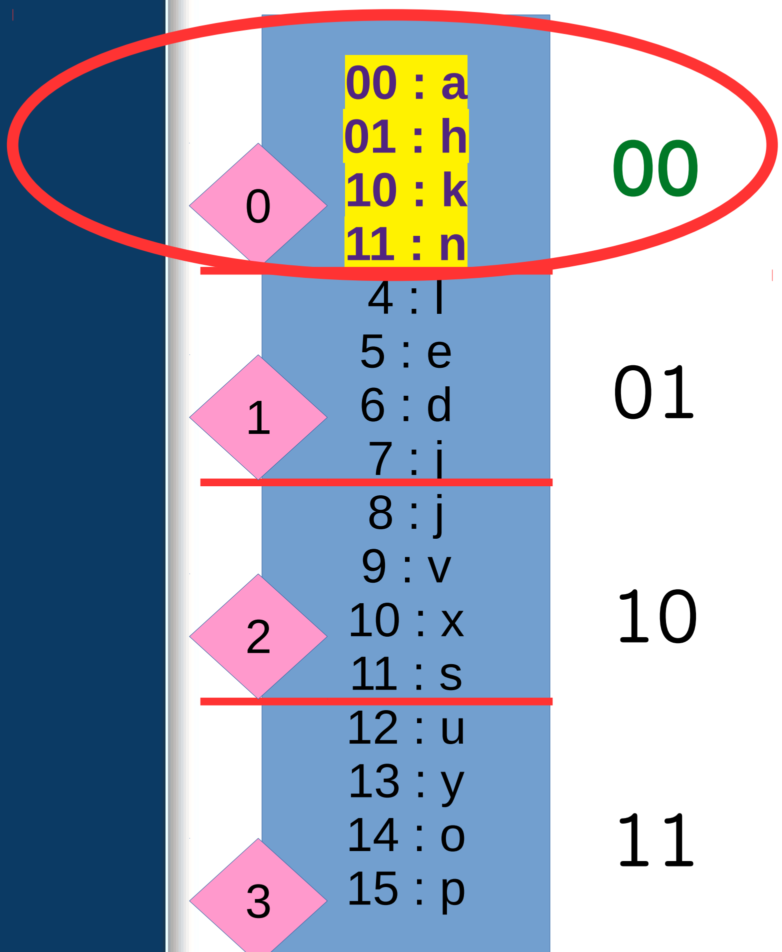
- Represent the element's order in binary
  - 0th a : bin(0) = **00**
  - 1st h : bin(1) = **01**
  - 2nd k : bin(2) = **10**
  - 3rd n : bin(3) = **11**

0 = 00 : a
1 = 01 : h
2 = 10 : k
3 = 11 : n

These values are called, "page offsets" and represent the locations in binary of chars.



