

# 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.

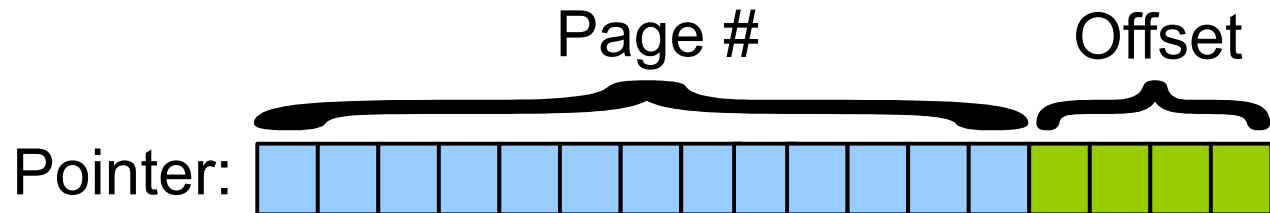
Single-level paging  
With PTBR/PTLR.

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.

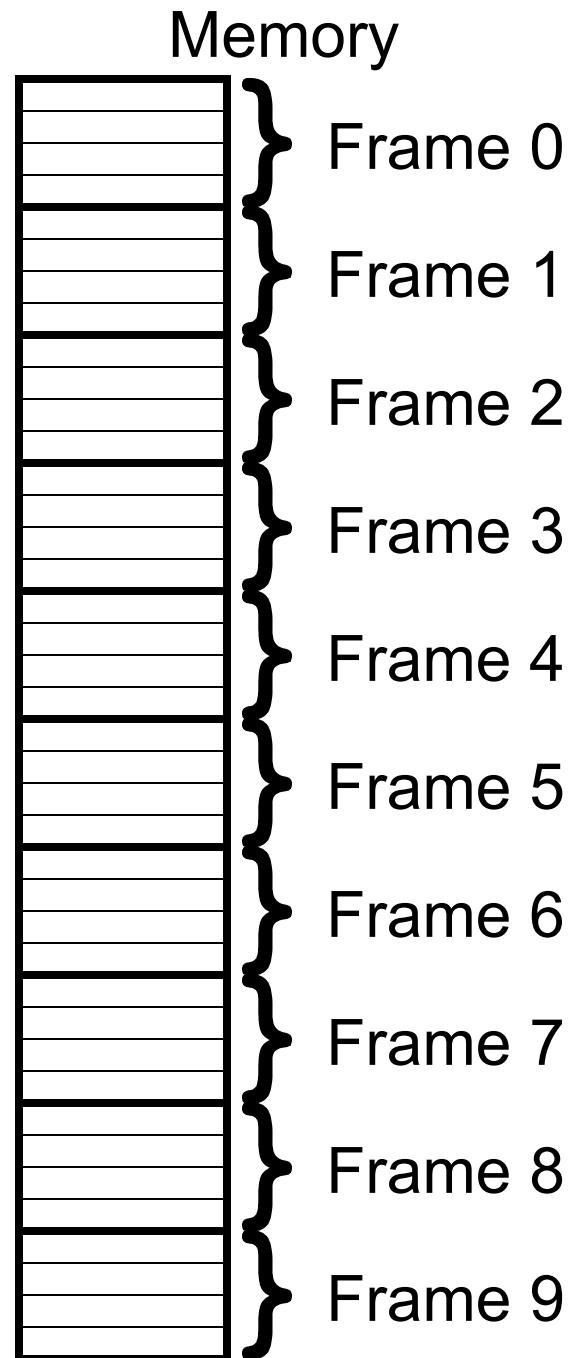
Pointers are 16 bits.

Page size = 4 words.

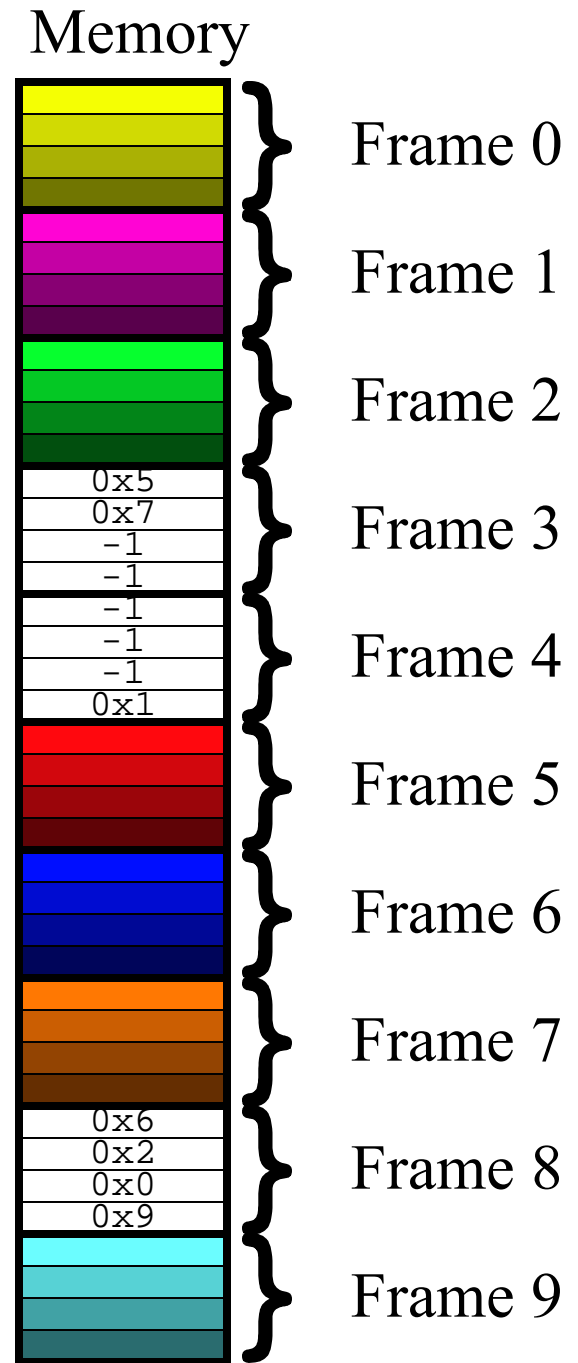
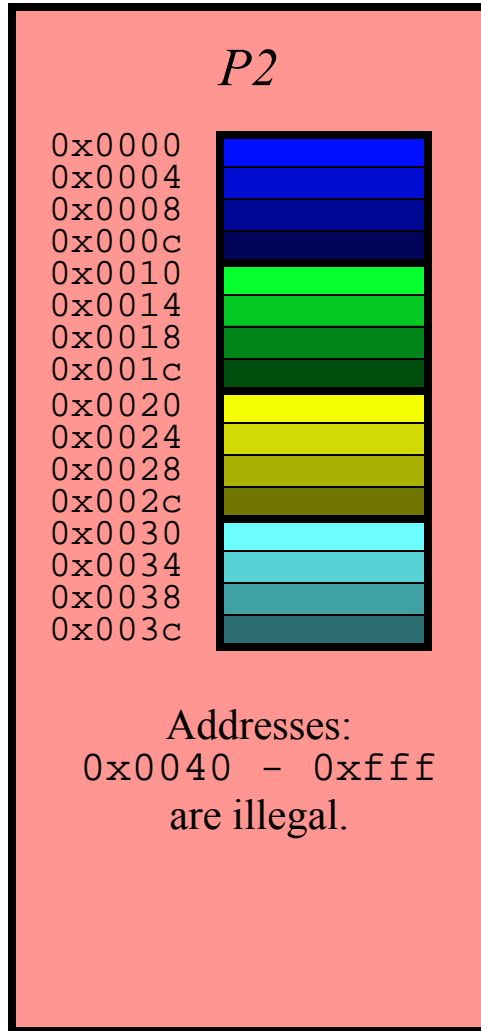
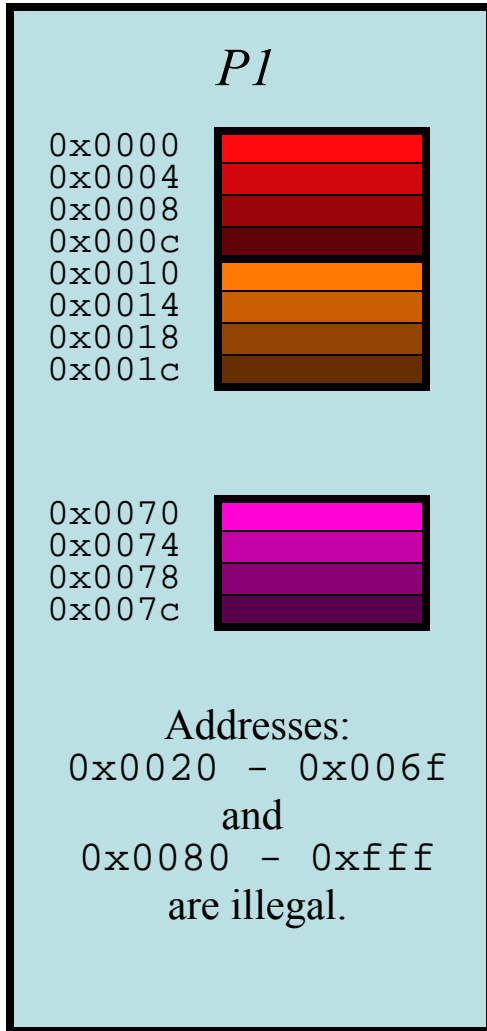
Single-level paging

With PTBR/PTLR.

