Detailed Paging Examples

James S. Plank
EECS Department
University of Tennessee

CS560: Operating Systems

Latest Revision: April, 2010

When starting with paging:

- The hardware defines the following:
  - Pointer size
  - Word size
  - Page size
  - Frames in memory
  - Paging scheme (Single level, etc).
- From that, you figure out how to lay out your address space.

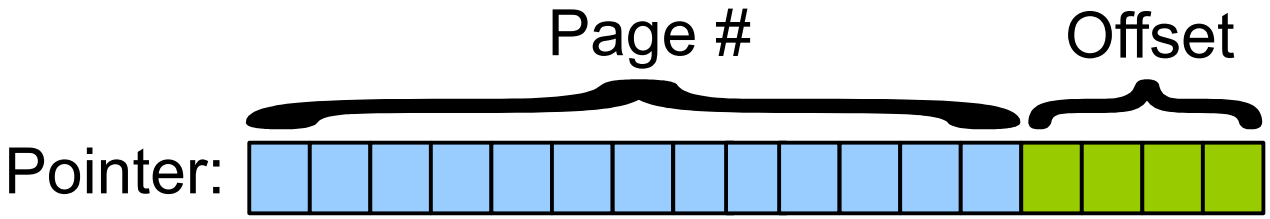
Example #1: Suppose:

Memory = 160 bytes.
Word size = 32 bits.
Pointers access bytes.
Pointers are 16 bits.
Page size = 4 words.

Single-level paging
With PTBR/PTLR.

Then:

Pages are 16 bytes.
Offset = 4 bits.
Page # = 12 bits.



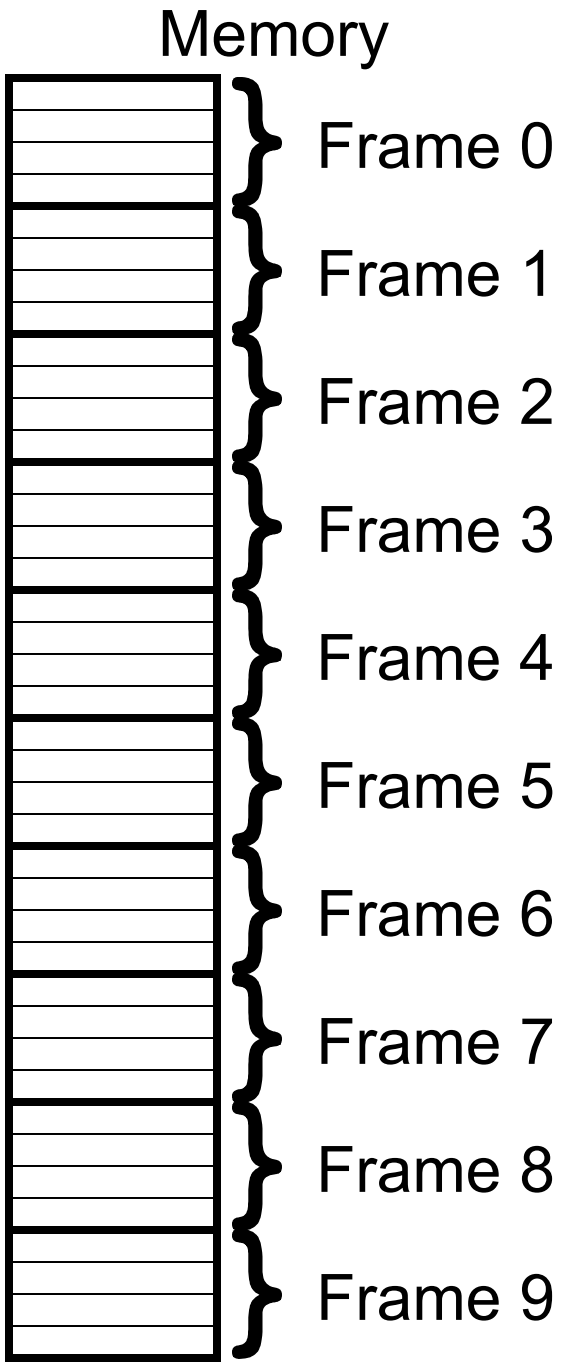
Processes can consume 4096 pages
(were there enough memory).

Example #1: Suppose:

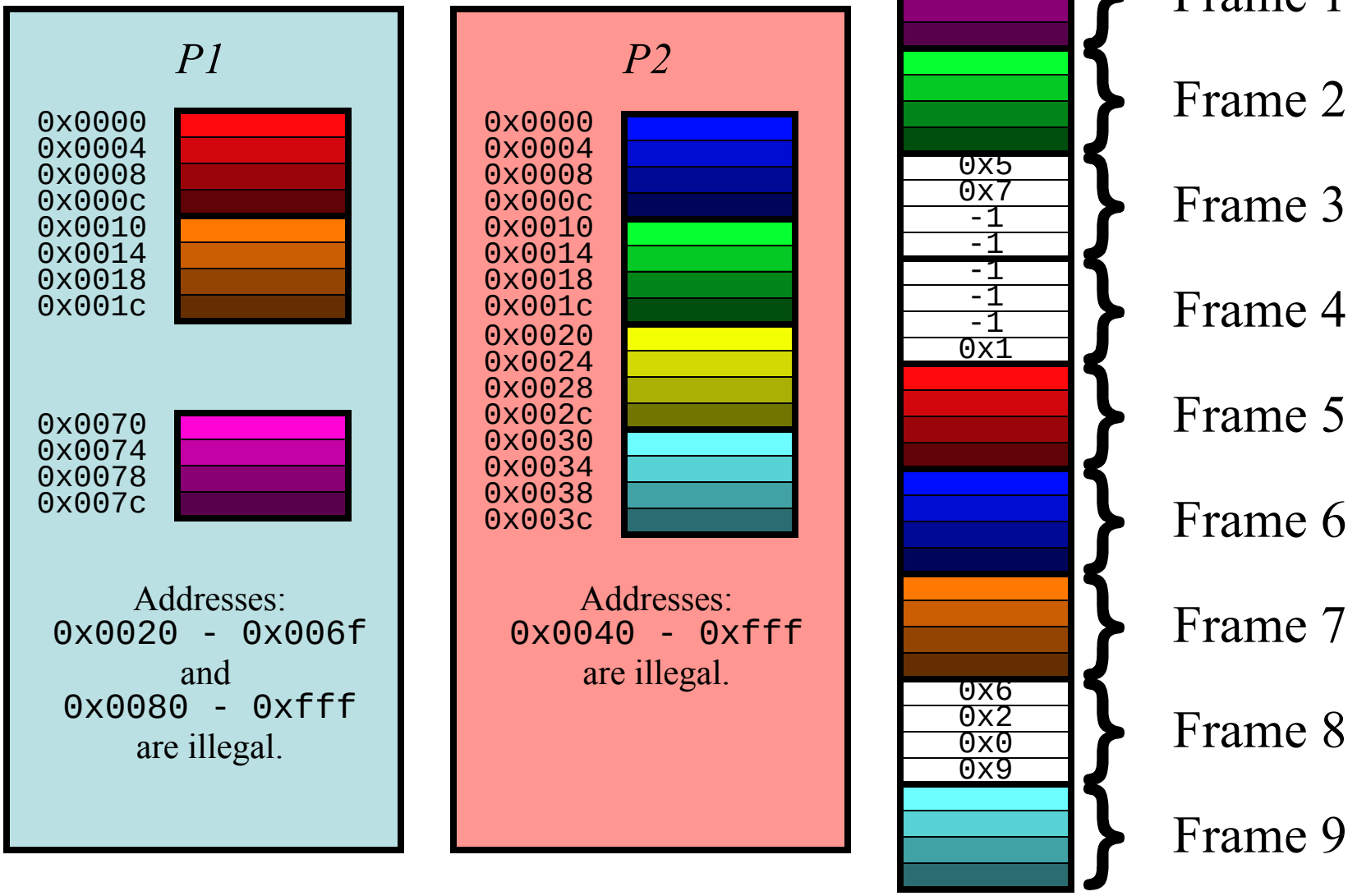
Memory = 160 bytes.
Word size = 32 bits.
Pointers access bytes.
Pointers are 16 bits.
Page size = 4 words.

Single-level paging
With PTBR/PTLR.

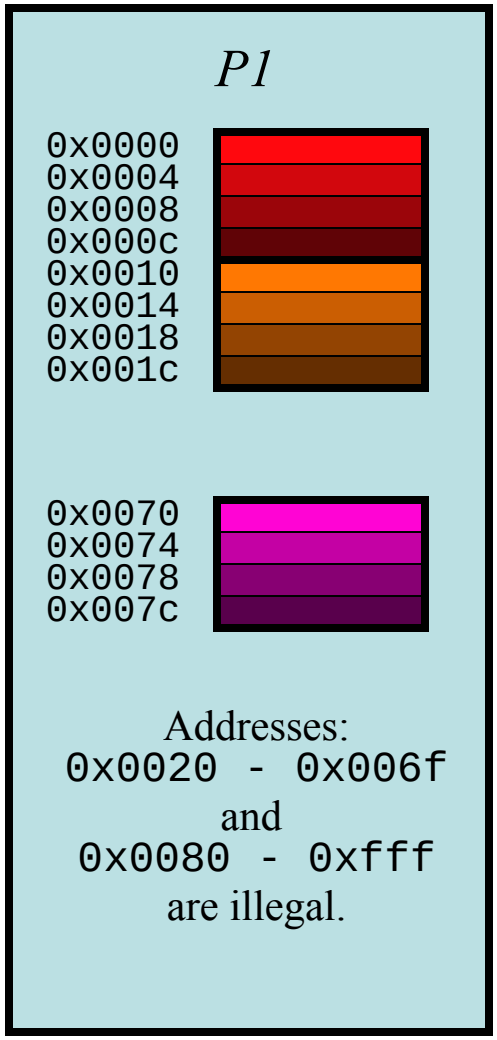
There are 10 frames in main memory.

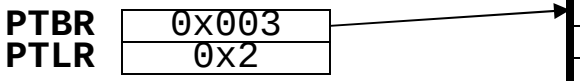


Here is a picture where there are two processes in memory:

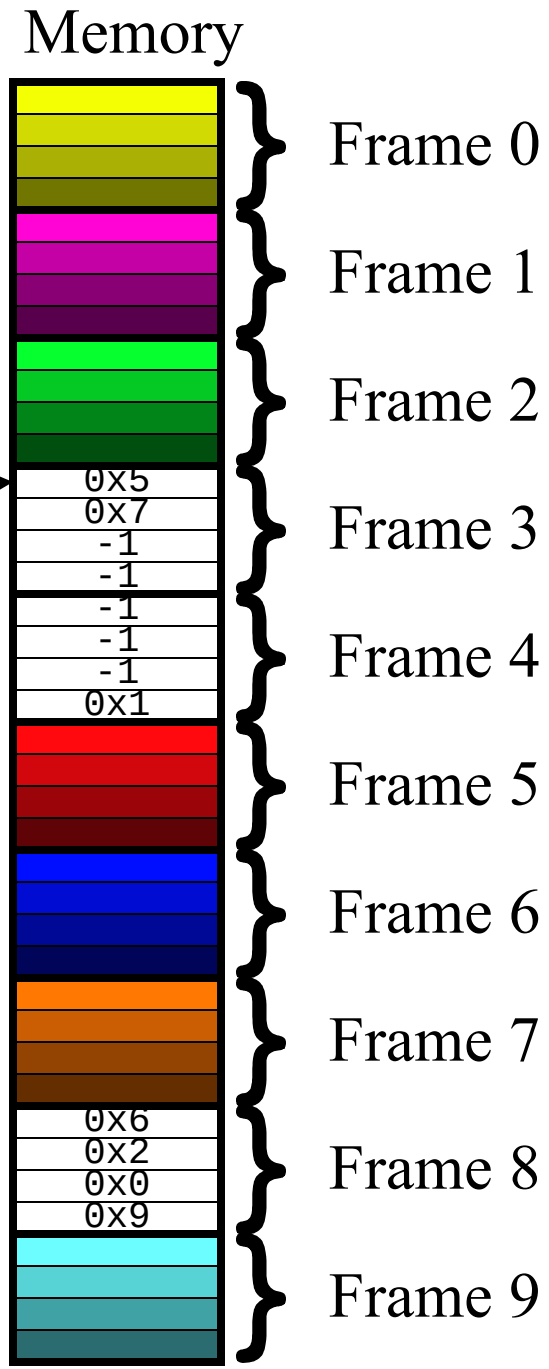


Suppose process P1 is running:

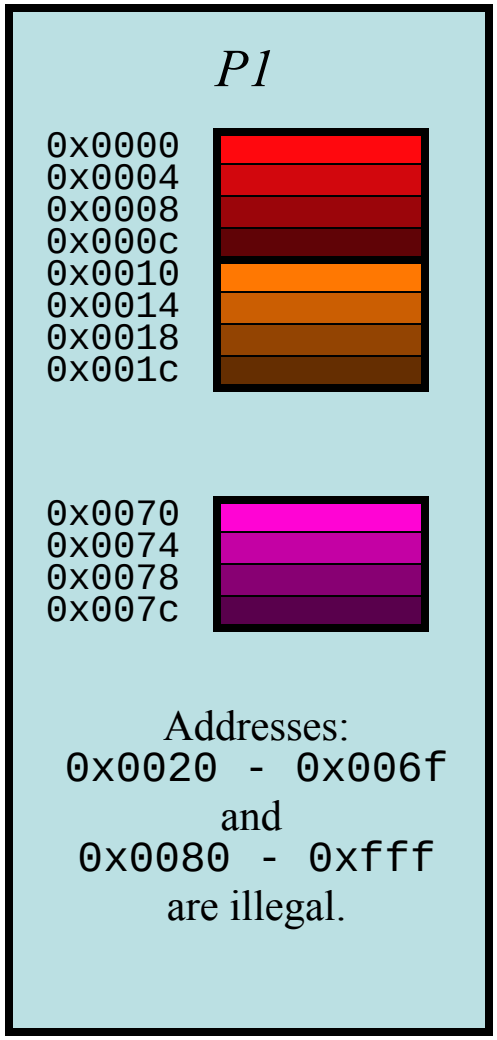


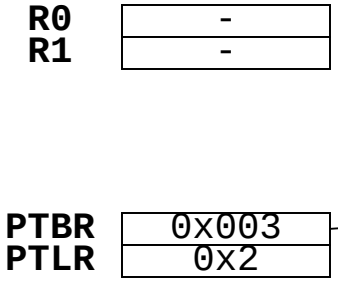


The PTBR and PTLR are set so that the hardware will find the right words.