Page Number and Offset



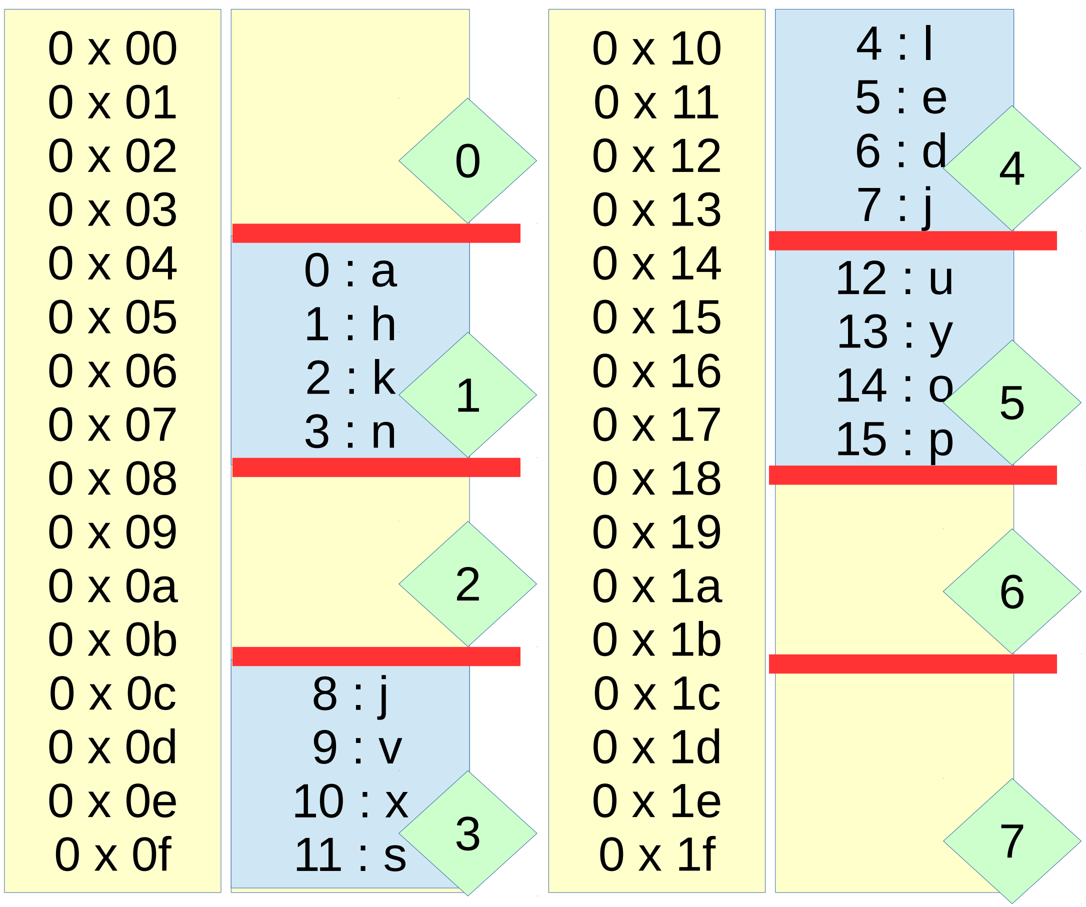
- With page-number code first (00), followed by offset (00)
- Page 1, 1st (**a**): 0000
- Page 1, 2nd (**h**): 0001
- Page 1, 3rd (**k**): 0010
- Page 1, 4th (**n**): 0011

Page-number, Offset

Frame Addresses

- There are eight frames.
- How many bits are necessary to represent eight frame addresses?