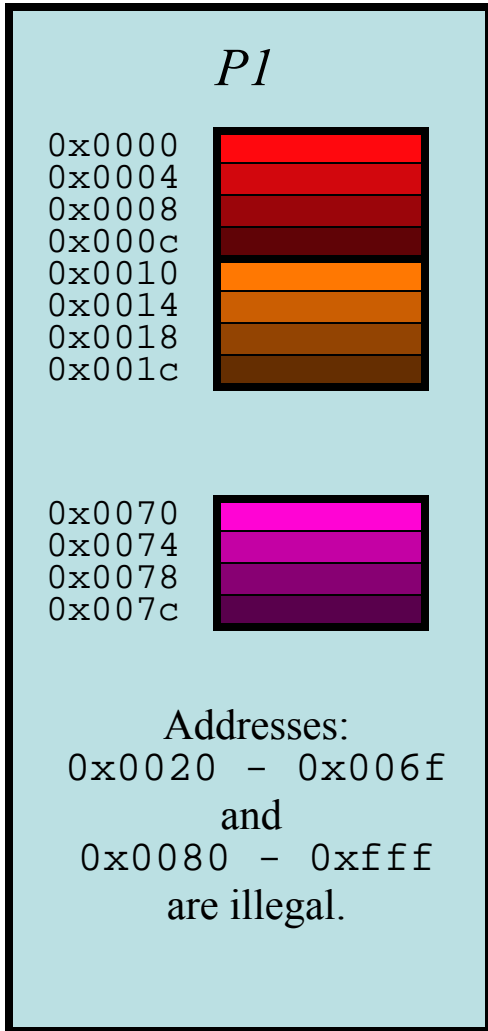
There are 10 frames  
in main memory.



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

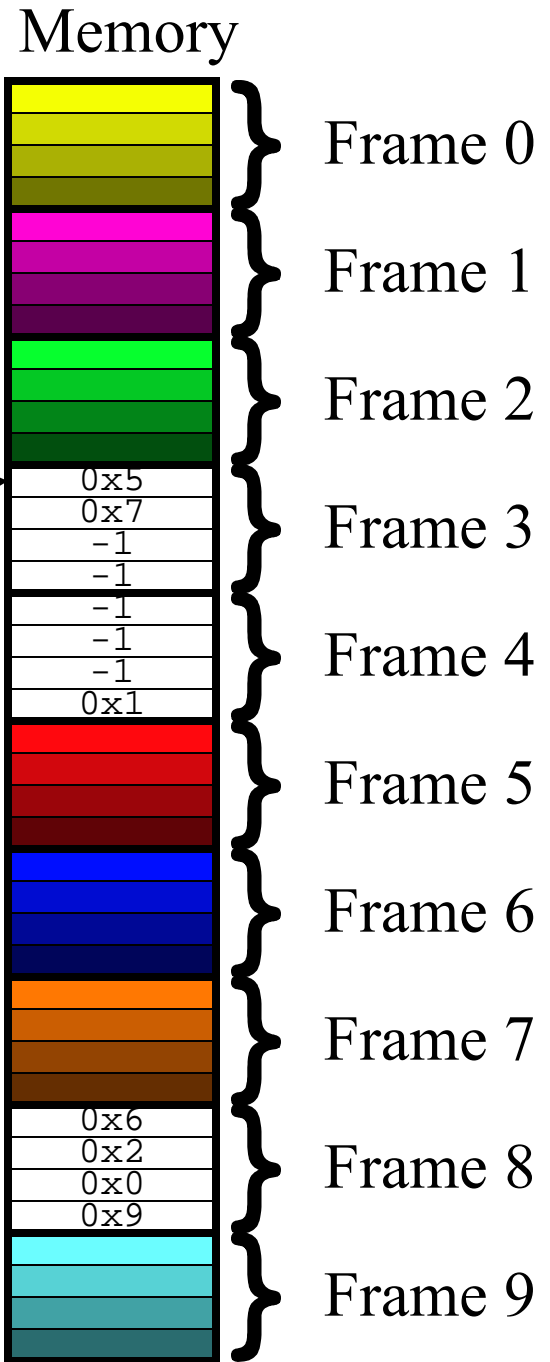


Suppose process P1 is running:

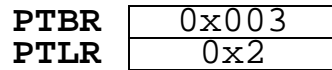
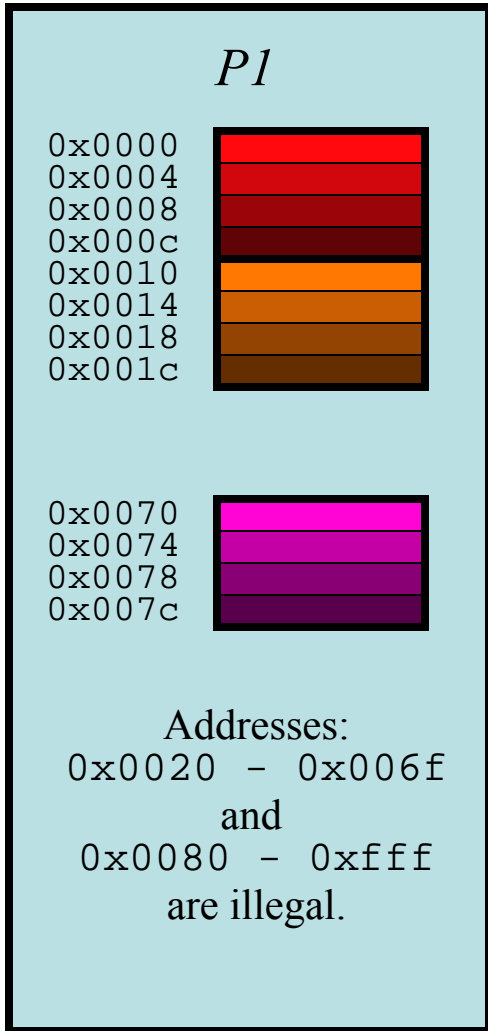


|             |       |
|-------------|-------|
| <b>PTBR</b> | 0x003 |
| <b>PTLR</b> | 0x2   |

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



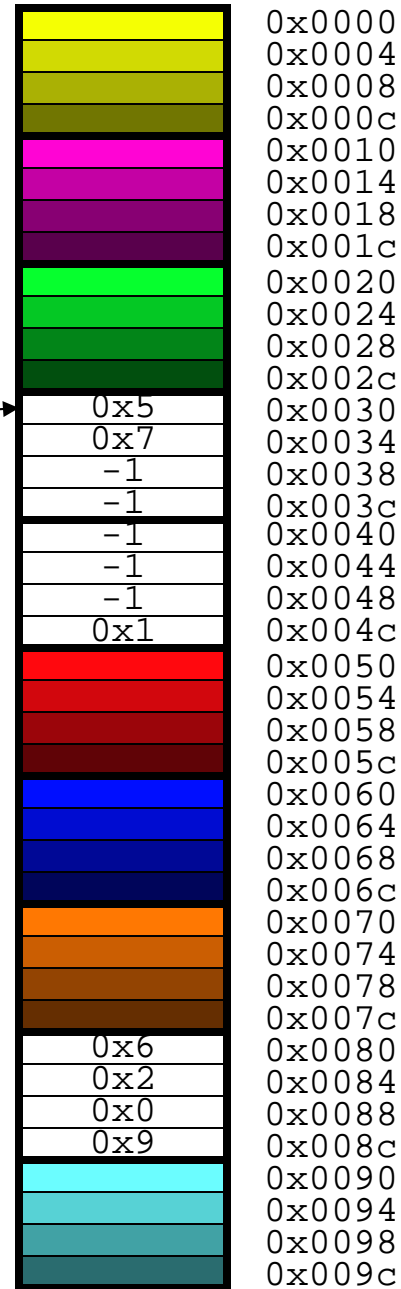
Suppose process P1 is running:



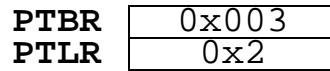
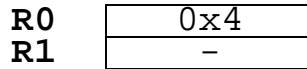
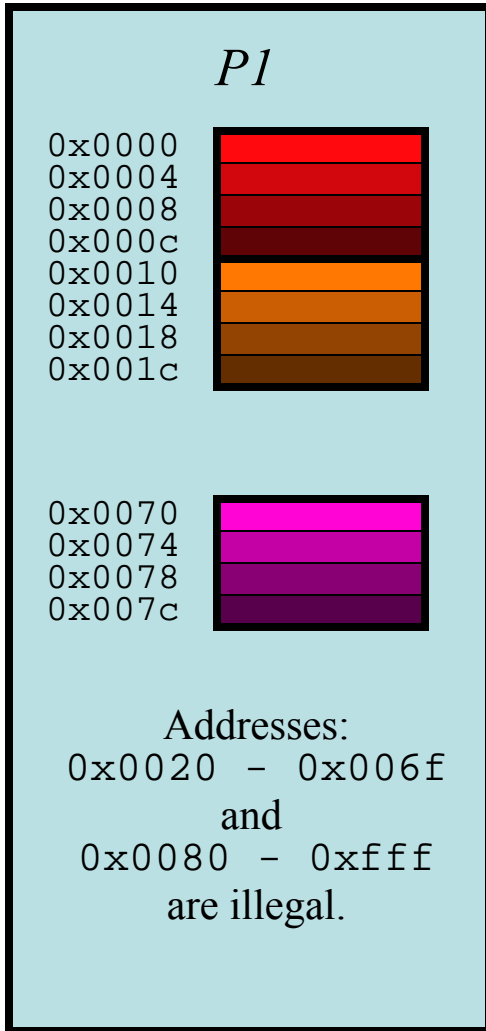
Suppose global variable *j* is at user location 0x0018, and we execute:

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

# Memory



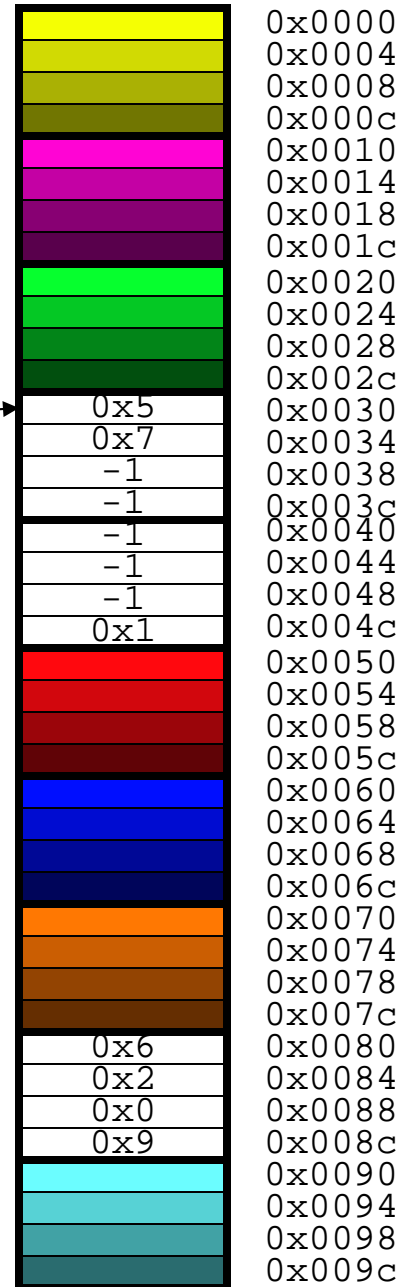
Suppose process P1 is running:



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

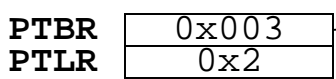
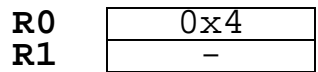
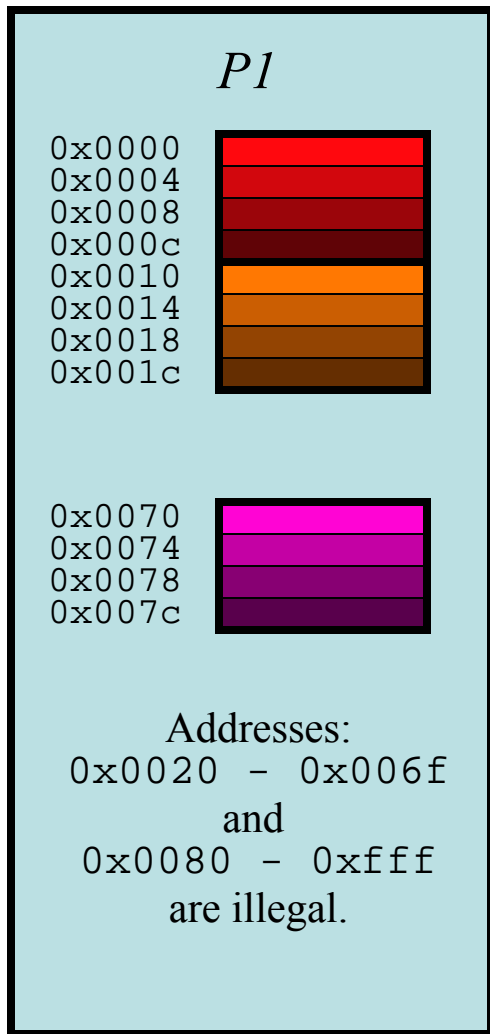
```
mv #4 -> %r0
st %r0 -> j
```

# Memory





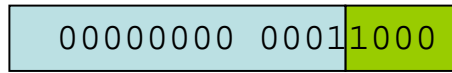
Suppose process P1 is running:



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

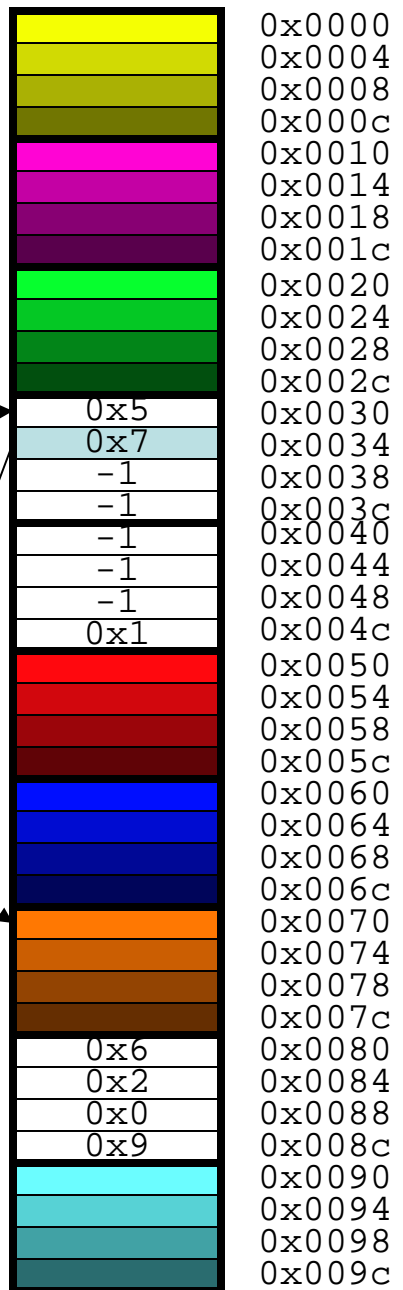
```
mv #4 -> %r0
st %r0 -> j
```

0x0018 =

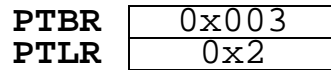
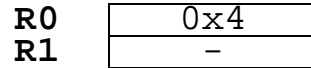
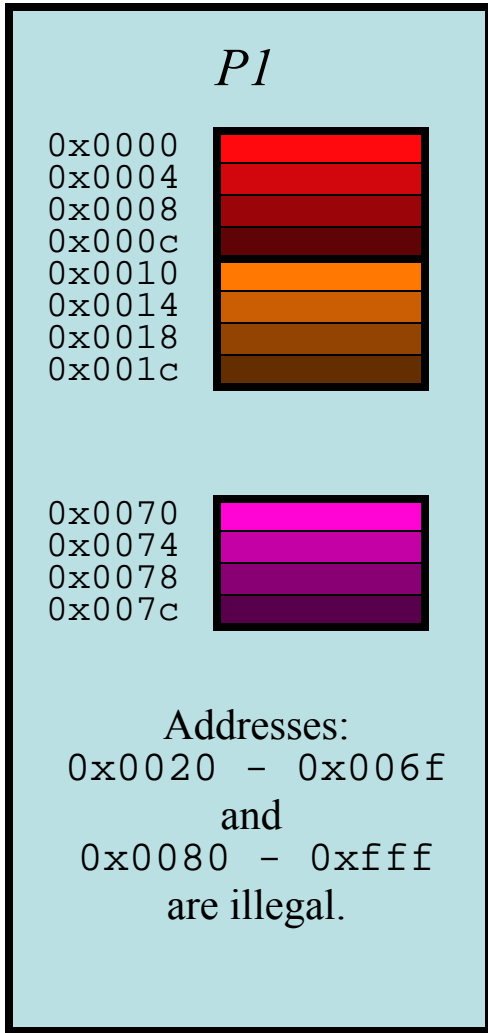


Frame 1

# Memory



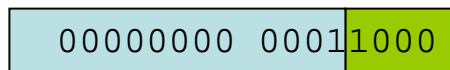
Suppose process P1 is running:



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

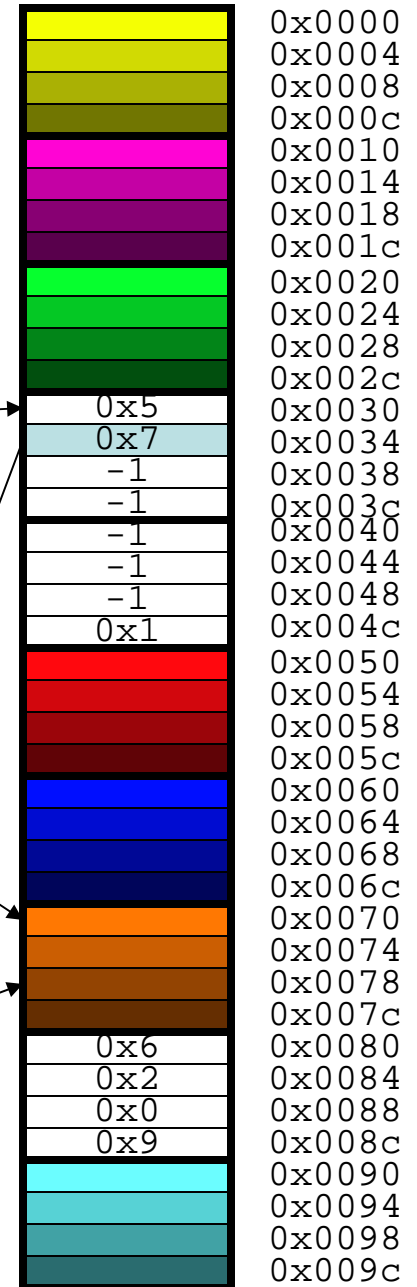
```
mv #4 -> %r0
st %r0 -> j
```

0x0018 =



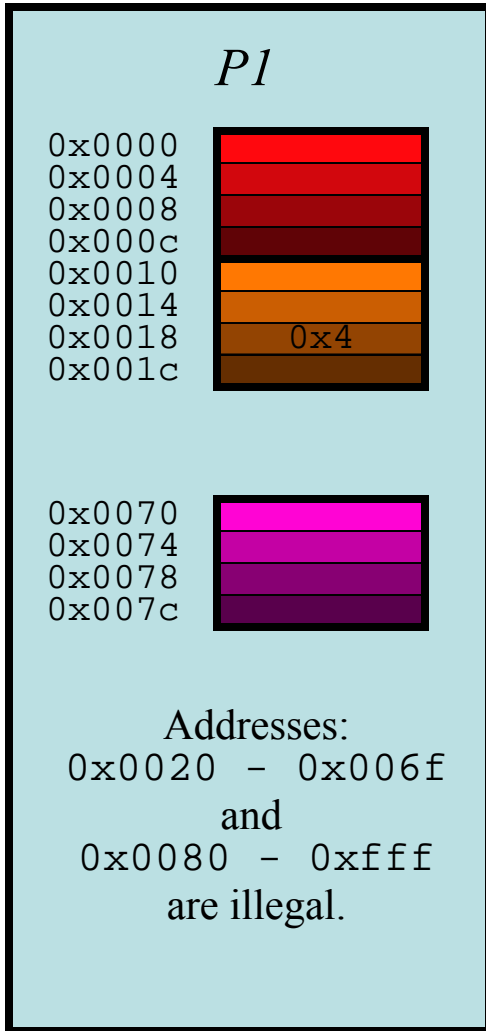
Offset 8

# Memory



Suppose process P1 is running:

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



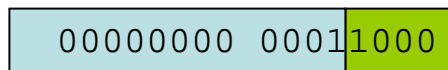
|    |     |
|----|-----|
| R0 | 0x4 |
| R1 | -   |

|      |       |
|------|-------|
| PTBR | 0x003 |
| PTLR | 0x2   |

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

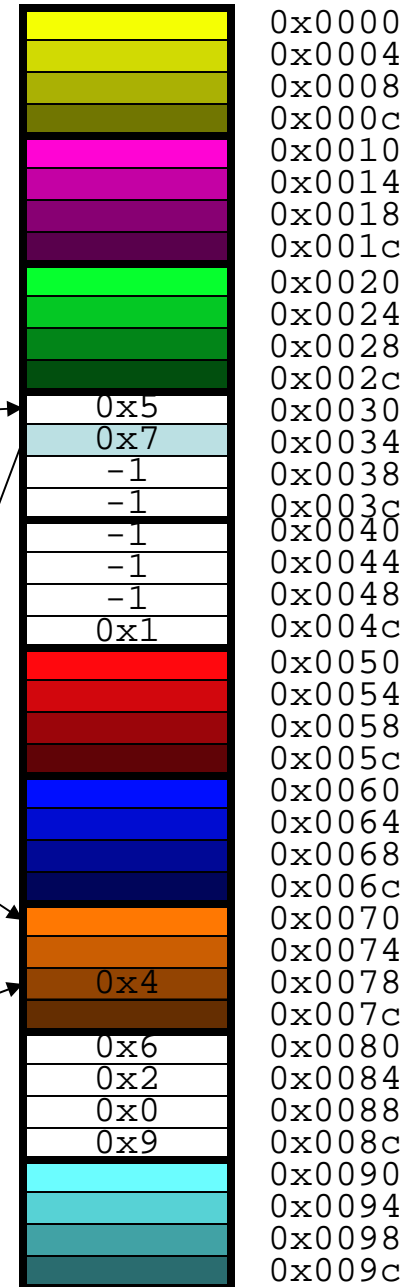
```
mv #4 -> %r0
st %r0 -> j
```

0x0018 =

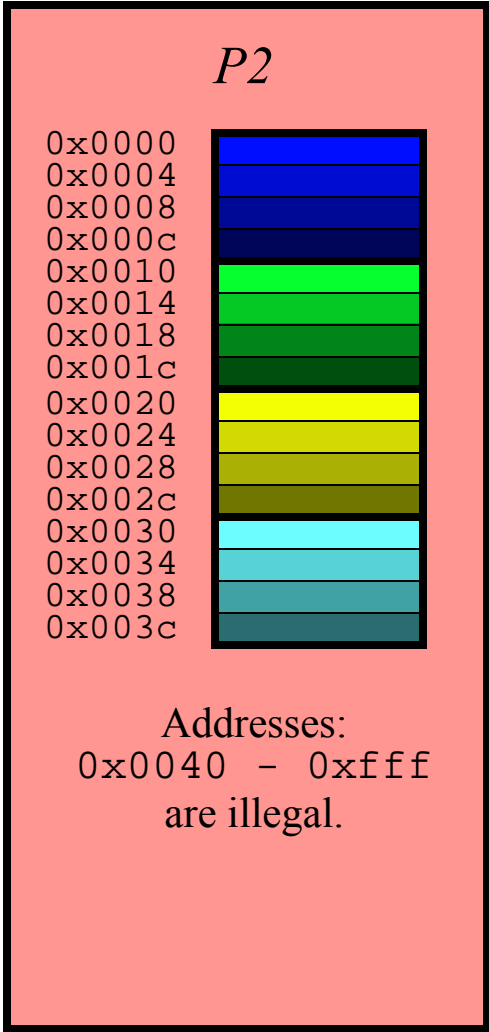


Offset 8

Memory



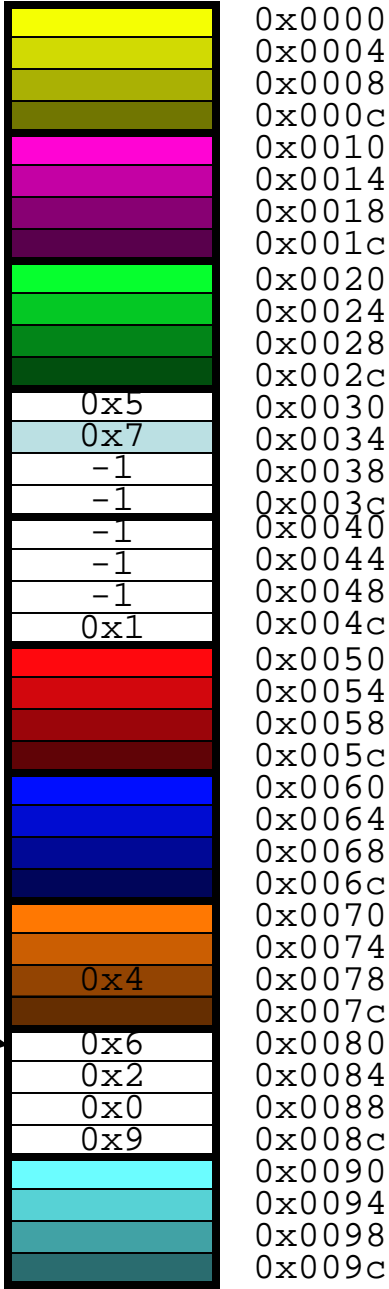
To run process *P2*, you need to switch the PTBR/PTLR:



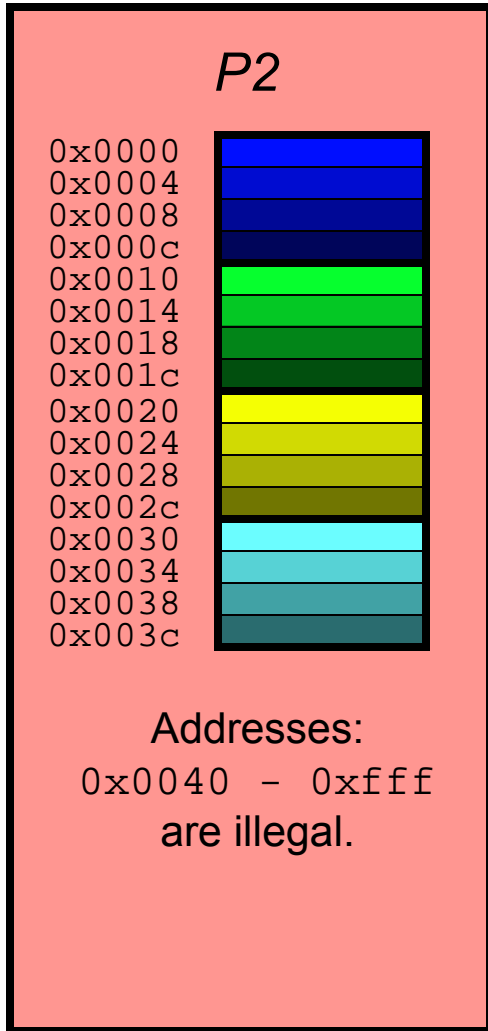
|    |     |
|----|-----|
| R0 | 0x4 |
| R1 | -   |

|      |       |
|------|-------|
| PTBR | 0x008 |
| PTLR | 0x1   |

# Memory



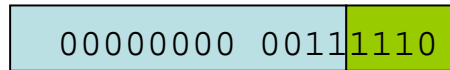
# What is user address 0x003e?



|    |     |
|----|-----|
| R0 | 0x4 |
| R1 | -   |

|      |       |
|------|-------|
| PTBR | 0x008 |
| PTLR | 0x1   |

0x003e =

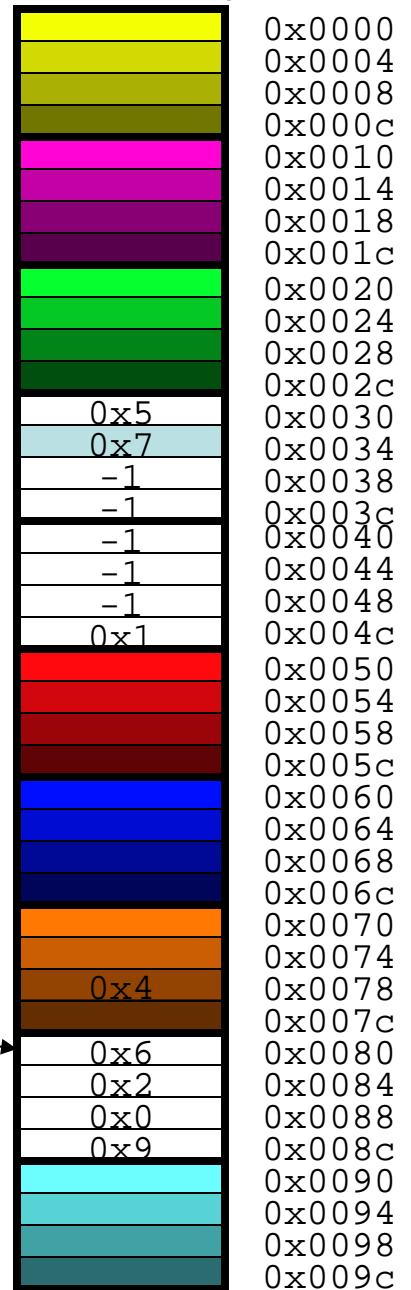


Page 0x3 =  
Frame 0x9

Offset e

Physical address 0x009e.