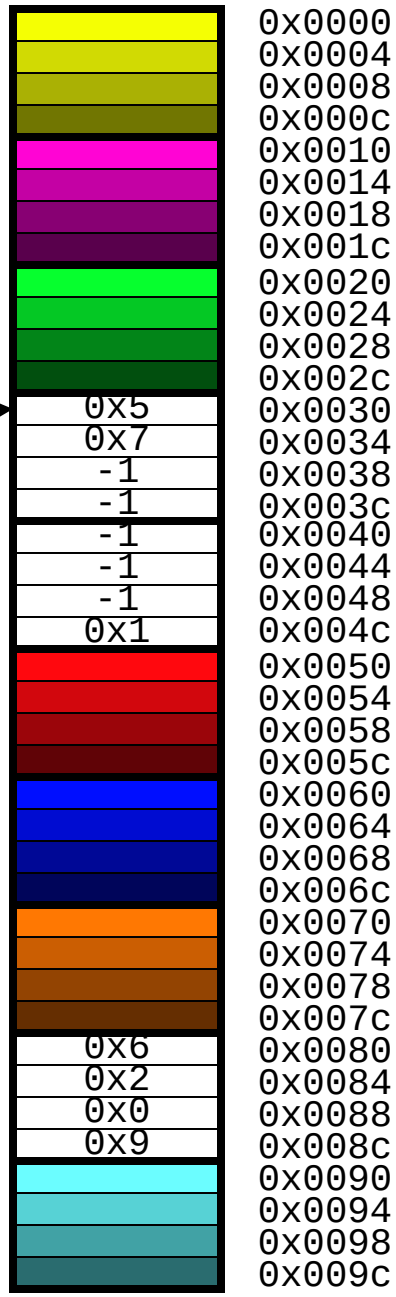
Suppose process P1 is running:



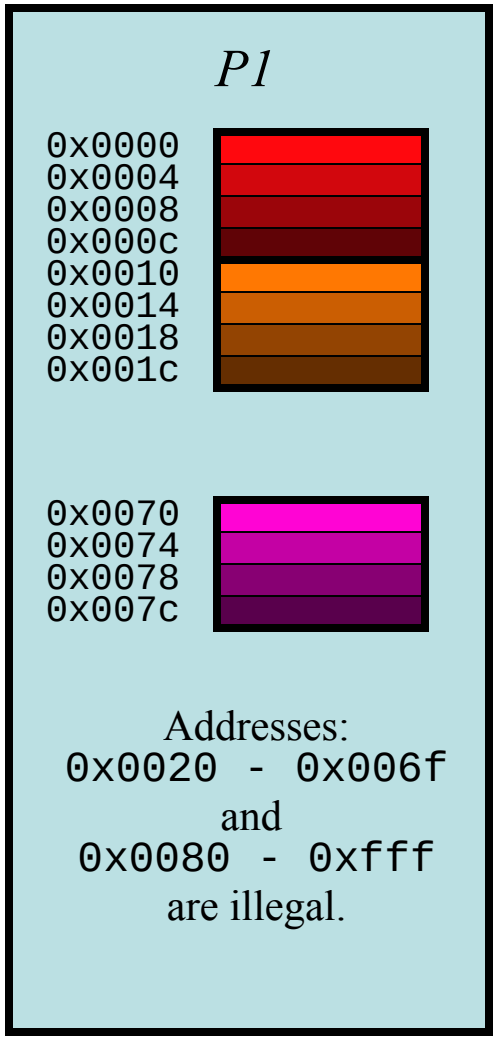


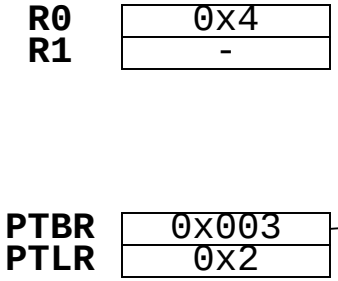
Suppose global variable j is at user location `0x0018`, and we execute:

```
mv #4 -> %r0
st %r0 -> j
```



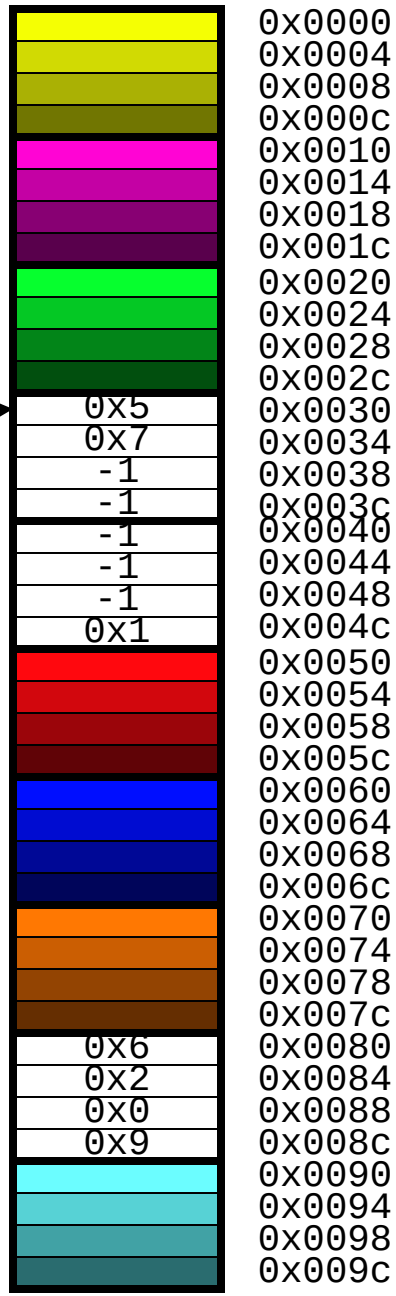
Suppose process P1 is running:



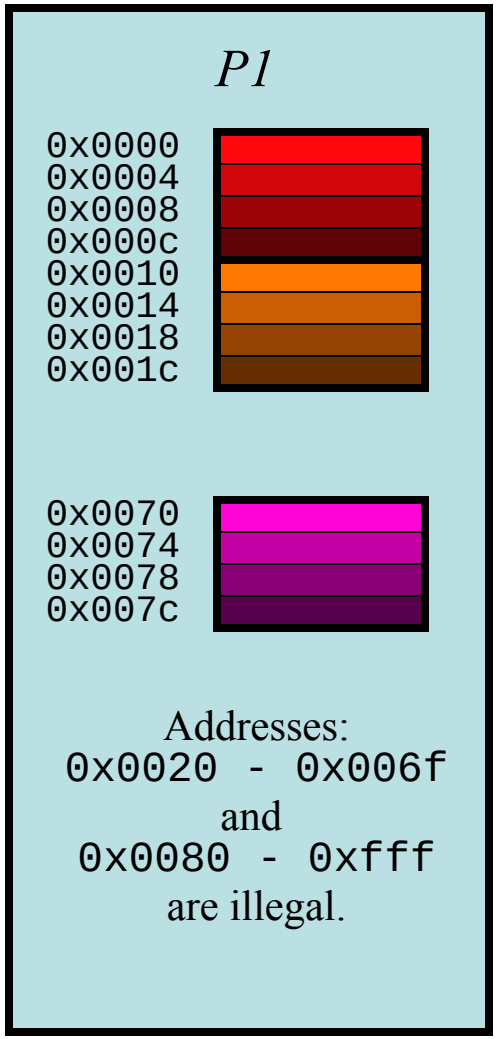


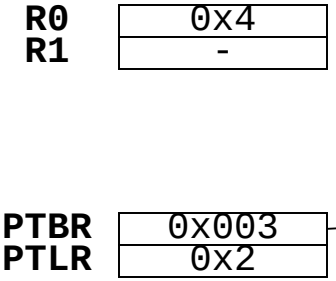
Suppose global variable j is at user location `0x0018`, and we execute:

```
mv #4 -> %r0
st %r0 -> j
```



Suppose process P1 is running:

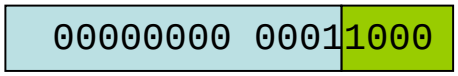




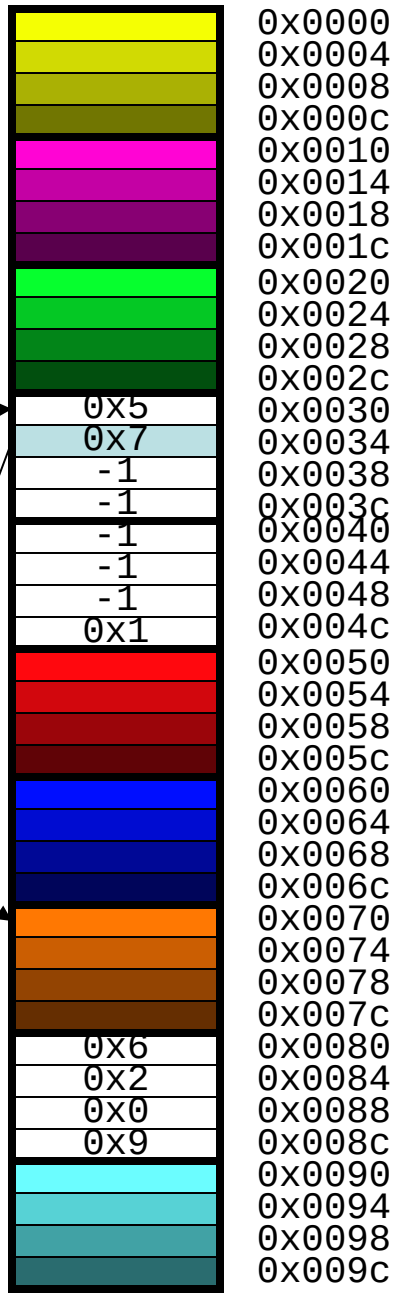
Suppose global variable j is at user location `0x0018`, and we execute:

```
mv #4 -> %r0
st %r0 -> j
```

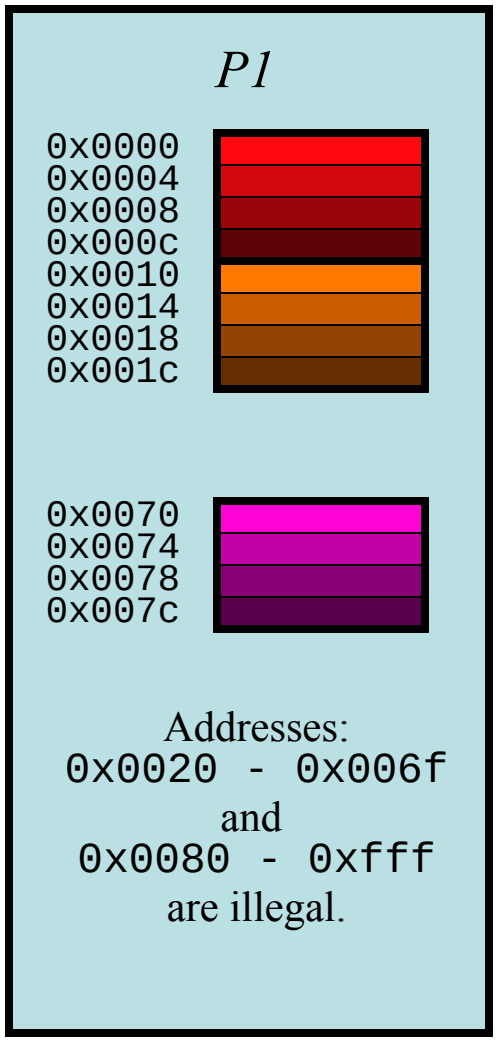
`0x0018 =`

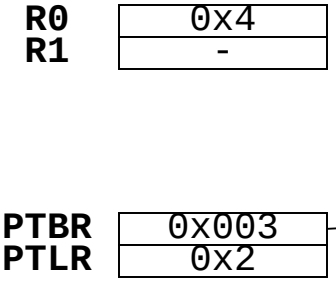


Frame 1



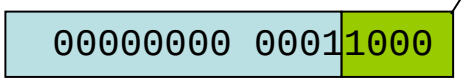
Suppose process P1 is running:

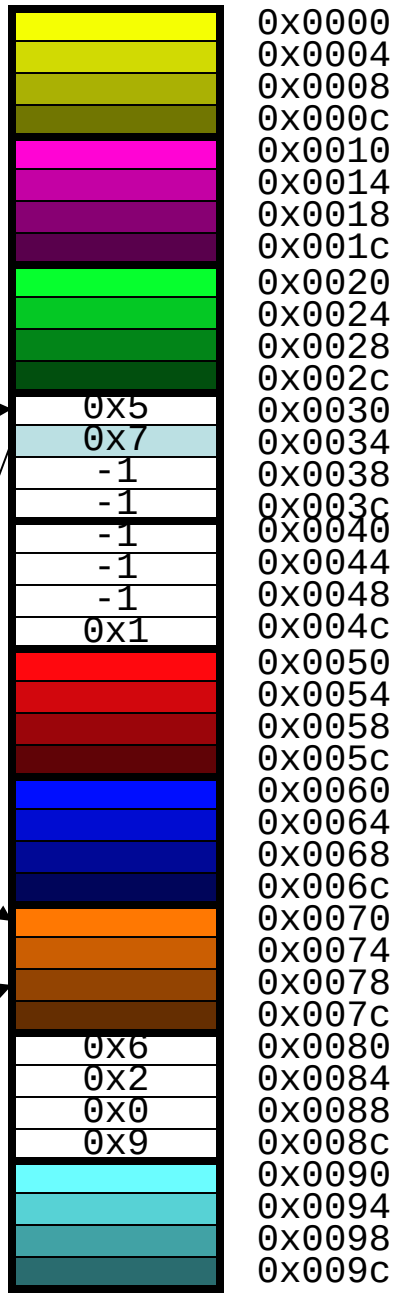




Suppose global variable j is at user location `0x0018`, and we execute:

```
mv #4 -> %r0
st %r0 -> j
```

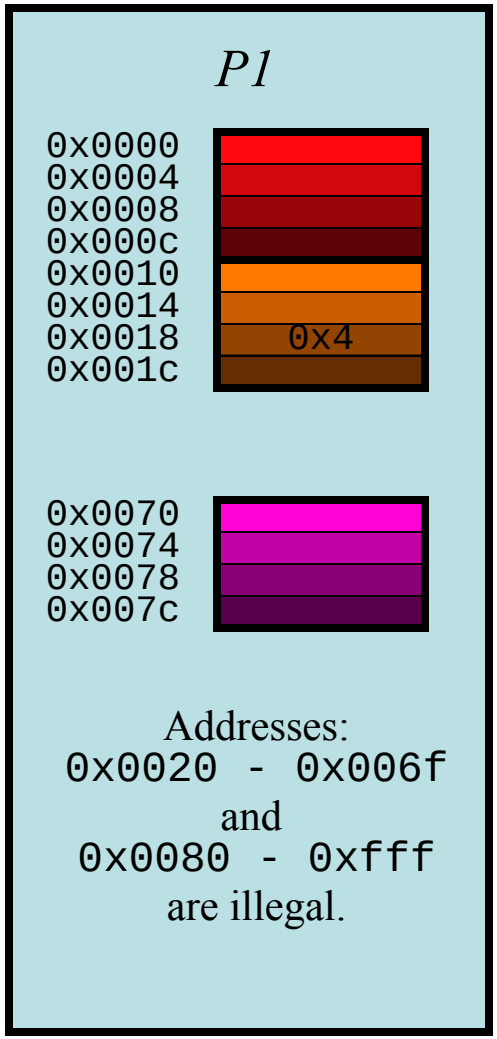


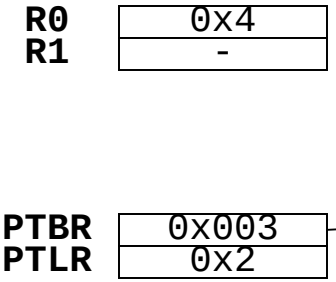


Offset 8

Suppose process P1 is running:

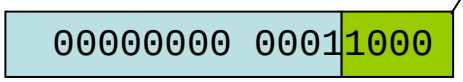
So, user address 0x0018 is physical address 0x0078.