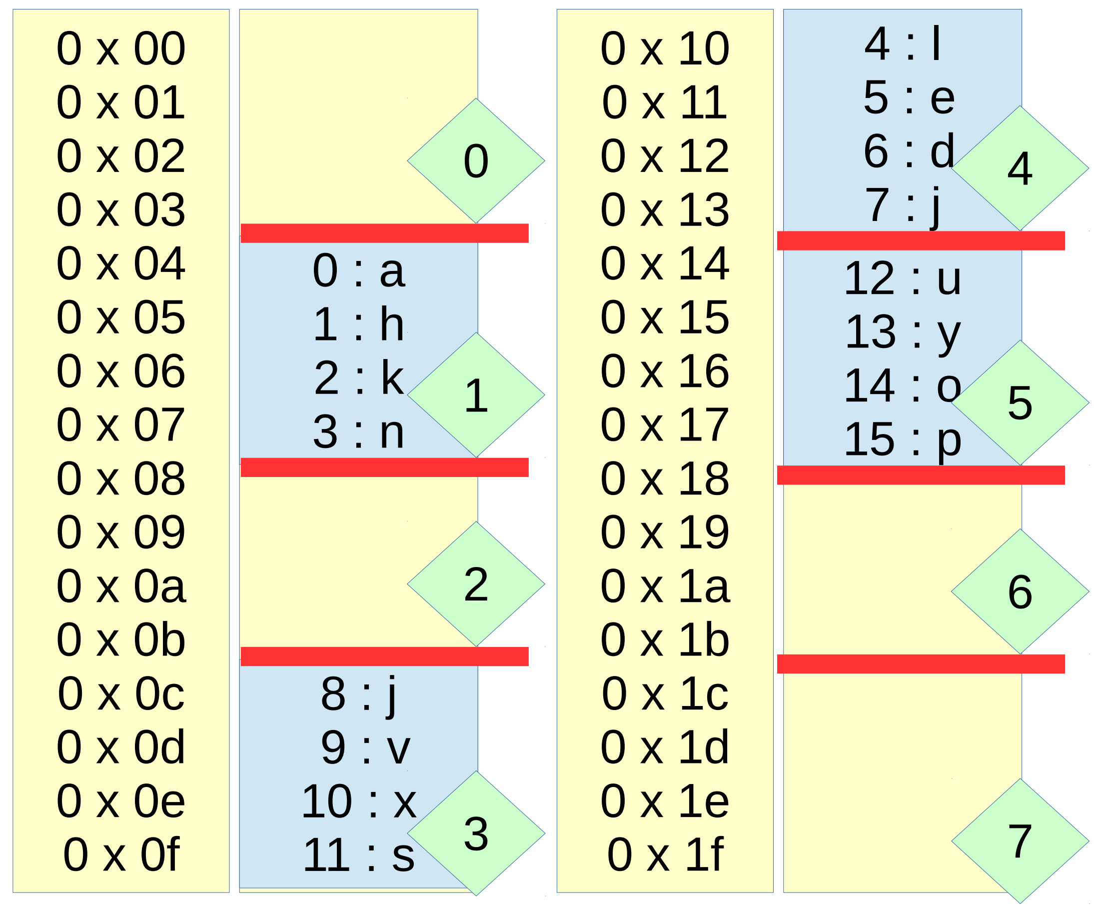
Frames: Physical addresses



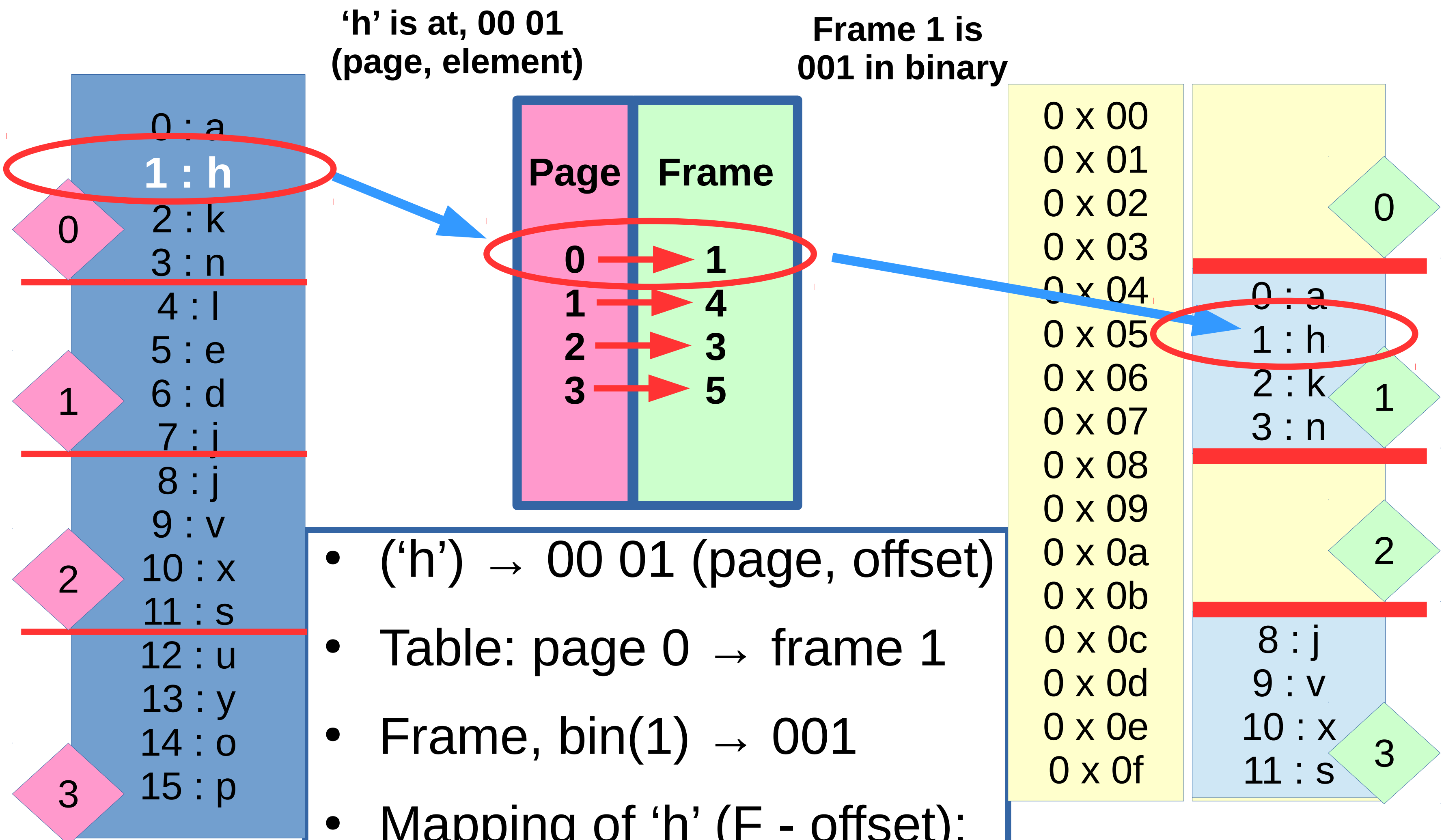
Eight Frames

- With two bits, the maximum number of frames is 4, so we have to use three bits (2 ^ 3 = 8) for all these frames