# Memory

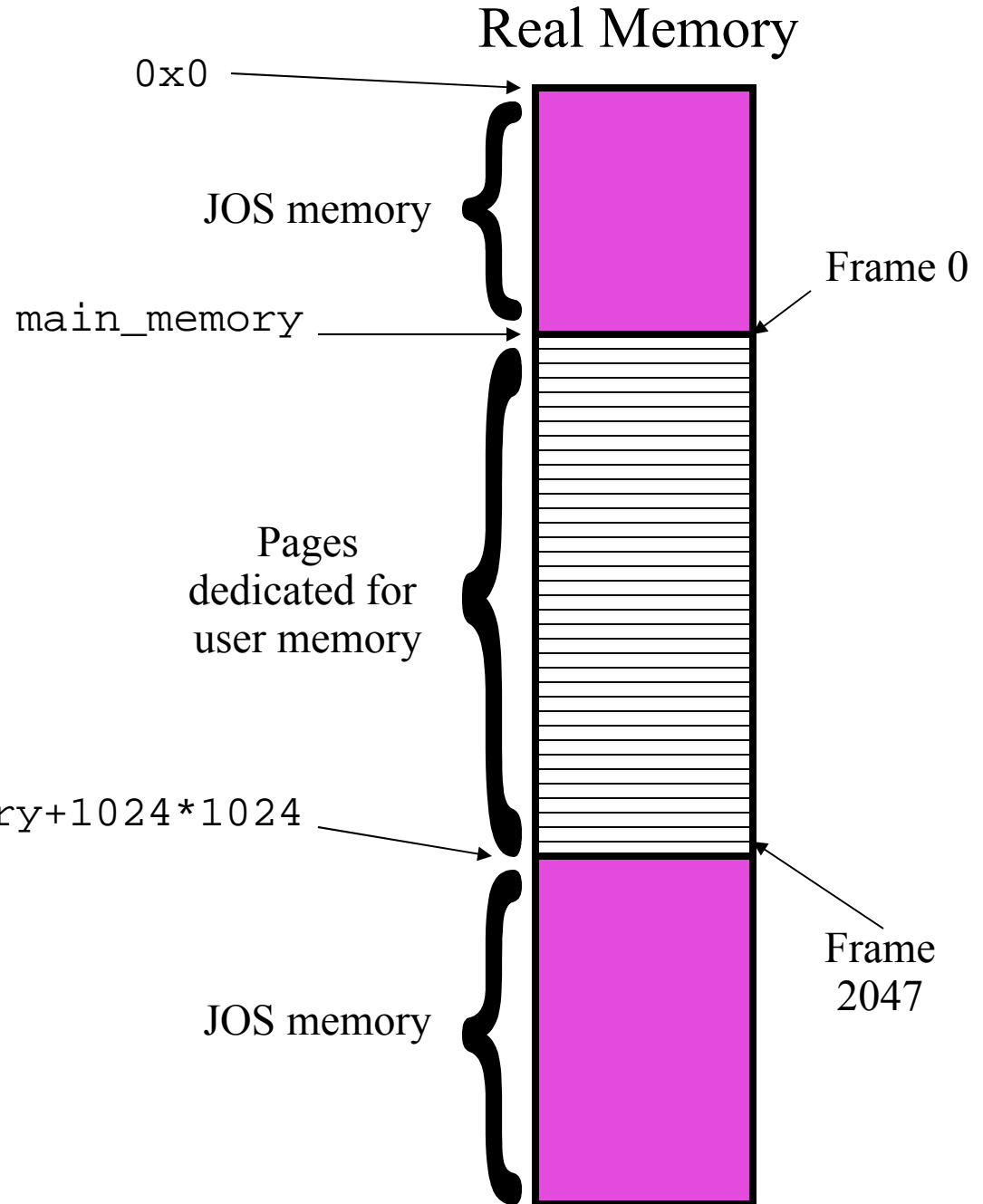


# VM in JOS:

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

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

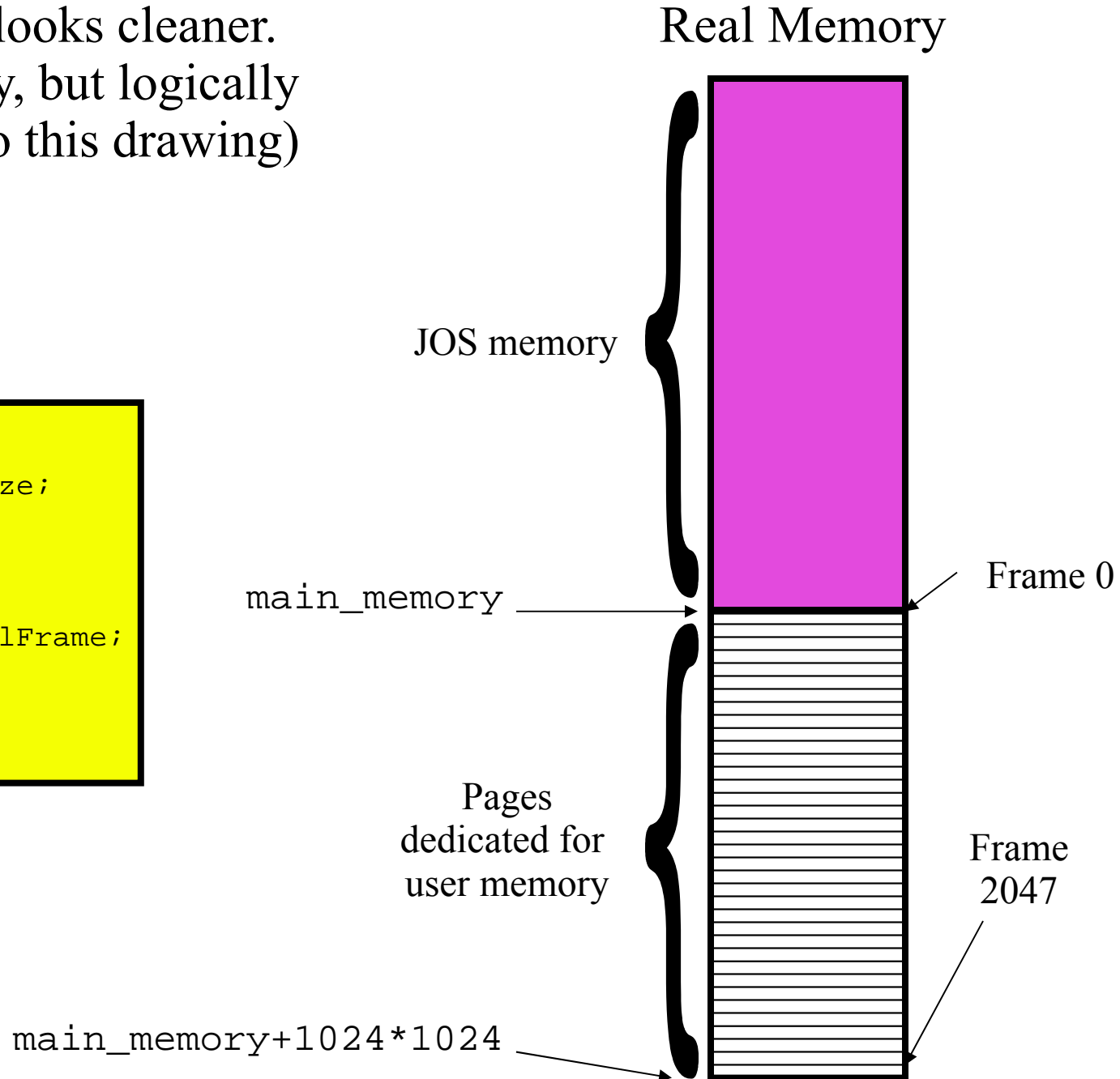
PageSize = 512.  
So there are 2K pages  
dedicated for users.



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

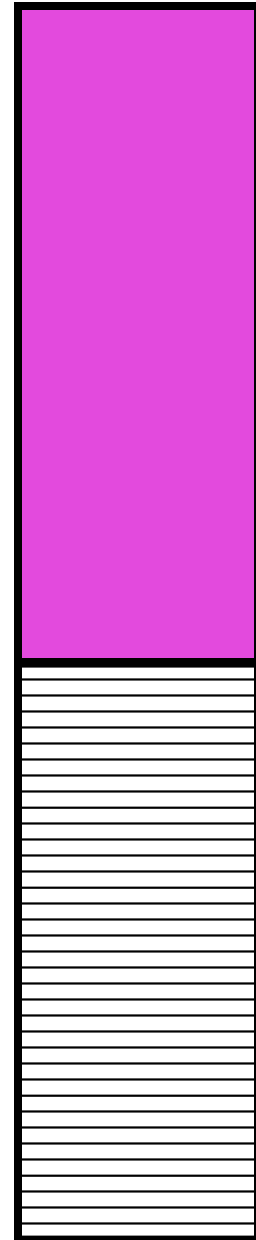
0x2000-0x234c: Globals

0x7ff000-0x7fffffff: Stack

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

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

Real Memory



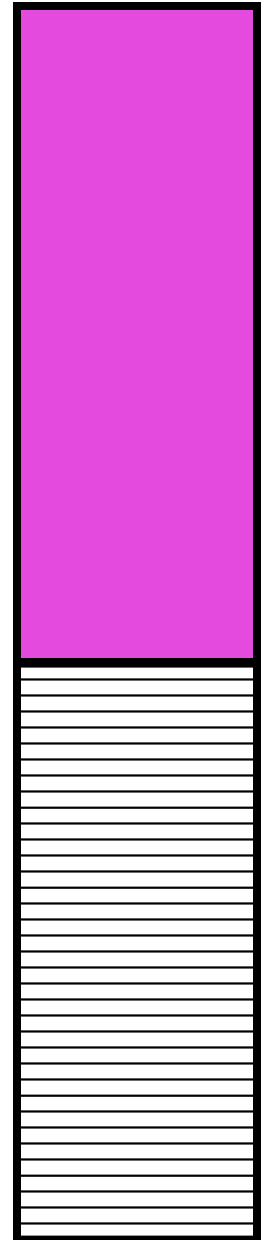


Now, suppose you have a process whose address space is as follows:

|                            |   |                   |
|----------------------------|---|-------------------|
| 0x0800-0x1fff: Code        | — | 12 pages: SP 4    |
| 0x2000-0x234c: Globals     | — | 2 pages: SP 16    |
| 0x7ff000-0x7fffffff: Stack | — | 8 pages: SP 16376 |

(sp = starting page)