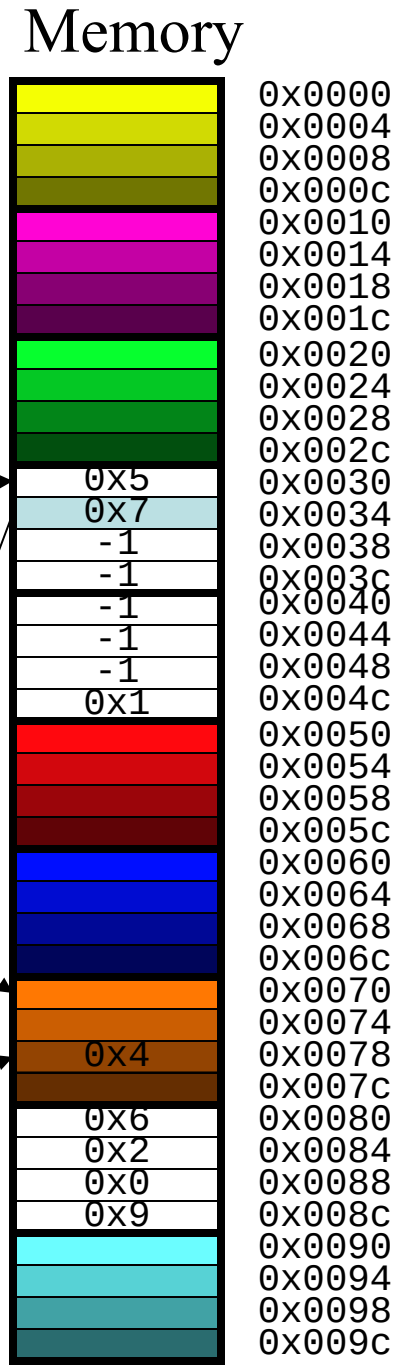




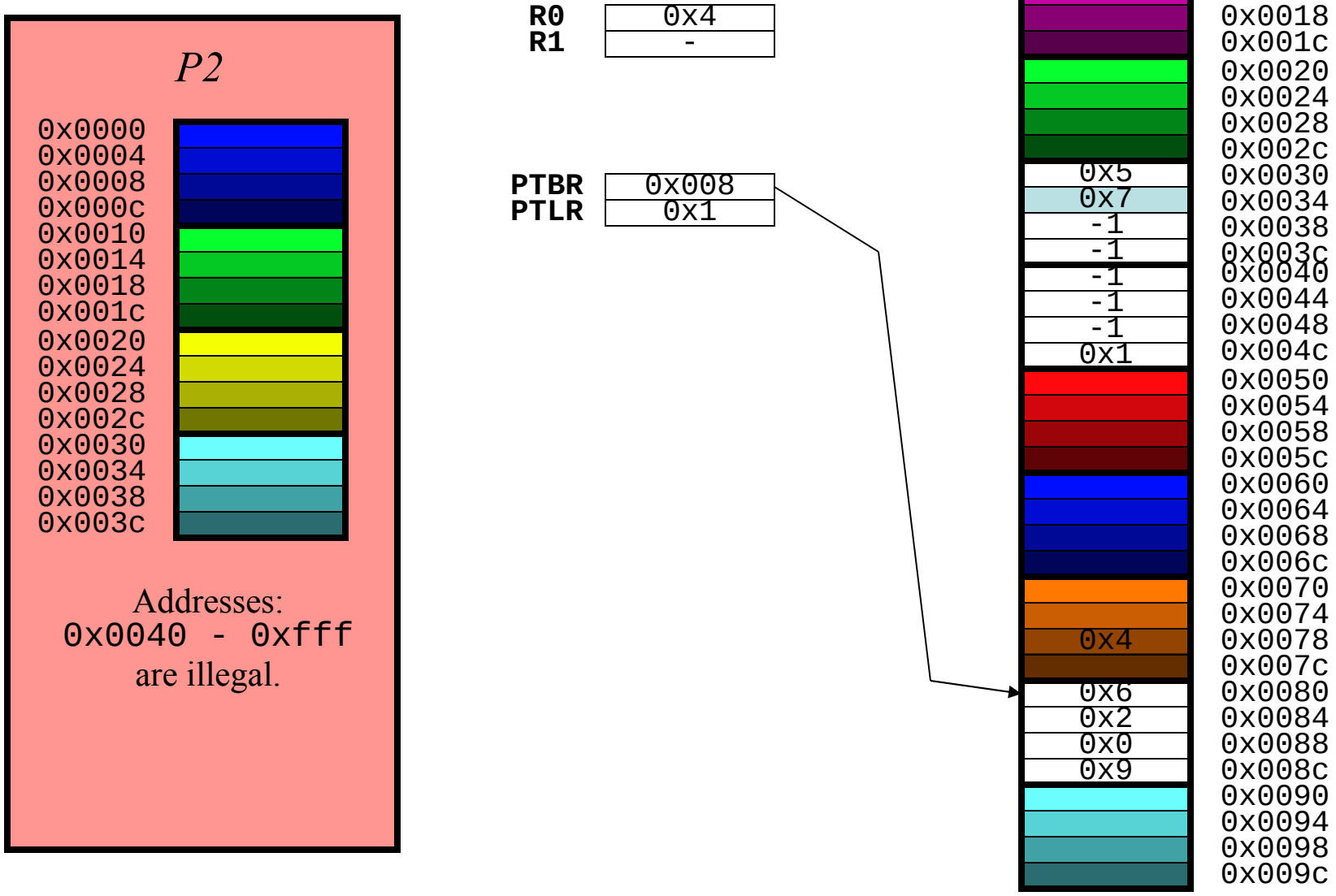
Suppose global variable j is at user location 0x0018, and we execute:

```
mv #4 -> %r0
st %r0 -> j
```

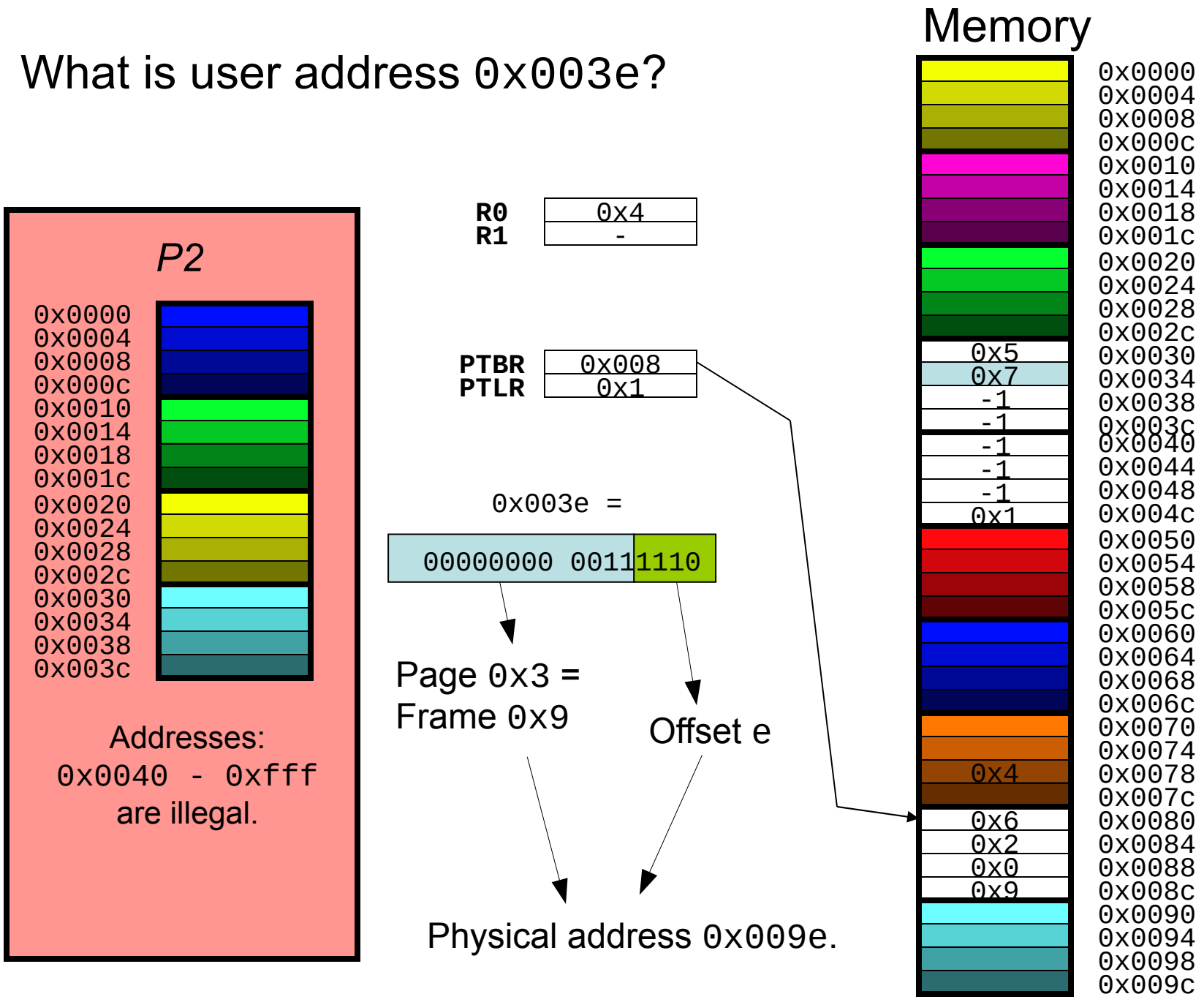




To run process *P2*, you need to switch the PTBR/PTLR:



What is user address 0x003e?

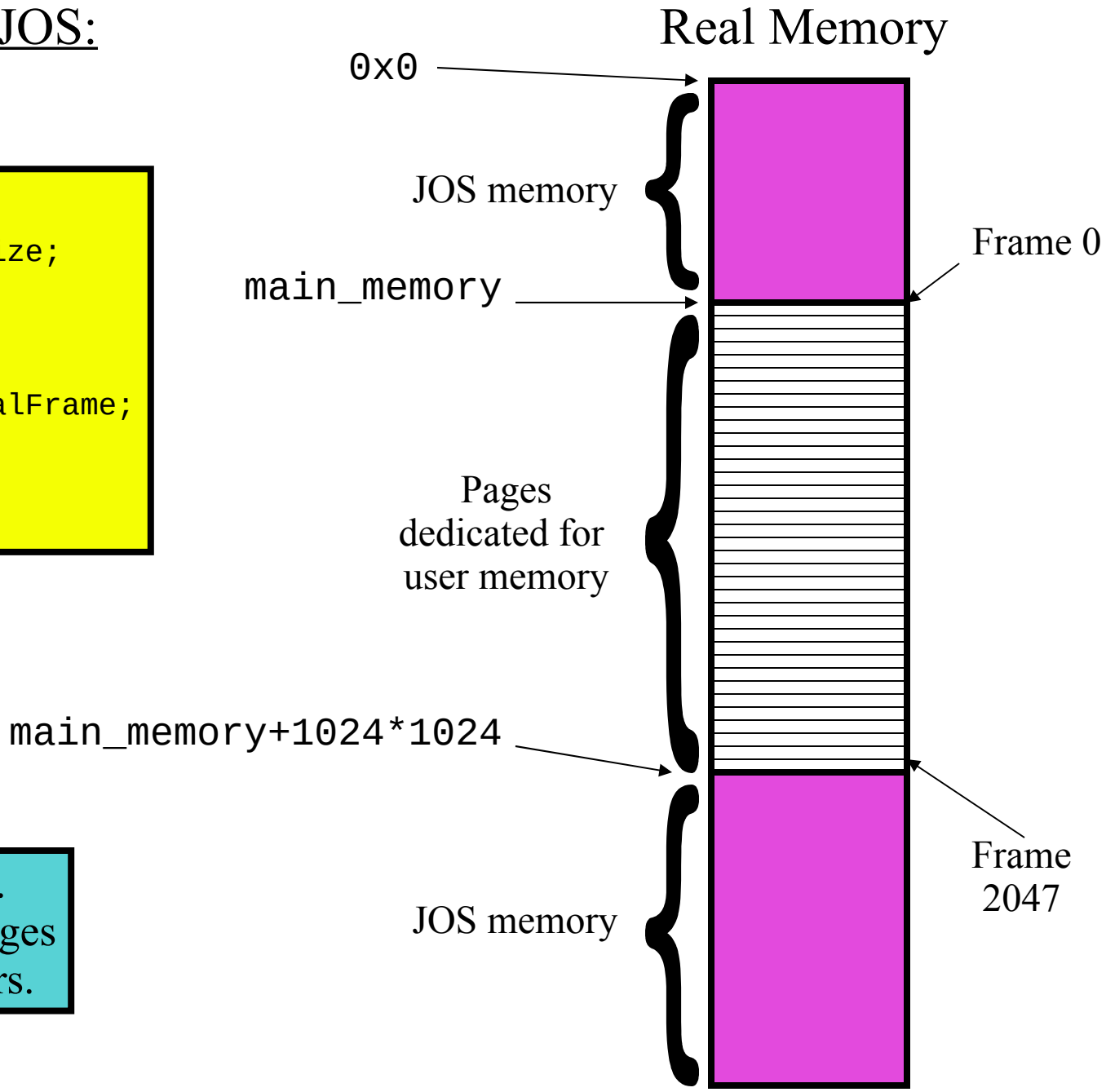


VM in JOS:

```
typedef struct {
  PTE     **table;
  int       tableSize;
} PageTable;


typedef struct {
  int       physicalFrame;
  bool      valid;
  bool      dirty;
  bool      use;
} PTE;
```

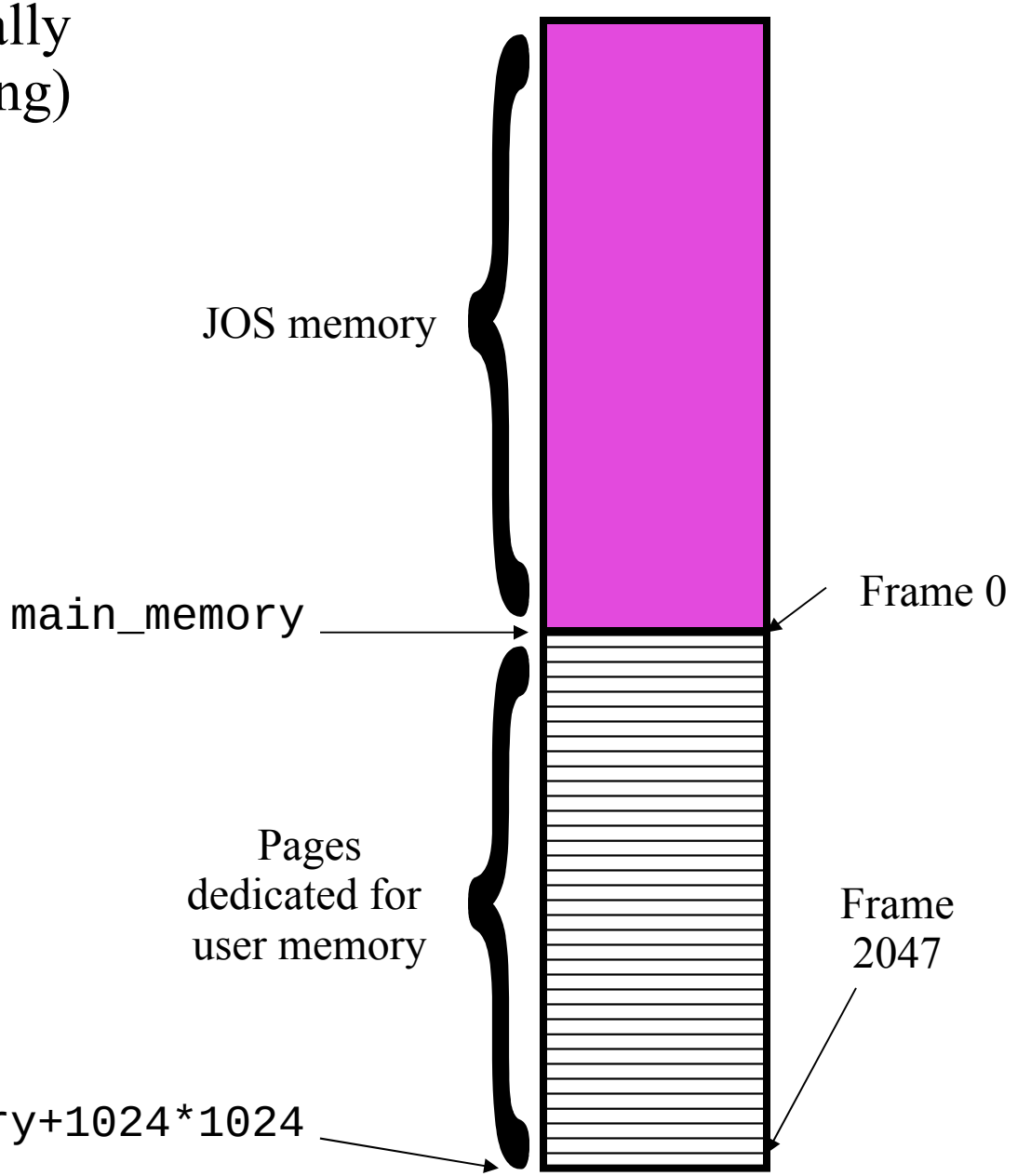
PageSize = 512.
So there are 2K pages dedicated for users.