- Frame # → binary(#)
  - F 0 → bin(0) = 000
  - F 1 → bin(1) = 001
  - F 2 → bin(2) = 010
  - F 3 → bin(3) = 011
  - F 4 → bin(4) = 100
  - Etc...

Frames: Physical addresses



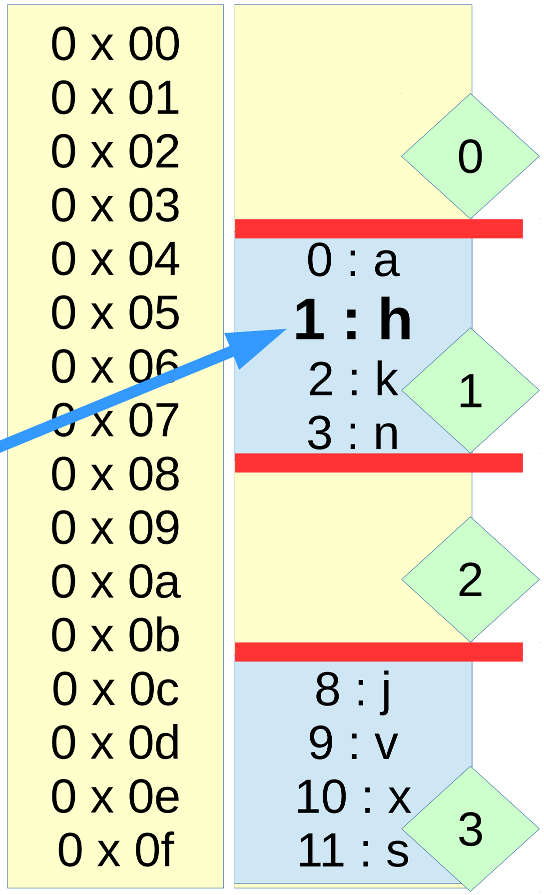
Frame and Page Addresses: ‘h’