Real Memory

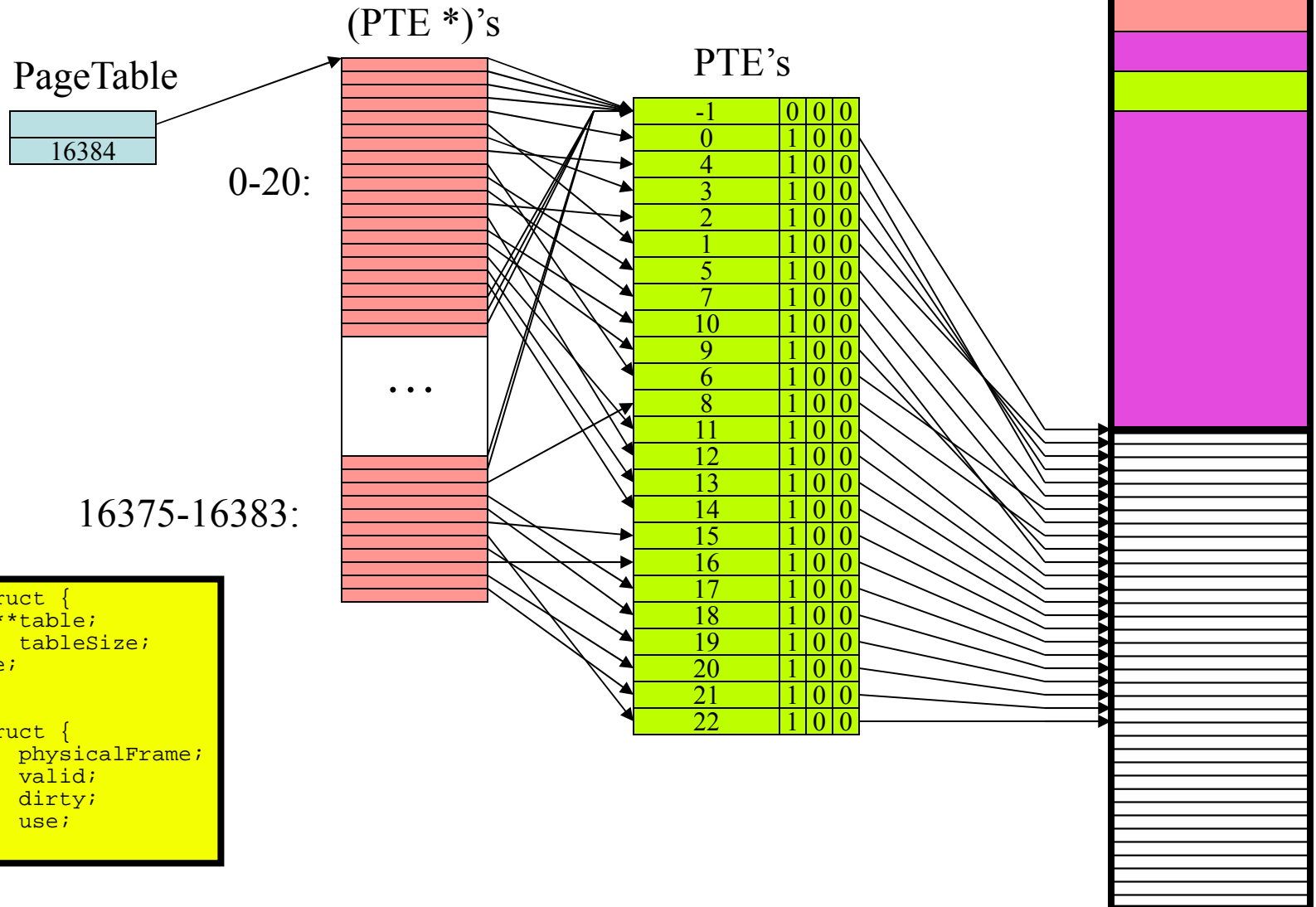


```
typedef struct {
    PTE    **table;
    int     tableSize;
} PageTable;

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

Real Memory



## 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 | - | 0x7ffffeff | - | NULL    | - | Pages 0x80004 - 0x1ffffb  |
| 0x7ffff000 | - | 0x7fffffff | - | Stack   | - | Pages 0x1ffffc - 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 | - | 0x7ffffeff | - | NULL    | - | Pages 0x80004 - 0x1ffffb  |
| 0x7ffff000 | - | 0x7fffffff | - | Stack   | - | Pages 0x1ffffc - 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 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

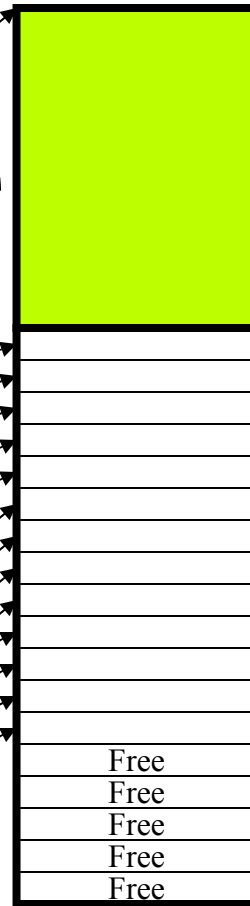
Main  
Memory

Pages  
0 - 8191

|      |      |
|------|------|
| PTBR | 0    |
| PTLR | 8192 |

0x2000 = 8192 Pages

|           |      |   |   |   |
|-----------|------|---|---|---|
| 0x0       | -1   | 0 | 0 | 0 |
| 0x1       | 8192 | 1 | 0 | 1 |
| 0x2       | 8193 | 1 | 0 | 1 |
| 0x3       | 8194 | 1 | 0 | 1 |
| 0x4       | -1   | 0 | 0 | 0 |
| 0x5       | 8195 | 1 | 1 | 0 |
| 0x6       | 8196 | 1 | 1 | 0 |
| 0x7       | -1   | 0 | 0 | 0 |
| ...       |      |   |   |   |
| 0x7ffff   | -1   | 0 | 0 | 0 |
| 0x80000   | 8197 | 1 | 1 | 0 |
| 0x80001   | 8198 | 1 | 1 | 0 |
| 0x80002   | 8199 | 1 | 1 | 0 |
| 0x80003   | 8200 | 1 | 1 | 0 |
| 0x80004   | -1   | 0 | 0 | 0 |
| ...       |      |   |   |   |
| 0x1fffffb | -1   | 0 | 0 | 0 |
| 0x1fffffc | 8201 | 1 | 1 | 0 |
| 0x1fffffd | 8202 | 1 | 1 | 0 |
| 0x1fffffe | 8203 | 1 | 1 | 0 |
| 0x1fffff  | 8204 | 1 | 1 | 0 |



8192  
8193  
8194  
8195  
8196  
8197  
8198  
8199  
8200  
8201  
8202  
8203  
8204  
8205  
8206  
8207  
8208  
8209

Free  
Free  
Free  
Free



# 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 | - 0x00001bfff | - Globals | - Pages 0x5 - 0x6            |
| 0x00001c00 | - 0x1fffffff  | - NULL    | - Pages 0x7 - 0x7ffff        |
| 0x20000000 | - 0x20000fff  | - Heap    | - Pages 0x80000 - 0x80003    |
| 0x20001000 | - 0x7ffffeff  | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff  | - Stack   | - Pages 0x1fffffc - 0x1fffff |

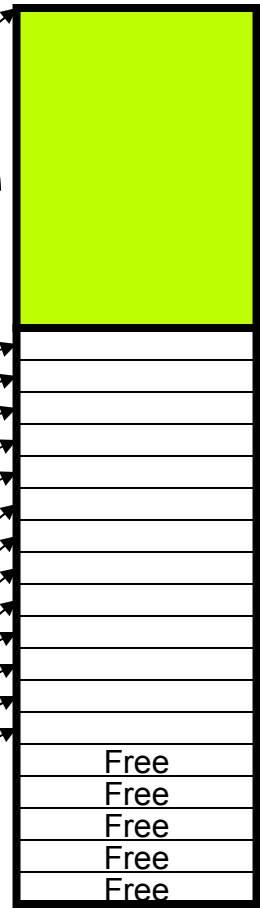
Main  
Memory

Pages  
0 - 8191

|      |      |
|------|------|
| PTBR | 0    |
| PTLR | 8192 |

0x2000 = 8192 Pages

|           |      |   |   |   |
|-----------|------|---|---|---|
| 0x0       | -1   | 0 | 0 | 0 |
| 0x1       | 8192 | 1 | 0 | 1 |
| 0x2       | 8193 | 1 | 0 | 1 |
| 0x3       | 8194 | 1 | 0 | 1 |
| 0x4       | -1   | 0 | 0 | 0 |
| 0x5       | 8195 | 1 | 1 | 0 |
| 0x6       | 8196 | 1 | 1 | 0 |
| 0x7       | -1   | 0 | 0 | 0 |
| ...       |      |   |   |   |
| 0x7ffff   | -1   | 0 | 0 | 0 |
| 0x80000   | 8197 | 1 | 1 | 0 |
| 0x80001   | 8198 | 1 | 1 | 0 |
| 0x80002   | 8199 | 1 | 1 | 0 |
| 0x80003   | 8200 | 1 | 1 | 0 |
| 0x80004   | -1   | 0 | 0 | 0 |
| ...       |      |   |   |   |
| 0x1fffffb | -1   | 0 | 0 | 0 |
| 0x1fffffc | 8201 | 1 | 1 | 0 |
| 0x1fffffd | 8202 | 1 | 1 | 0 |
| 0x1fffffe | 8203 | 1 | 1 | 0 |
| 0x1fffff  | 8204 | 1 | 1 | 0 |



8192  
8193  
8194  
8195  
8196  
8197  
8198  
8199  
8200  
8201  