Redraw it so it looks cleaner.
(It is not this way, but logically it is equivalent to this drawing)

```
typedef struct {
  PTE     **table;
  int       tableSize;
} PageTable;


typedef struct {
  int       physicalFrame;
  bool      valid;
  bool      dirty;
  bool      use;
} PTE;
```



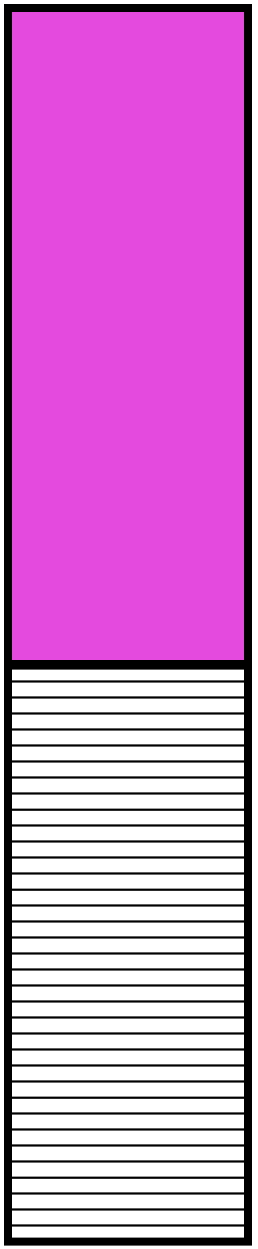
Now, suppose you have a process whose address space is as follows:

`0x0800-0x1fff`: Code
`0x2000-0x234c`: Globals
`0x7ff000-0x7fffff`: Stack

```
typedef struct {
  PTE     **table;
  int       tableSize;
} PageTable;


typedef struct {
  int       physicalFrame;
  bool      valid;
  bool      dirty;
  bool      use;
} PTE;
```

Real Memory



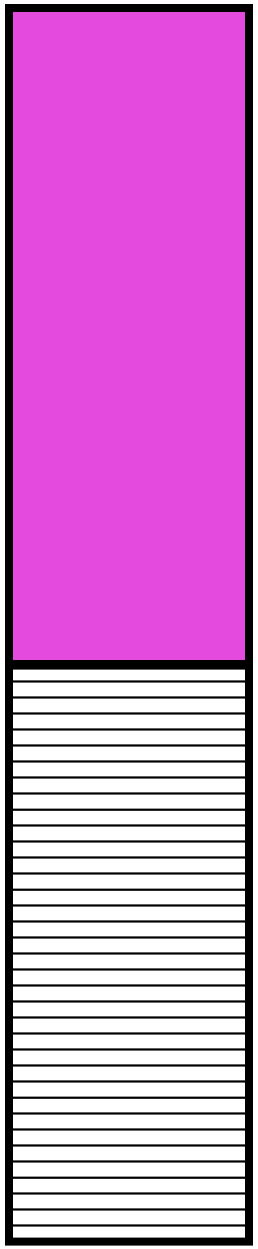
Now, suppose you have a process whose address space is as follows:

(sp = starting page)

`0x0800-0x1fff`: Code —— 12 pages: SP 4

`0x2000-0x234c`: Globals —— 2 pages: SP 16

`0x7ff000-0x7fffff`: Stack – 8 pages: SP 16376



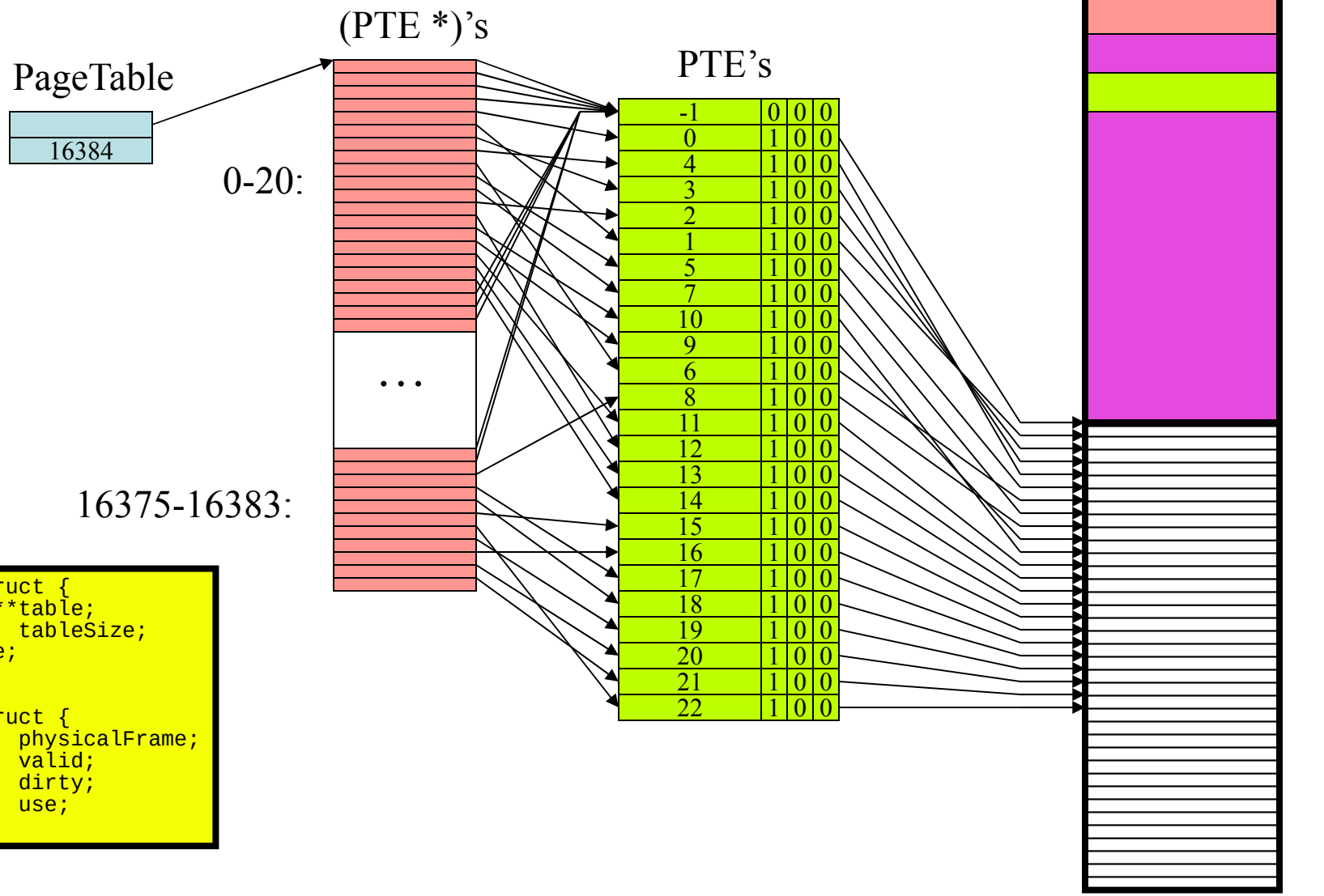

```
typedef struct {
  PTE     **table;
  int       tableSize;
} PageTable;


typedef struct {
  int       physicalFrame;
  bool      valid;
  bool      dirty;
  bool      use;
} PTE;
```

0x0800-0x1fff: Code —— 12 pages: SP 1

0x2000-0x234c: Globals — 2 pages: SP 16

0x7fd000-0x7fffff: Stack – 8 pages: SP 16376




```
typedef struct {
  PTE     **table;
  int       tableSize;
} PageTable;


typedef struct {
  int       physicalFrame;
  bool      valid;
  bool      dirty;
  bool      use;
} PTE;
```

New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

    - 2G address spaces
    - Page 0 is NULL
    - Code starts at page 1
    - 1 NULL page between code & globals
    - Heap starts 512M from beginning of address space.
    - Stack starts at the last address & grows up, but is of fixed size.

New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

Q1: Show what a pointer looks like in terms of page number / offset:

New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

Q2: Suppose we have a user process with 3 pages of code, 2 pages of globals, 4 pages of heap, and a 4-page stack.

Specify the contents of each byte of the user's address space:

New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

Q2: Suppose we have a user process with 3 pages of code, 2 pages of globals, 4 pages of heap, and a 4-page stack.

Specify the contents of each byte of the user's address space:

```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

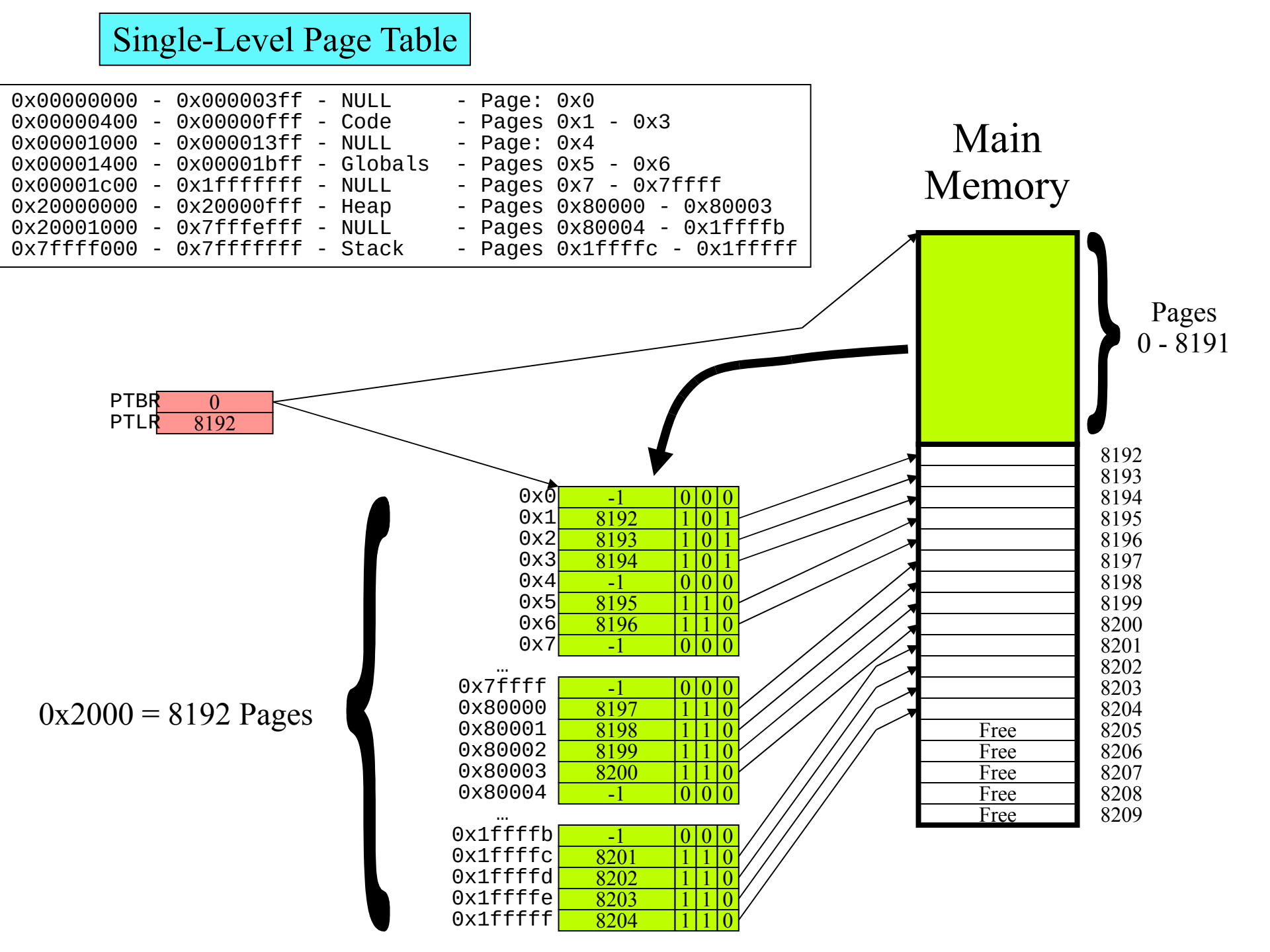
Q3: Now, suppose we have a single-level page table, implemented with a PTBR/PTLR.

Draw an example memory.

```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

Single-Level Page Table

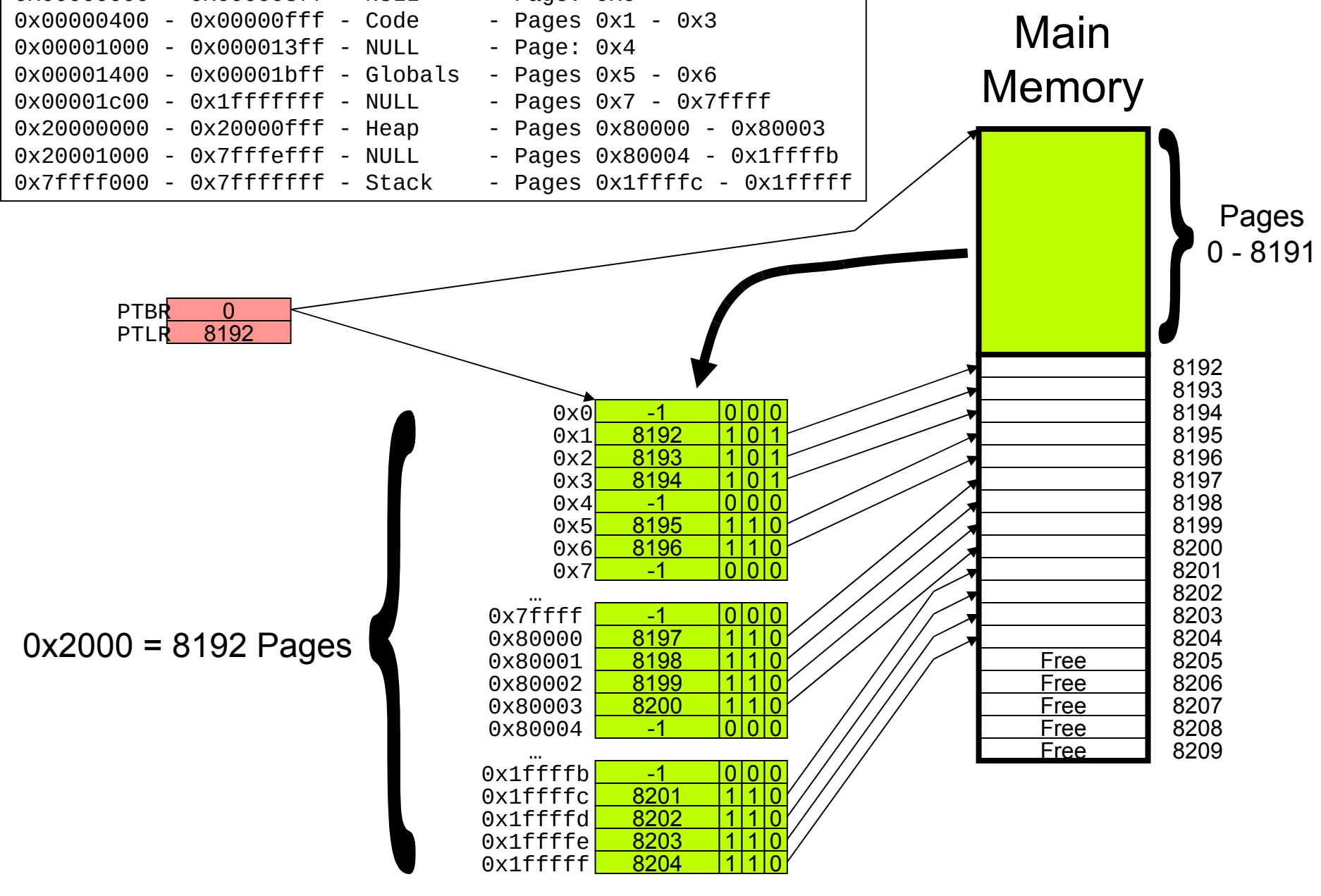
```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1fffc - 0x1fffff
```



Single-Level Page Table

Memory usage: 8205 KB: Horrible!

```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1fffc - 0x1fffff
```