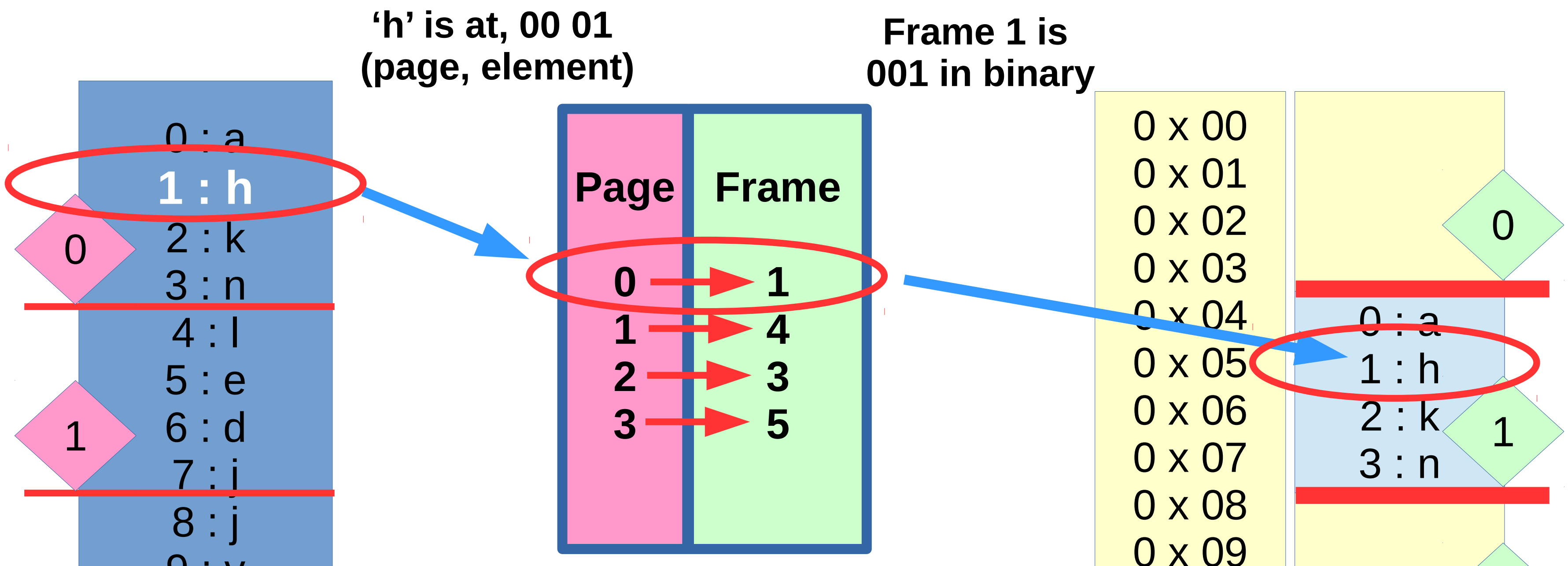
- (‘h’) → 00 01 (page, offset)
- Table: page 0 → frame 1
- Frame, bin(1) → 001
- Mapping of ‘h’ (F - offset): 001-01

Frame and Page Addresses: ‘h’

- We see, ‘h’ is mapped to physical memory at 001-01.
- Int(001-01) → 5
- Look at address 5



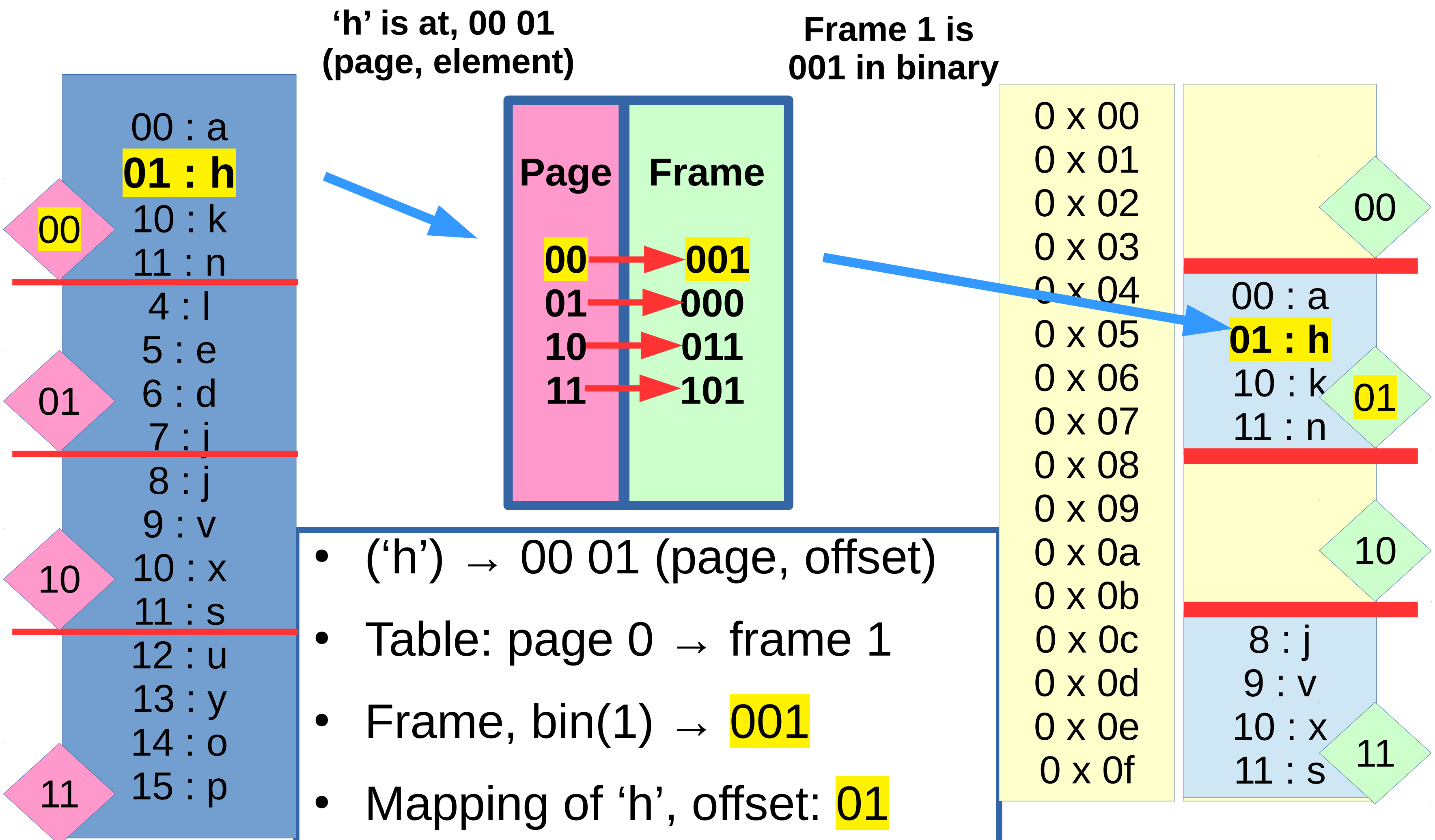
Again, The Pathway



- (‘h’) → 00 01 (page, offset)
- Table: page 0 → frame 1
- Frame, bin(1) → 001
- Mapping of ‘h’, offset: 01

  (Frame - offset): 001-01

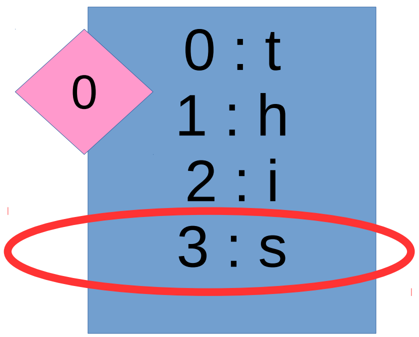
Now, All in Binary

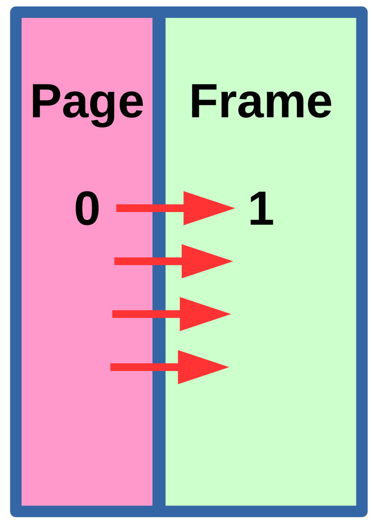


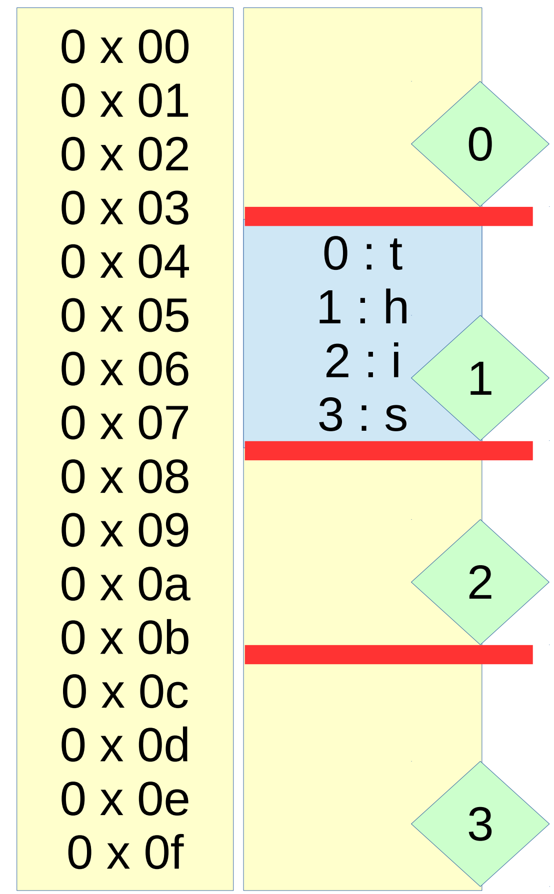
- ('h') → 00 01 (page, offset)
- Table: page 0 → frame 1
- Frame, bin(1) → 001
- Mapping of 'h', offset: 01