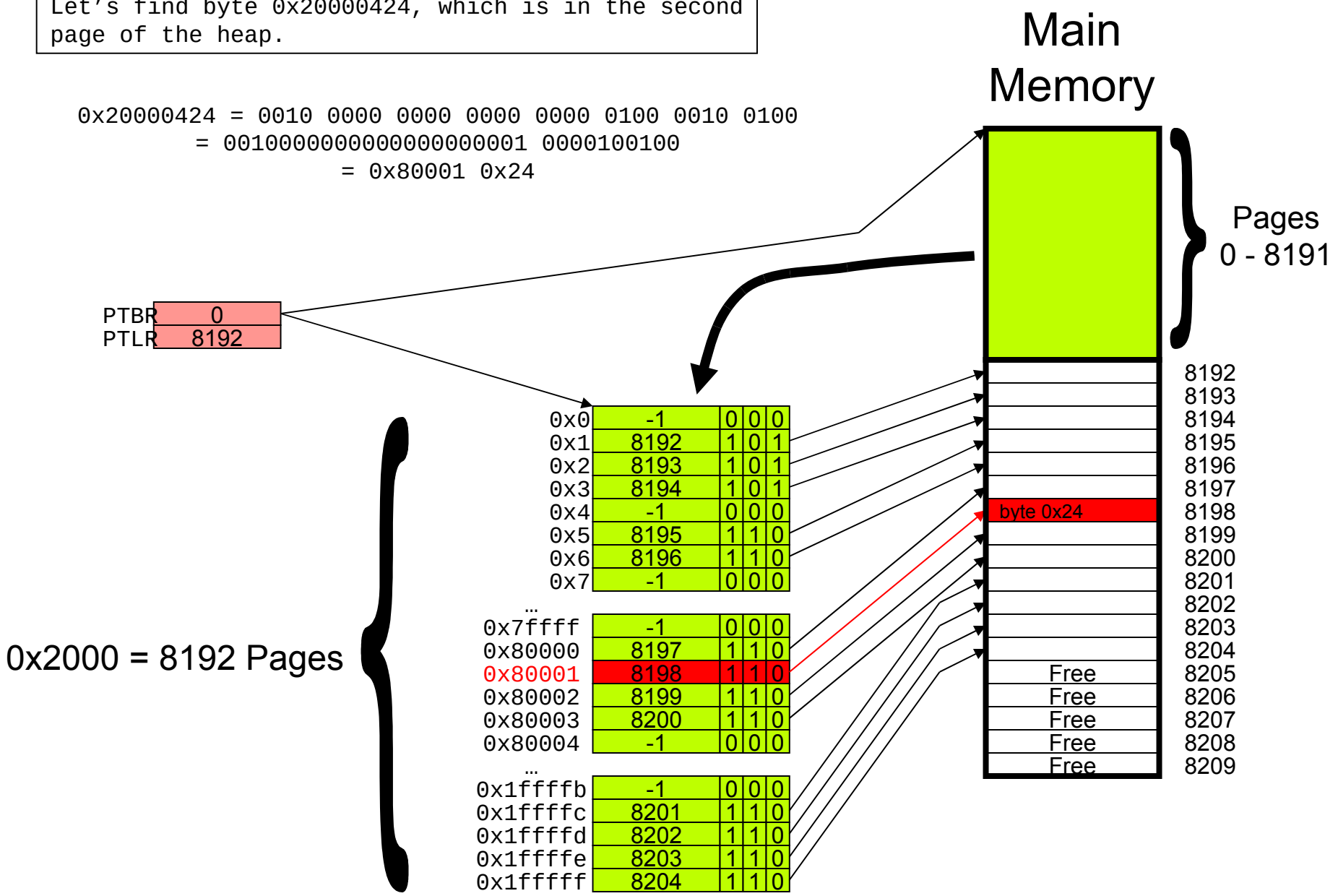


Single-Level Page Table

Physical page 8198, offset 0x24:
0x2006 0x24 = .. 0010 0000 0000 0110 00 0010 0100 =
0000 0000 1000 0000 0001 1000 0010 0100 =
0x00801824

Let's find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 00100000000000000001 0000100100
= 0x80001 0x24

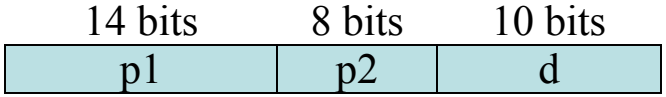


New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

Now, try a two-level page table:



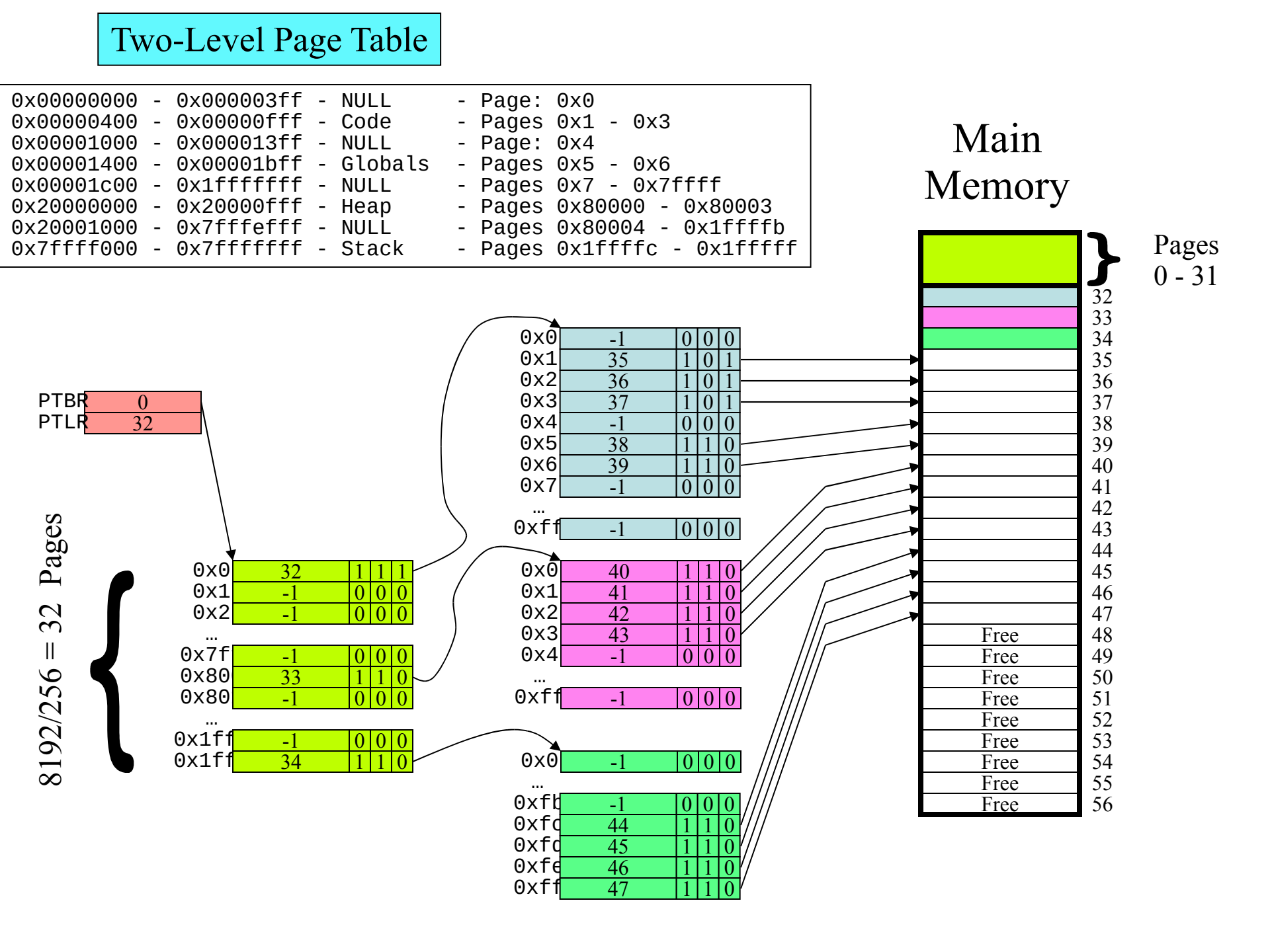
d = 10 bits: 1K pages
p2 = 8 bits: 256 PTEs/Page
p1 = 14 bits: Remainder

There are 2^{13} = 8K first-level PTE's.
Why? The first bit is always 0.

```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

Two-Level Page Table

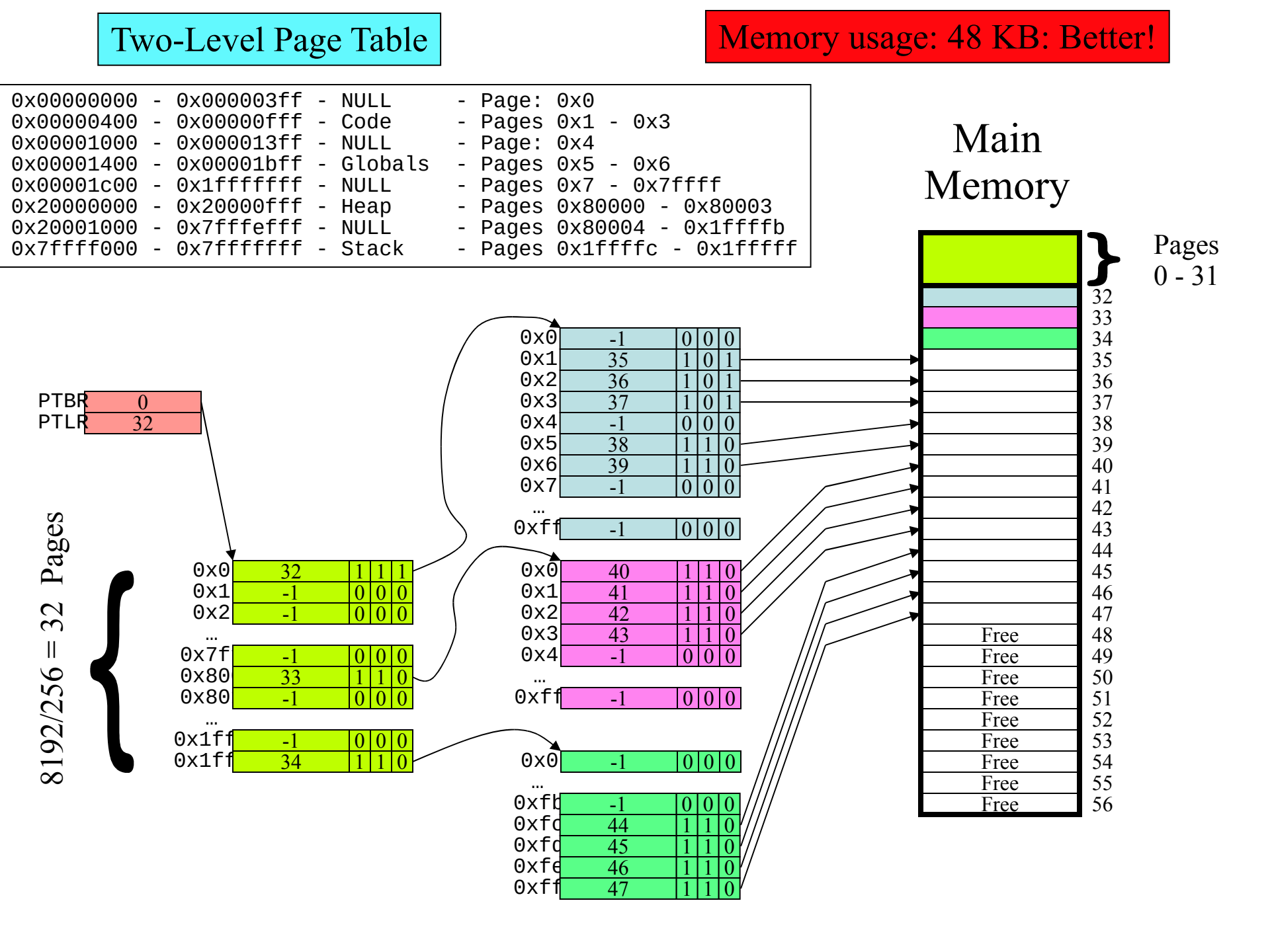
```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1ffffc - 0x1fffff
```



Two-Level Page Table

Memory usage: 48 KB: Better!

```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1fffc - 0x1fffff
```

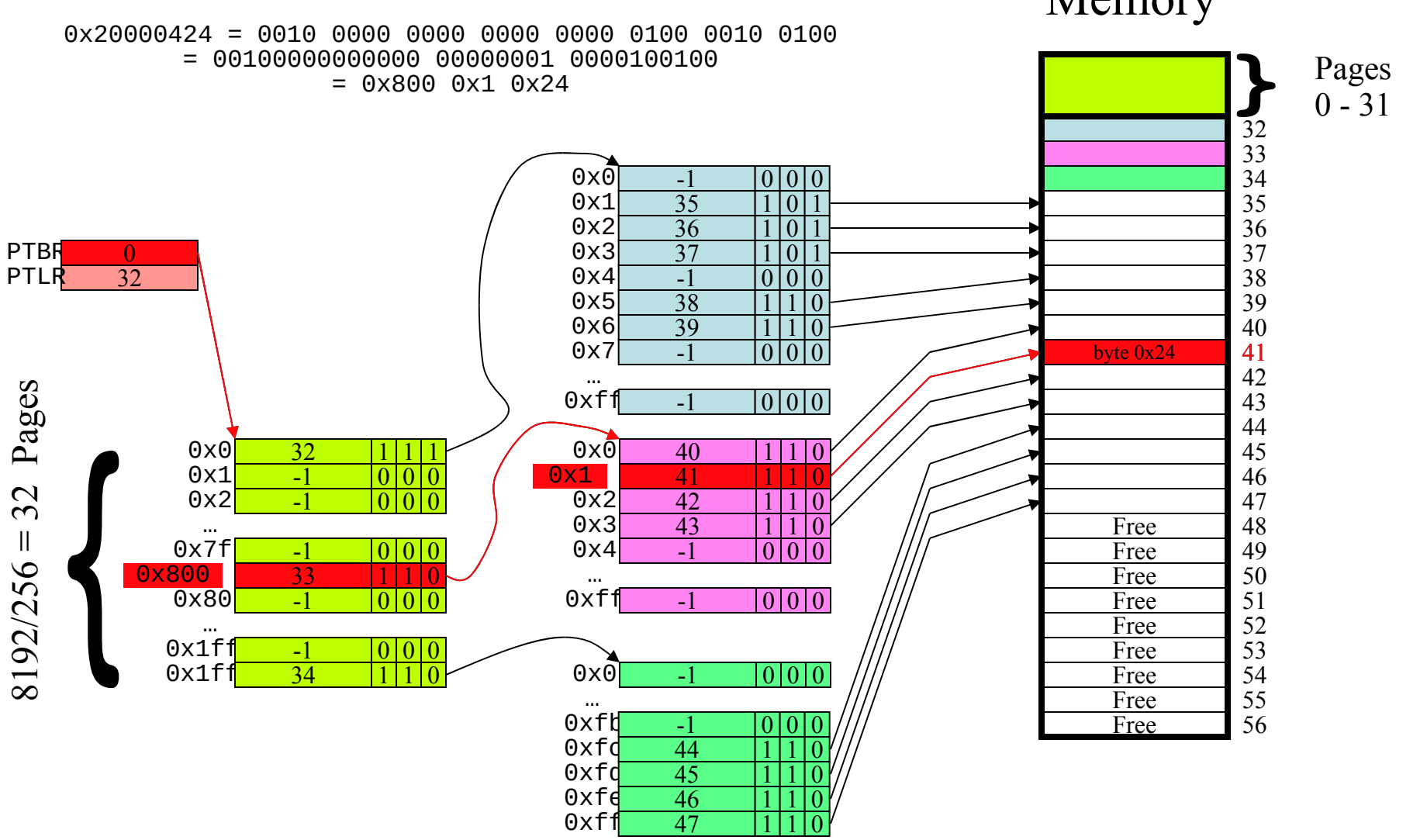


Two-Level Page Table

Let's find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 00100000000000 00000001 0000100100
= 0x800 0x1 0x24

Physical page 41, offset 0x24:
0x29 0x24 = .. 0000 0010 1001 00 0010 0100 =
0000 0000 0000 0000 1010 0100 0010 0100 =
0x0000a424

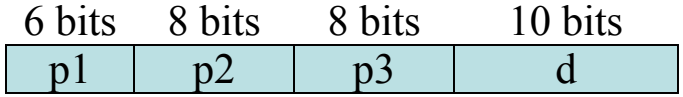


New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

How about a 3-level page table:



d = 10 bits: 1K pages
p2/3 = 8 bits: 256 PTEs/Page
p1 = 6 bits: Remainder

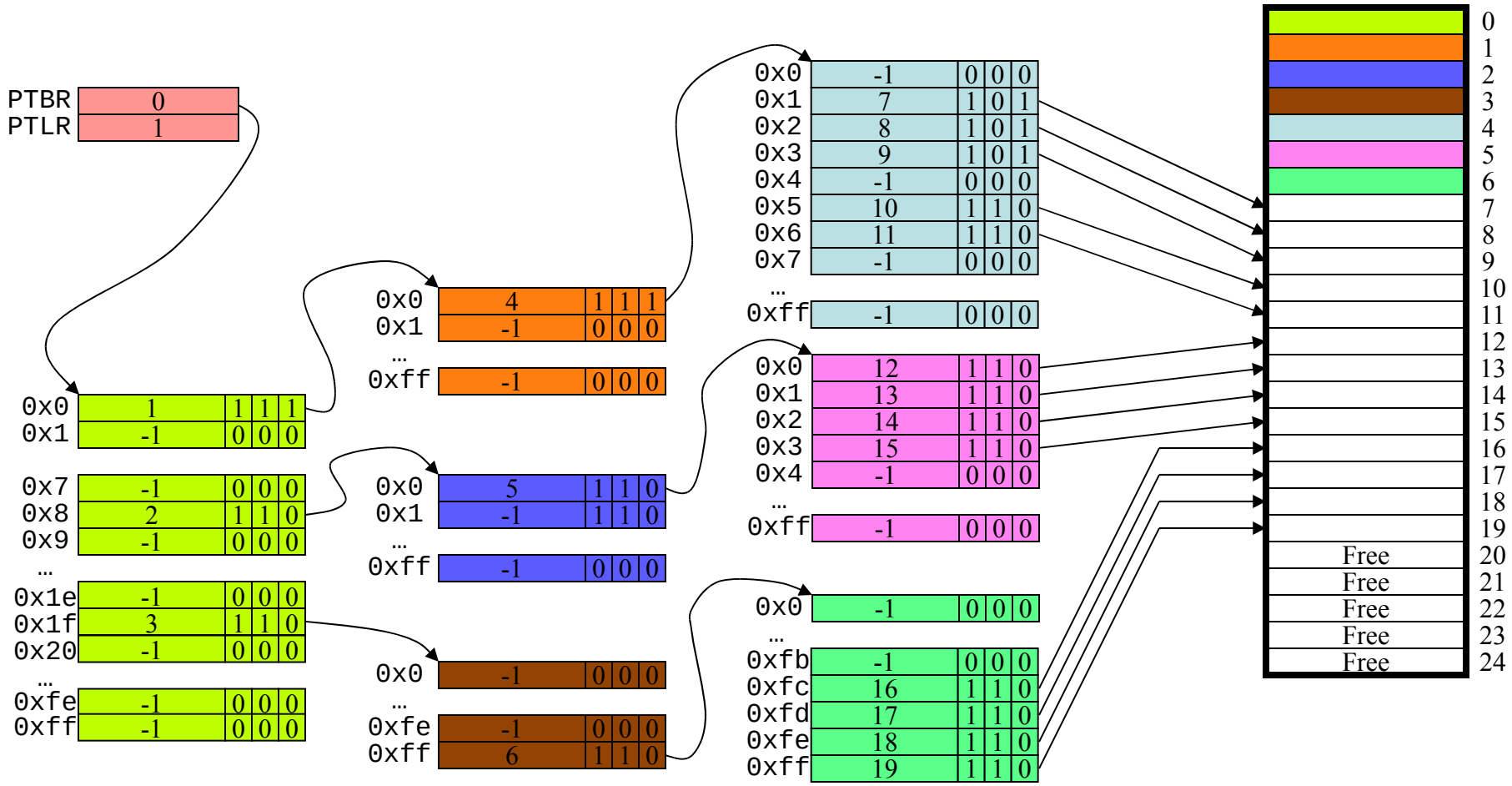
There are $2^5 = 32$ first-level PTE's.

```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

Three-Level Page Table

Memory usage: 20 KB: Excellent!

```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1fffc - 0x1fffff
```

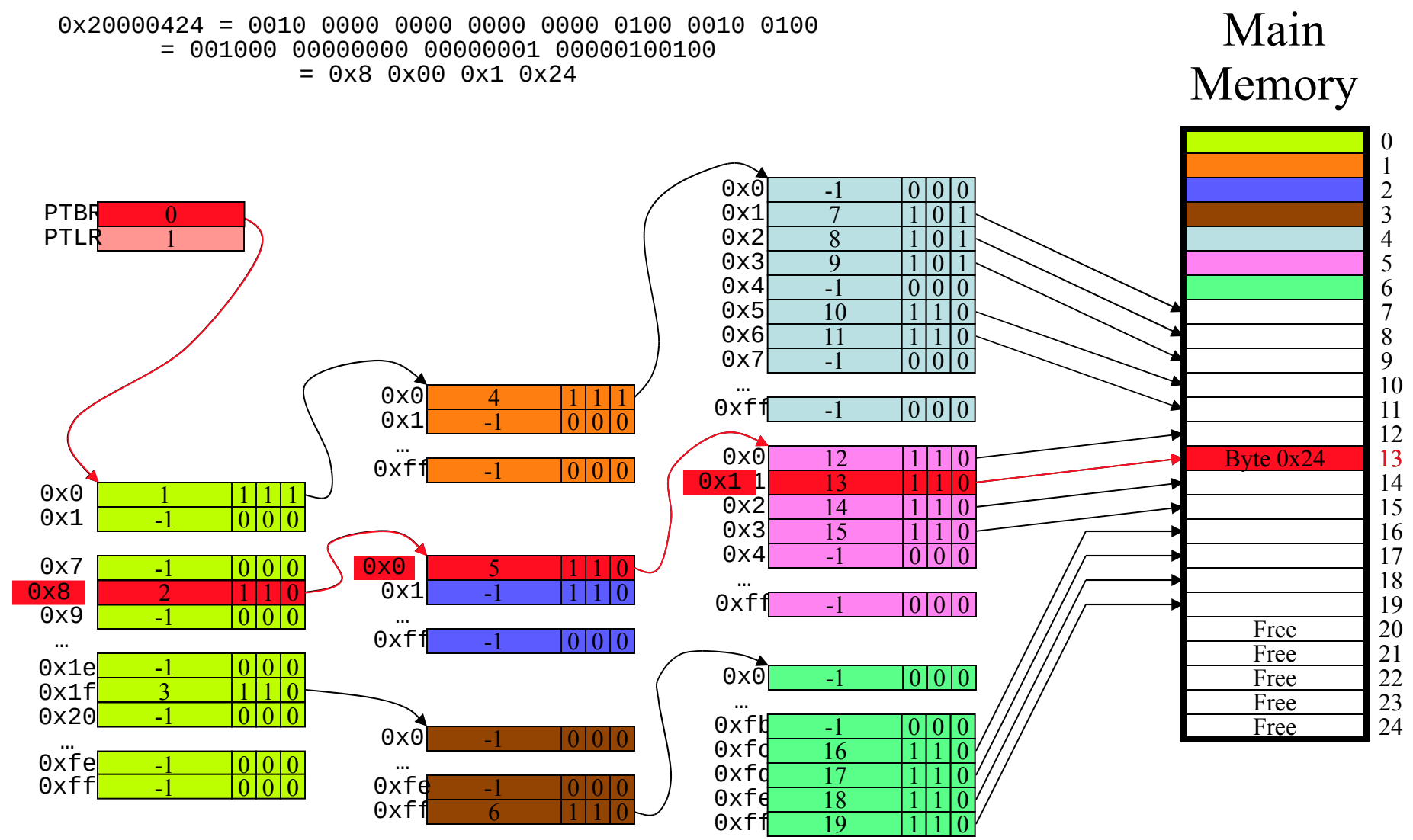


Three-Level Page Table

Let's find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 001000 00000000 00000001 00000100100
= 0x8 0x00 0x1 0x24

Physical page 13, offset 0x24:
0x0d 0x24 = .. 0000 0000 1101 00 0010 0100 =
0000 0000 0000 0000 0011 0100 0010 0100 =
0x00003424

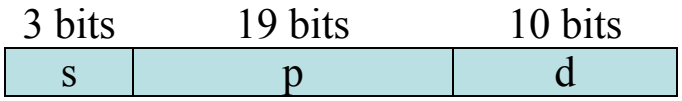


New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

Now, how about a segmented scheme as follows:



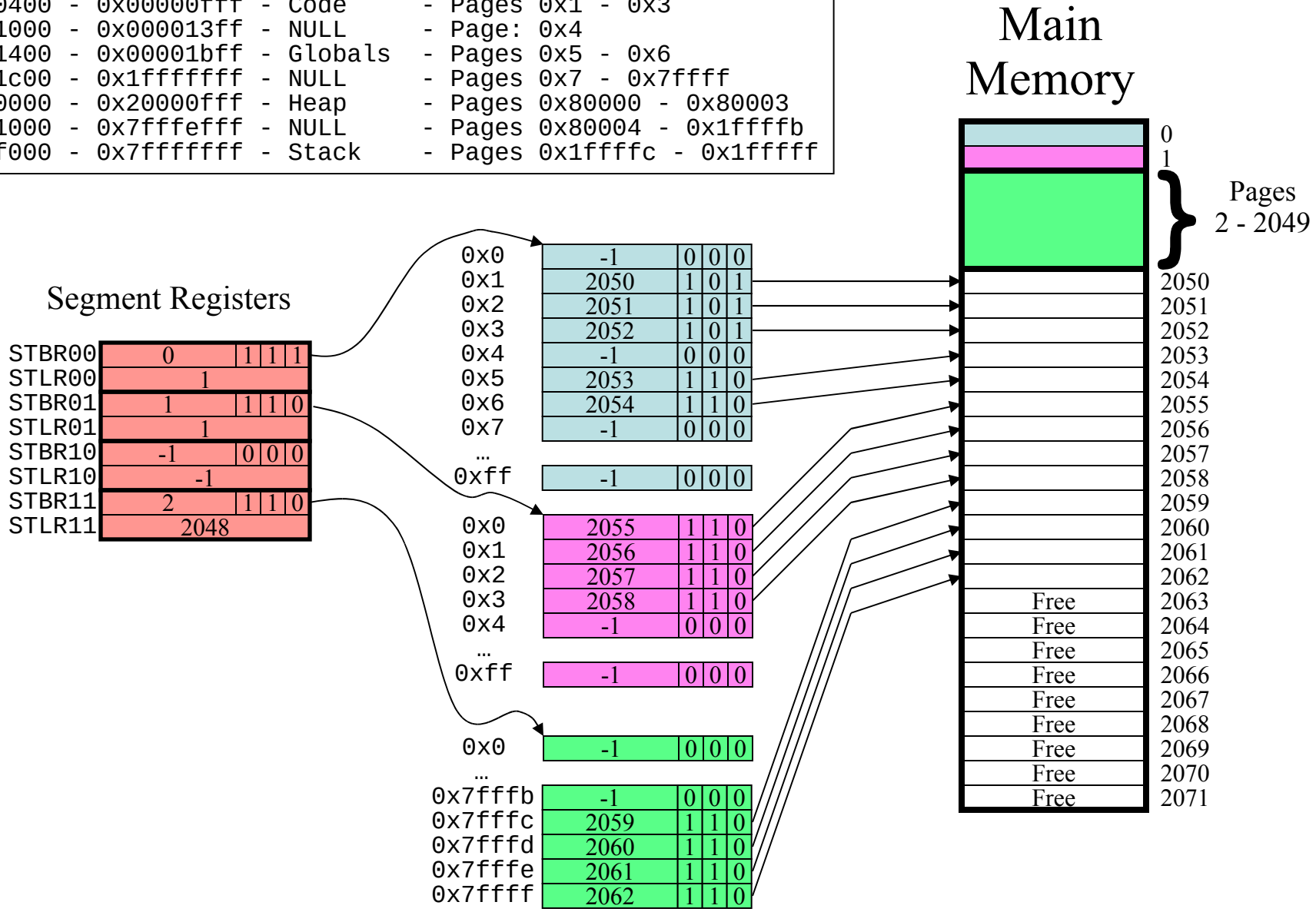
There are just four segments, defined by four STBR/STLR pairs of registers.

```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

Segmented Page Table

Memory usage: 2063 KB: Bad!

```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1fffc - 0x1fffff
```

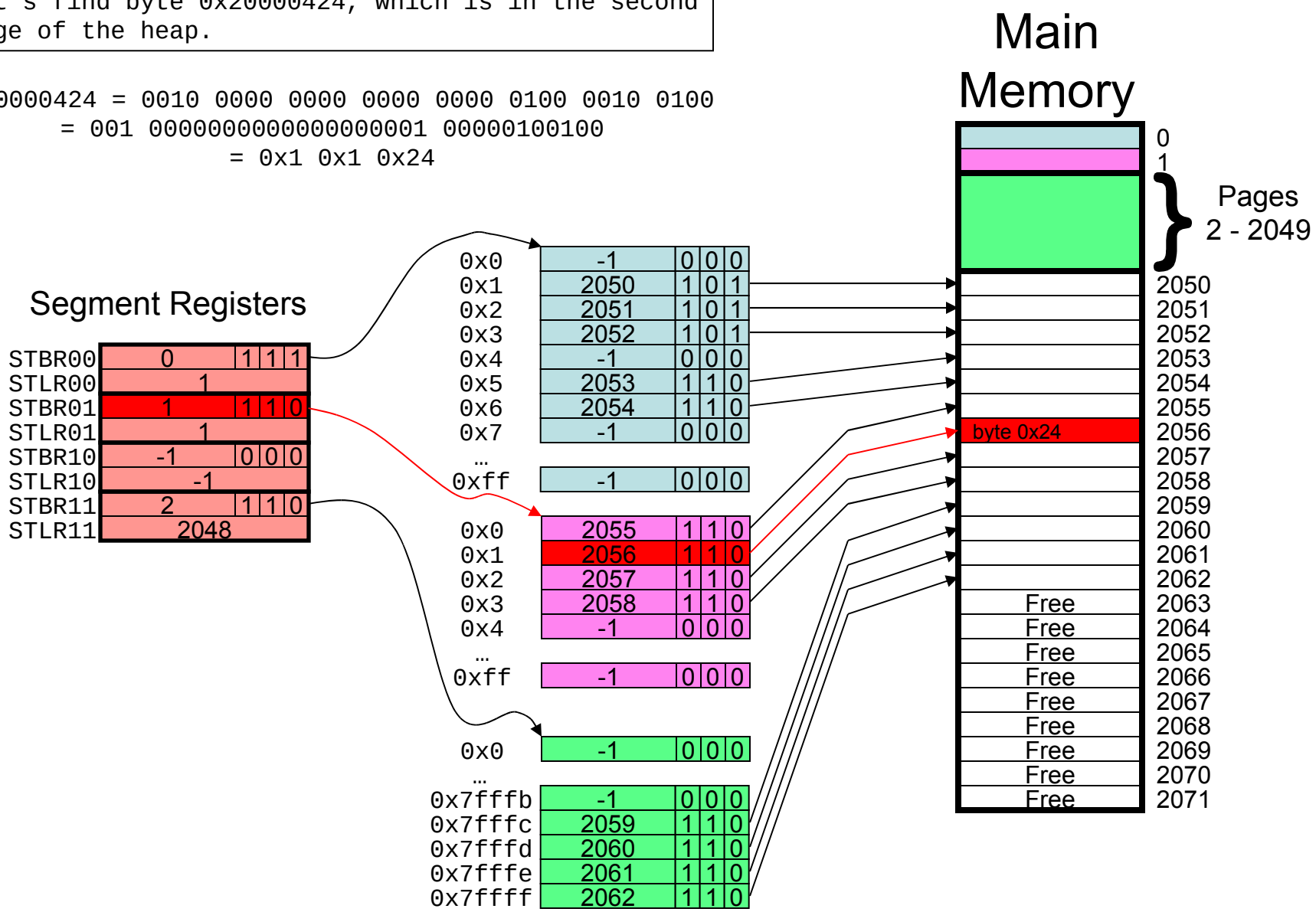


Segmented Page Table

Let's find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 001 00000000000000001 00000100100
= 0x1 0x1 0x24

Physical page 2056, offset 0x24:
0x808 0x24 = .. 1000 0000 1000 00 0010 0100 =
0000 0000 0010 0000 0010 0000 0010 0100 =
0x00202024

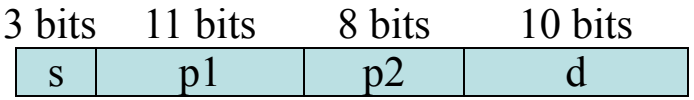


New Example - Our machine has:

- 256M of RAM.
- 1K pages.
- 32-bit pointers; byte-addressed.
- PTE's are 4 bytes & point to actual frame numbers.
- PTE's also have valid, write, execute, dirty, use bits.
- Page tables are allocated from user memory.
- User processes laid out as follows:

  - 2G address spaces
  - Page 0 is NULL
  - Code starts at page 1
  - 1 NULL page between code & globals
  - Heap starts 512M from beginning of address space
  - Stack starts at the last address & grows up, but is of fixed size.

Finally, let's add another layer of paging:

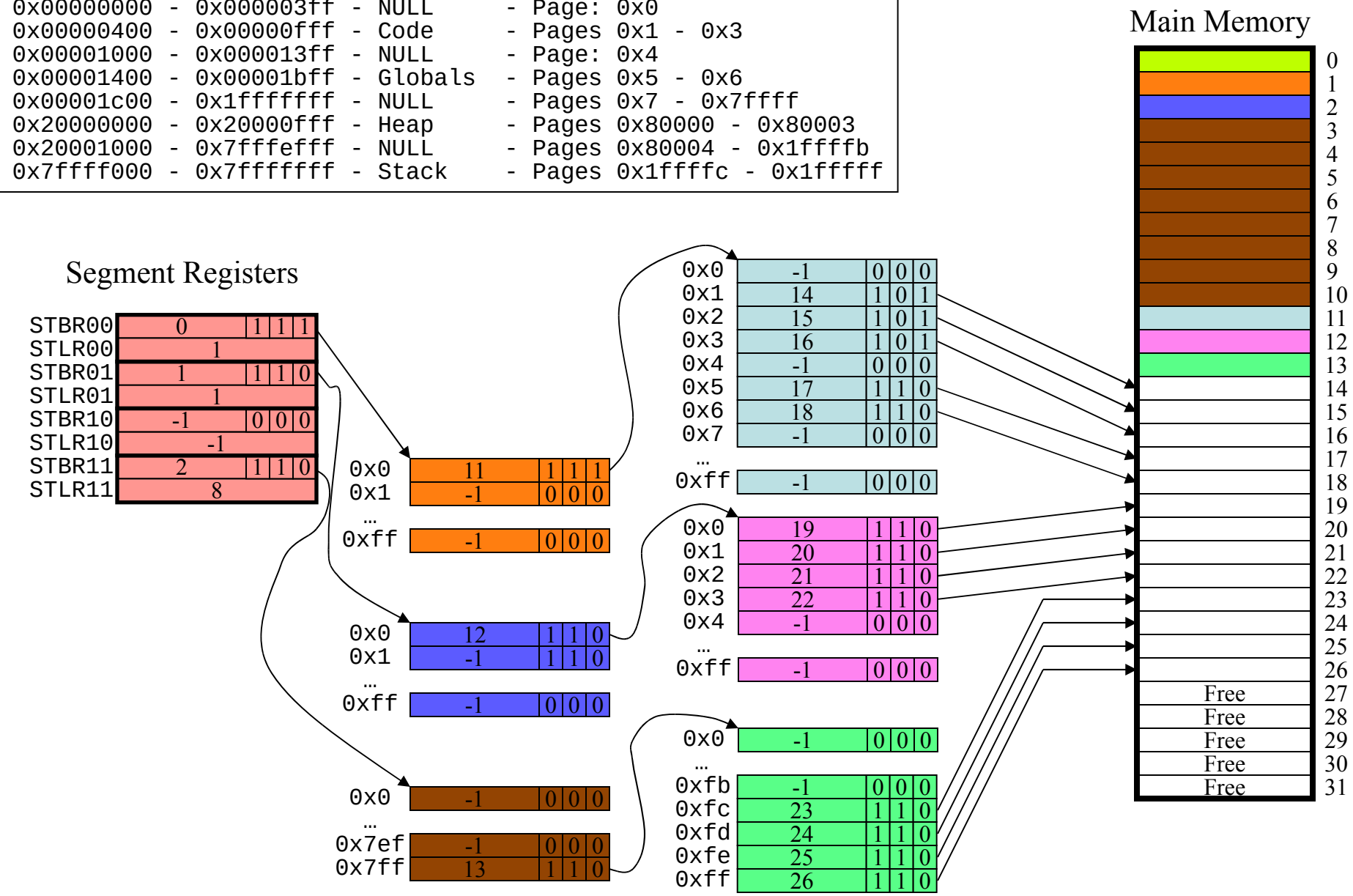