  (Frame - offset): 001-01

Consider This ...



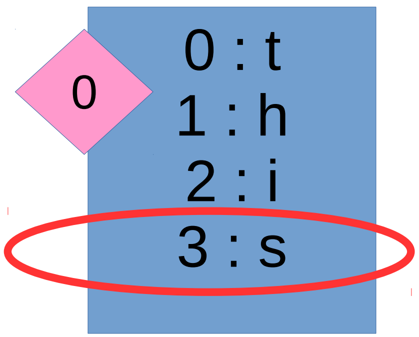


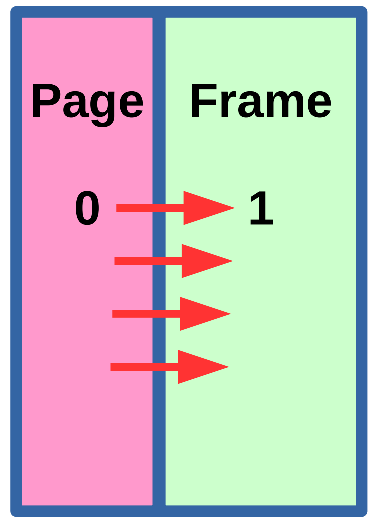


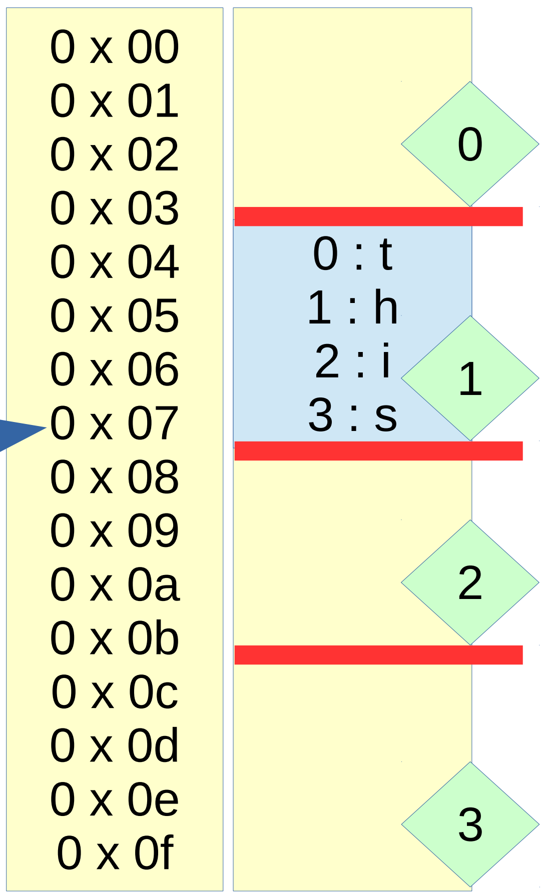
For element ‘s’, fill in the following in binary:

- **Page: ??**
- **Offset: ??**
- **Frame: ???**
- **Frame; offset: ???-??**
- **Int(Frame, offset) = ?**
- **Explain how you know.**

Consider This: Solution







For element 's', fill in the following in binary:

- **Page: 00 (in binary)**
- **Offset: 11**
- **Frame: 001**
- **Frame; offset: 001-11**
- **Int(001-11) = 7**

Consider this: Linked Lists

- Look at the code for this lesson (13): *linkedList_demo.c*
- *Compile: gcc -o linkedListDemo linkedList_demo.c*
- *Run the outputted file: linkedListDemo*
- *Why does storing values in linked lists make for more efficient memory usage?*