```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

Two-Level Segments

Memory usage: 27 KB: Good!

```
0x00000000 - 0x000003ff - NULL      - Page: 0x0
0x00000400 - 0x00000fff - Code      - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL      - Page: 0x4
0x00001400 - 0x00001bff - Globals   - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL      - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap      - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL      - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack     - Pages 0x1fffc - 0x1fffff
```

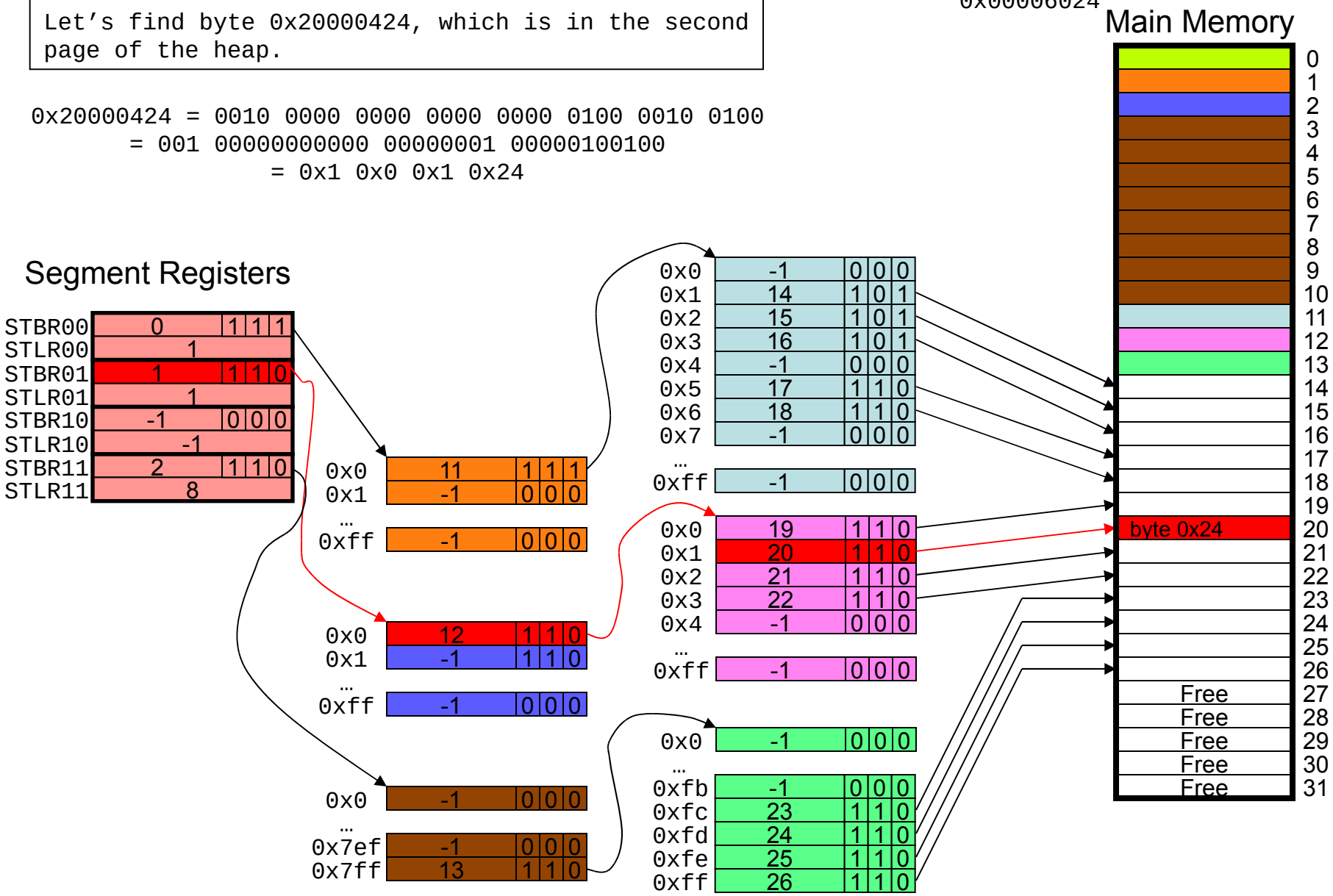


Two-Level Segments

Let's find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 001 00000000000 00000001 00000100100
= 0x1 0x0 0x1 0x24

Physical page 20, offset 0x24:
0x14 0x24 = .. 0000 0001 1000 00 0010 0100 =
0000 0000 0000 0000 0110 0000 0010 0100 =
0x00006024

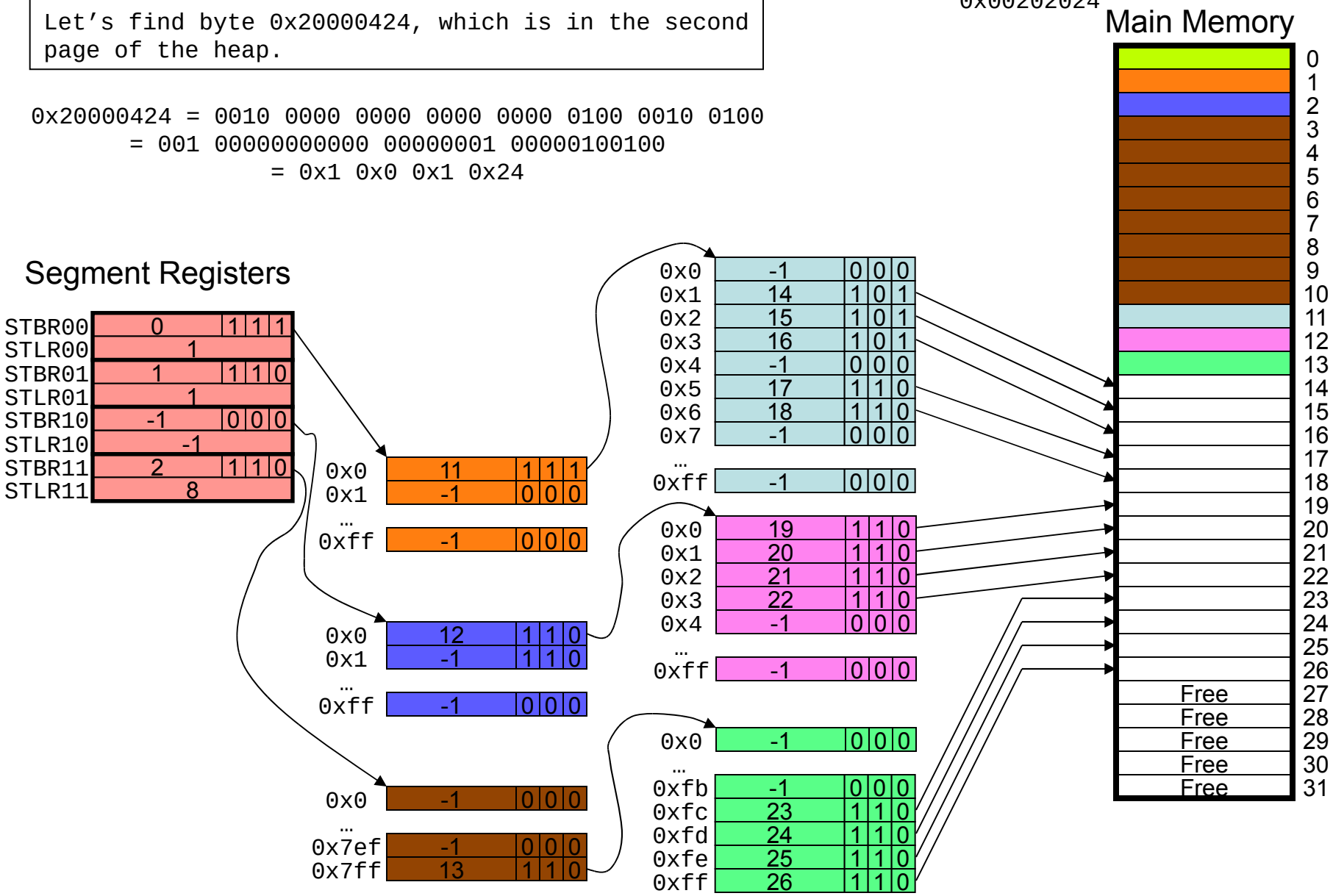


Two-Level Segments

Let's find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 001 00000000000 00000001 00000100100
= 0x1 0x0 0x1 0x24

Physical page 2056, offset 0x24:
0x808 0x24 = .. 1000 0000 1000 00 0010 0100 =
0000 0000 0010 0000 0010 0000 0010 0100 =
0x00202024



Inverted Page Table

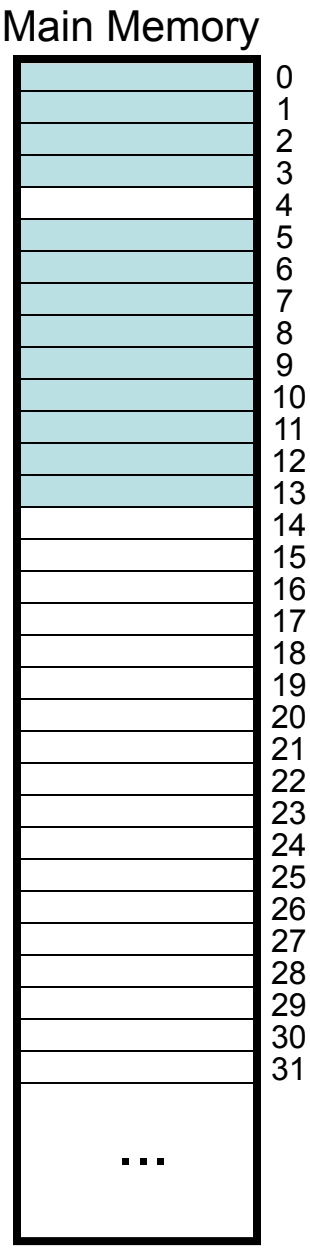
```
0x00000000 - 0x000003ff - NULL     - Page: 0x0
0x00000400 - 0x00000fff - Code     - Pages 0x1 - 0x3
0x00001000 - 0x000013ff - NULL     - Page: 0x4
0x00001400 - 0x00001bff - Globals  - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL     - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap     - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffefff - NULL     - Pages 0x80004 - 0x1fffb
0x7ffff000 - 0x7fffffff - Stack    - Pages 0x1fffc - 0x1fffff
```

In this example, we draw the inverted page table as not sharing the processes' memory. In this example, we splatter the pages of the address space among the first 13 pages. We also assume that the PID of the process is 551.

So, the code is in physical pages 0, 1 and 12. The globals are in pages 13 and 5. The heap pages are 2, 6, 11 and 9. The stack is in pages 7, 10, 3 and 8.

Since there are 256x1024 pages of main memory, there are 256x1024 PTE's in the inverted page table.

	page	v	w	x	pid
0x0	0x1	1	0	1	551
0x1	0x2	1	0	1	551
0x2	0x80000	1	1	0	551
0x3	0x1ffffe	1	1	0	551
0x4	-1	0	0	0	-1
0x5	0x6	1	1	0	551
0x6	0x80001	1	1	0	551
0x7	0x1ffffc	1	1	0	551
0x8	0x1fffff	1	1	0	551
0x9	0x80003	1	1	0	551
0xa	0x1ffffd	1	1	0	-1
0xb	0x80002	1	1	0	551
0xc	0x3	1	0	1	551
0xd	0x5	1	1	0	551
0xe	-1	0	0	0	-1
...					...
0x3fffe	-1	0	0	0	-1
0x3ffff	-1	0	0	0	-1



Inverted Page Table

Let's find byte 0x20000424, which is in the second page of the heap.

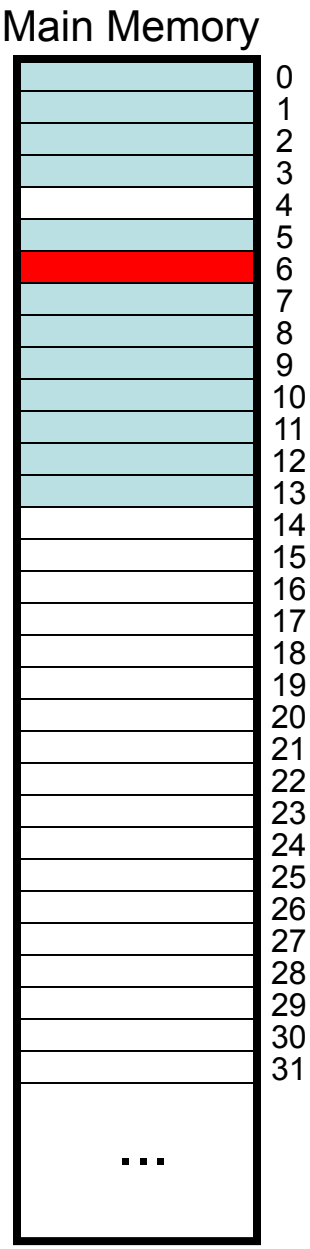
0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 00100000000000000001 00000100100
= 0x80001 0x24

We perform a linear search of the inverted page table until we find page 0x80001 and PID 551. That is page 0x6.

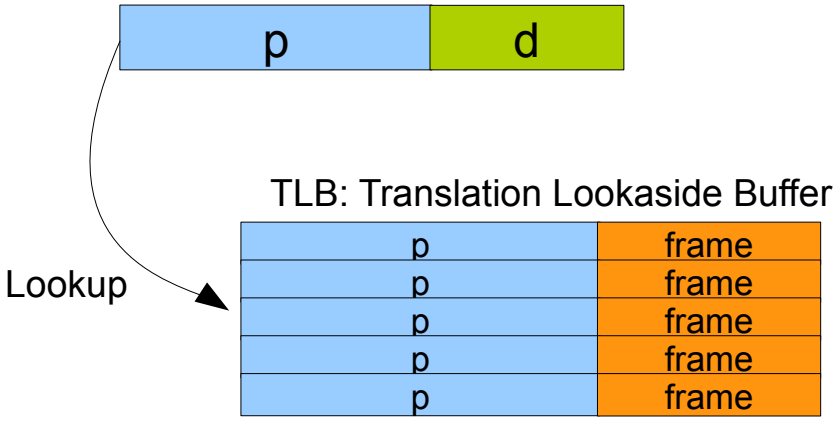
So the address is 0x6 0x24:

110 0000100100 = 0x1824

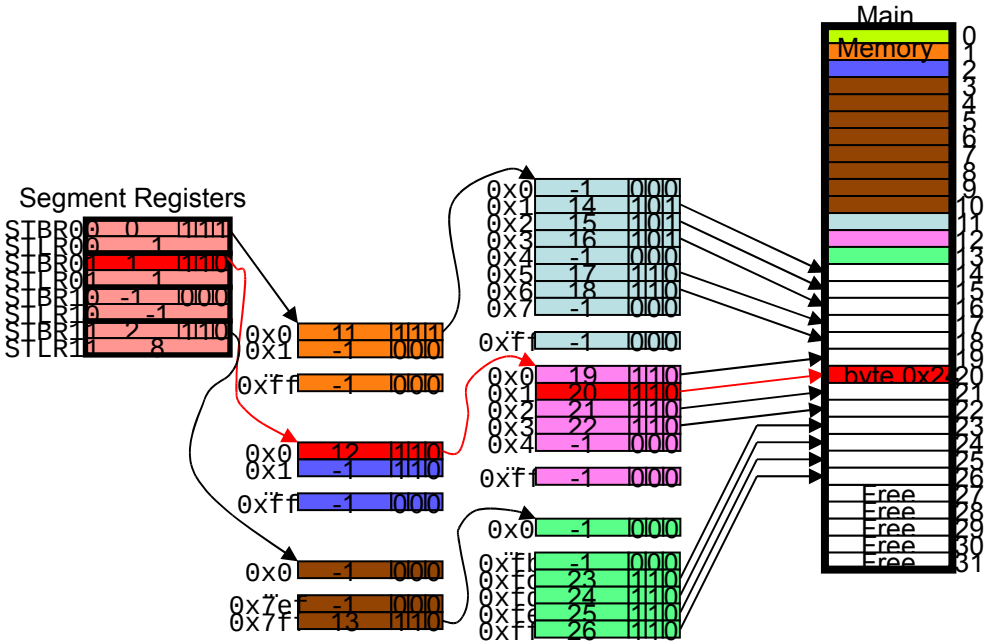
	page	v	w	x	pid
0x0	0x1	1	0	1	551
0x1	0x2	1	0	1	551
0x2	0x80000	1	1	0	551
0x3	0x1ffffe	1	1	0	551
0x4	-1	0	0	0	-1
0x5	0x6	1	1	0	551
0x6	0x80001	1	1	0	551
0x7	0x1ffffc	1	1	0	551
0x8	0x1fffff	1	1	0	551
0x9	0x80003	1	1	0	551
0xa	0x1ffffd	1	1	0	-1
0xb	0x80002	1	1	0	551
0xc	0x3	1	0	1	551
0xd	0x5	1	1	0	551
0xe	-1	0	0	0	-1
...					...
0x3fffe	-1	0	0	0	-1
0x3ffff	-1	0	0	0	-1



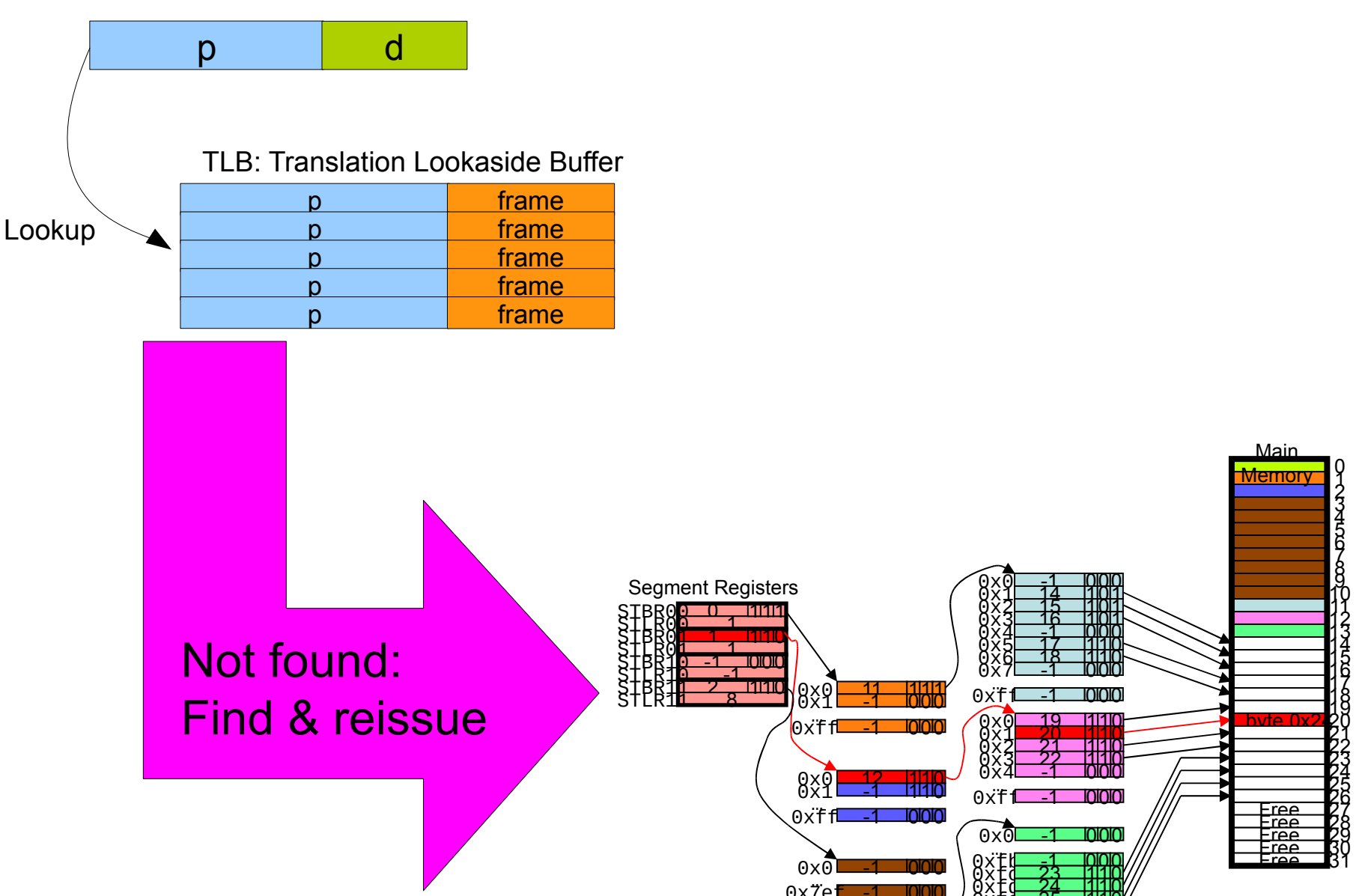
With a TLB & Cache



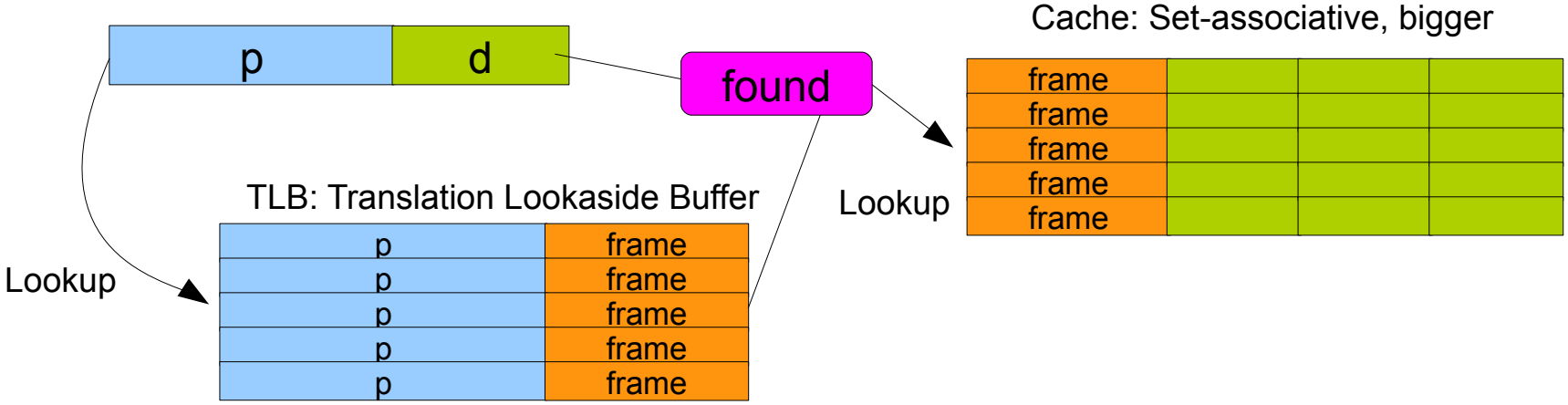
Associative Memory:
Small:
128-512 entries

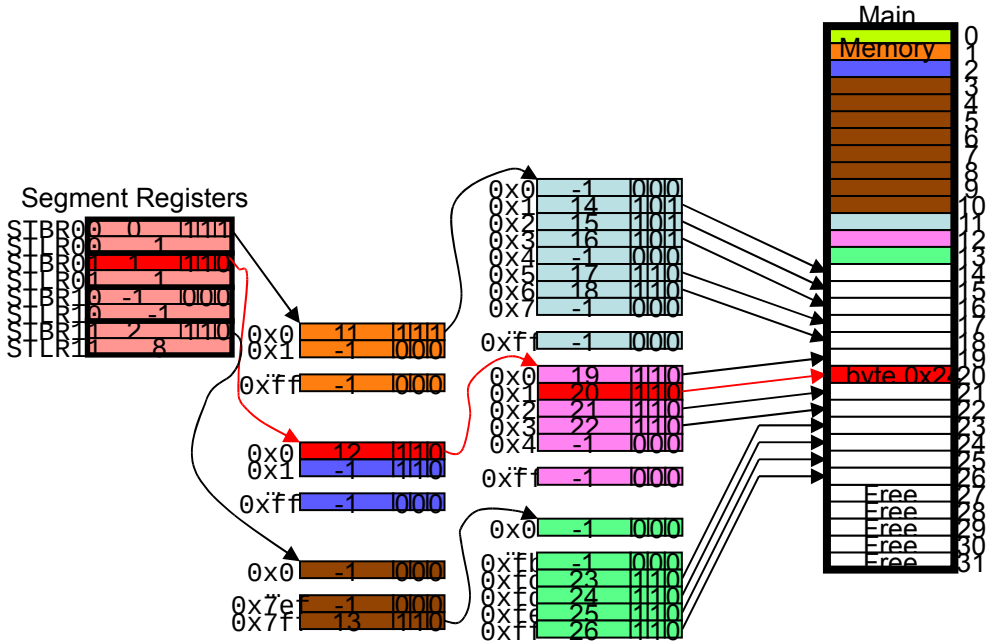


With a TLB & Cache

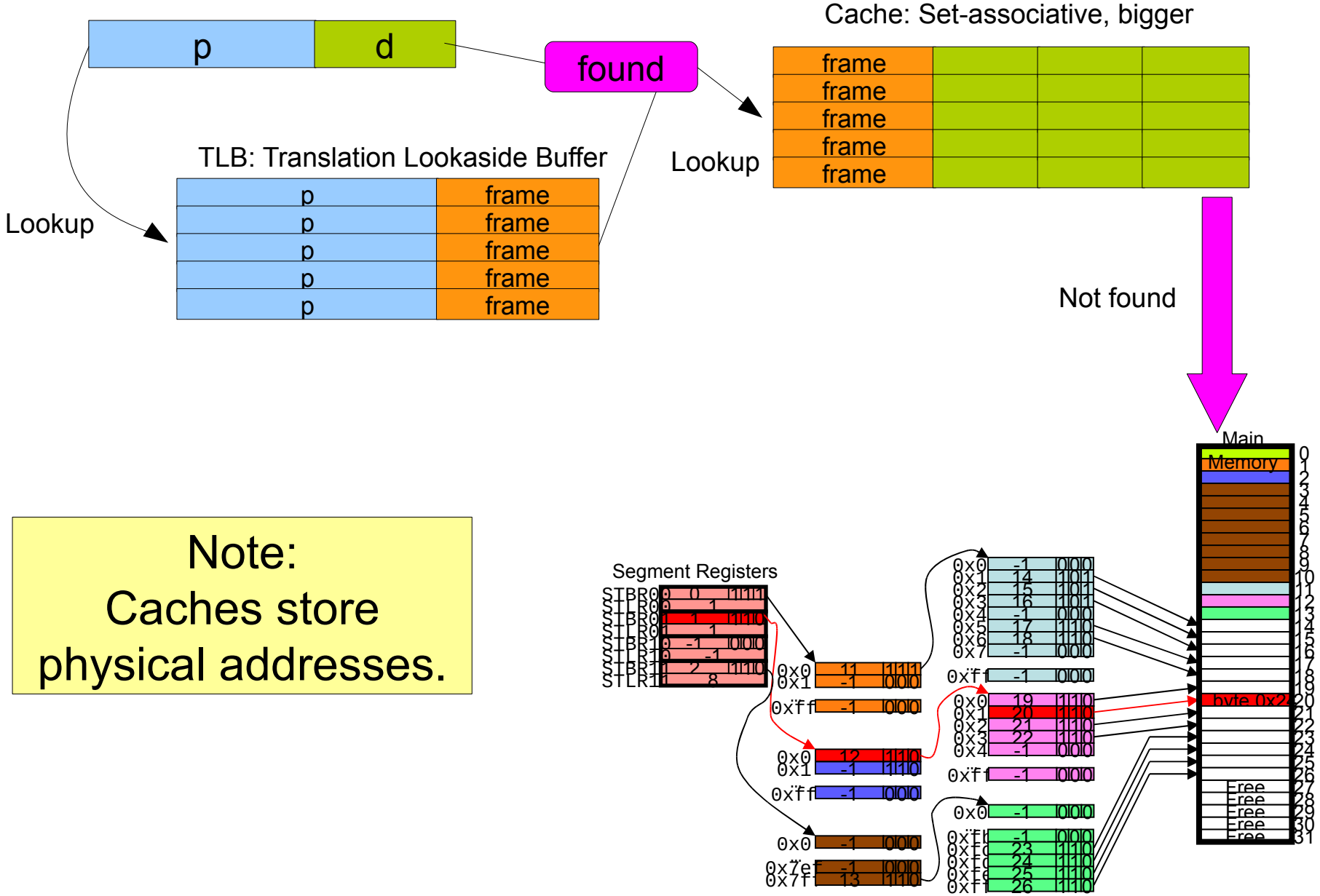


With a TLB & Cache





With a TLB & Cache



Note:
Caches store
physical addresses.