8202  
8203  
8204  
Free 8205  
Free 8206  
Free 8207  
Free 8208  
Free 8209

# 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 0x2000424, which is in the second page of the heap.

0x2000424 = 0010 0000 0000 0000 0000 0100 0010 0100  
= 001000000000000000000001 0000100100  
= 0x80001 0x24

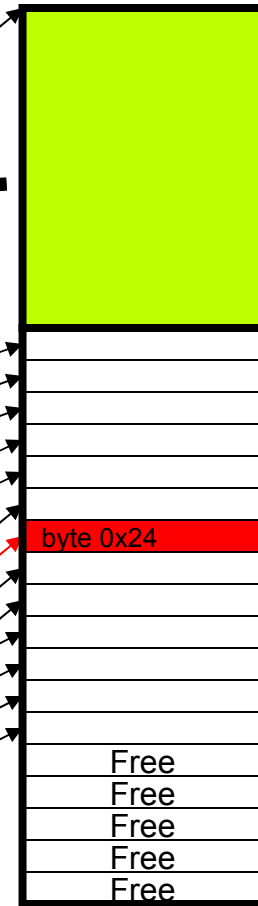
## Main Memory

Pages  
0 - 8191

PTBR 0  
PTLR 8192

0x2000 = 8192 Pages

|           |      |       |
|-----------|------|-------|
| 0x0       | -1   | 0 0 0 |
| 0x1       | 8192 | 1 0 1 |
| 0x2       | 8193 | 1 0 1 |
| 0x3       | 8194 | 1 0 1 |
| 0x4       | -1   | 0 0 0 |
| 0x5       | 8195 | 1 1 0 |
| 0x6       | 8196 | 1 1 0 |
| 0x7       | -1   | 0 0 0 |
| ...       |      |       |
| 0x7ffff   | -1   | 0 0 0 |
| 0x80000   | 8197 | 1 1 0 |
| 0x80001   | 8198 | 1 1 0 |
| 0x80002   | 8199 | 1 1 0 |
| 0x80003   | 8200 | 1 1 0 |
| 0x80004   | -1   | 0 0 0 |
| ...       |      |       |
| 0x1fffffb | -1   | 0 0 0 |
| 0x1fffffc | 8201 | 1 1 0 |
| 0x1fffffd | 8202 | 1 1 0 |
| 0x1fffffe | 8203 | 1 1 0 |
| 0x1ffffff | 8204 | 1 1 0 |



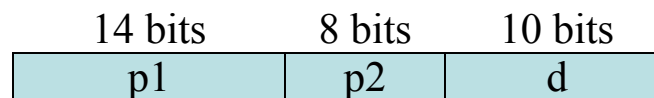
8192  
8193  
8194  
8195  
8196  
8197  
8198  
8199  
8200  
8201  
8202  
8203  
8204  
8205  
8206  
8207  
8208  
8209

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  | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7fffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

# Two-Level Page Table

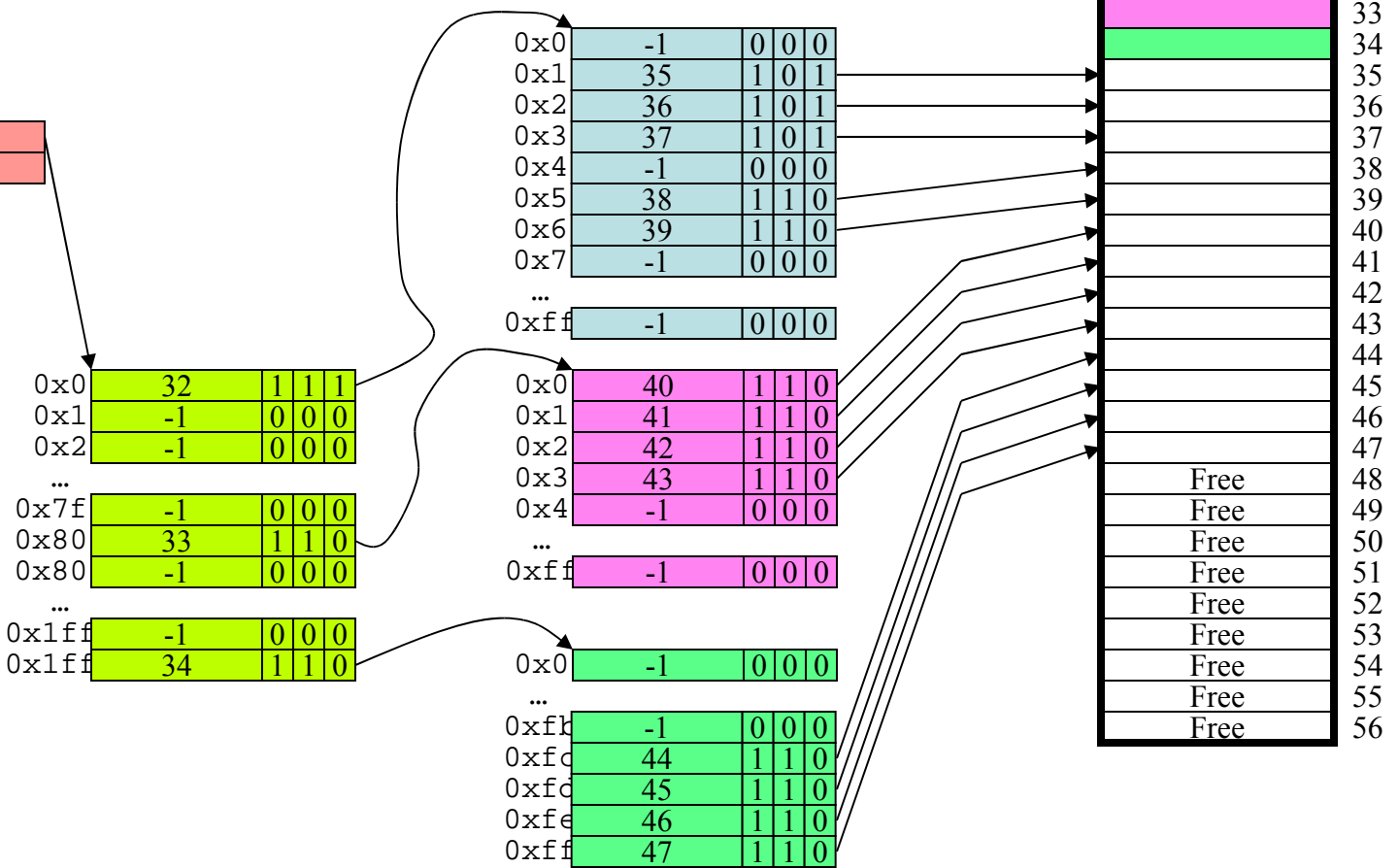
|            |              |           |                              |
|------------|--------------|-----------|------------------------------|
| 0x00000000 | - 0x000003ff | - NULL    | - Page: 0x0                  |
| 0x00000400 | - 0x00000fff | - Code    | - Pages 0x1 - 0x3            |
| 0x00001000 | - 0x000013ff | - NULL    | - Page: 0x4                  |
| 0x00001400 | - 0x00001bff | - Globals | - Pages 0x5 - 0x6            |
| 0x00001c00 | - 0x1fffffff | - NULL    | - Pages 0x7 - 0x7ffff        |
| 0x20000000 | - 0x20000fff | - Heap    | - Pages 0x80000 - 0x80003    |
| 0x20001000 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

## Main Memory

Pages  
0 - 31

PTBR 0  
PTLR 32

8192/256 = 32 Pages



# Two-Level Page Table

Memory usage: 48 KB: Better!

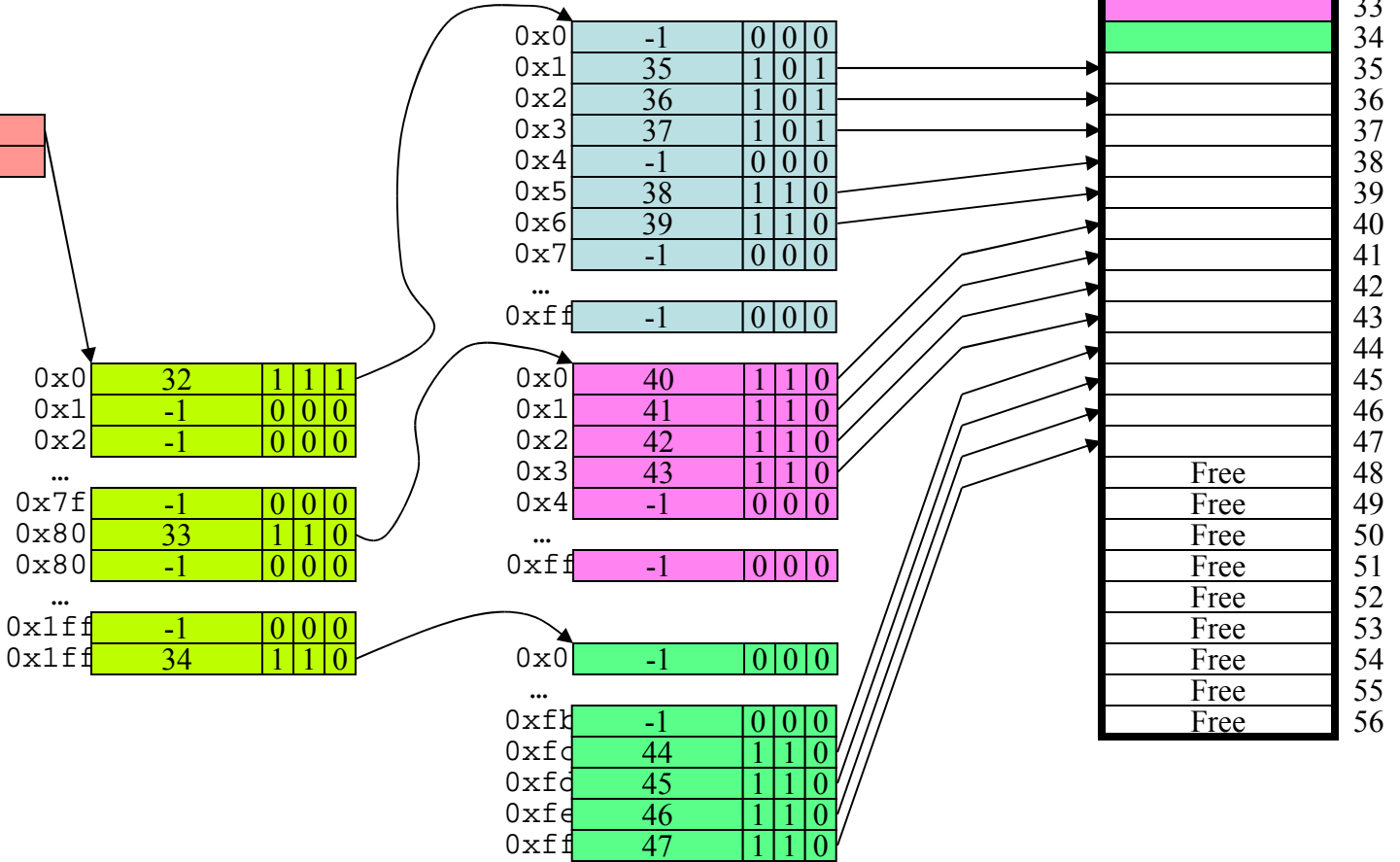
|            |              |           |                              |
|------------|--------------|-----------|------------------------------|
| 0x00000000 | - 0x000003ff | - NULL    | - Page: 0x0                  |
| 0x00000400 | - 0x00000fff | - Code    | - Pages 0x1 - 0x3            |
| 0x00001000 | - 0x000013ff | - NULL    | - Page: 0x4                  |
| 0x00001400 | - 0x00001bff | - Globals | - Pages 0x5 - 0x6            |
| 0x00001c00 | - 0x1fffffff | - NULL    | - Pages 0x7 - 0x7ffff        |
| 0x20000000 | - 0x20000fff | - Heap    | - Pages 0x80000 - 0x80003    |
| 0x20001000 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

## Main Memory

Pages 0 - 31

PTBR 0  
PTLR 32

8192/256 = 32 Pages



# Two-Level Page Table

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

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

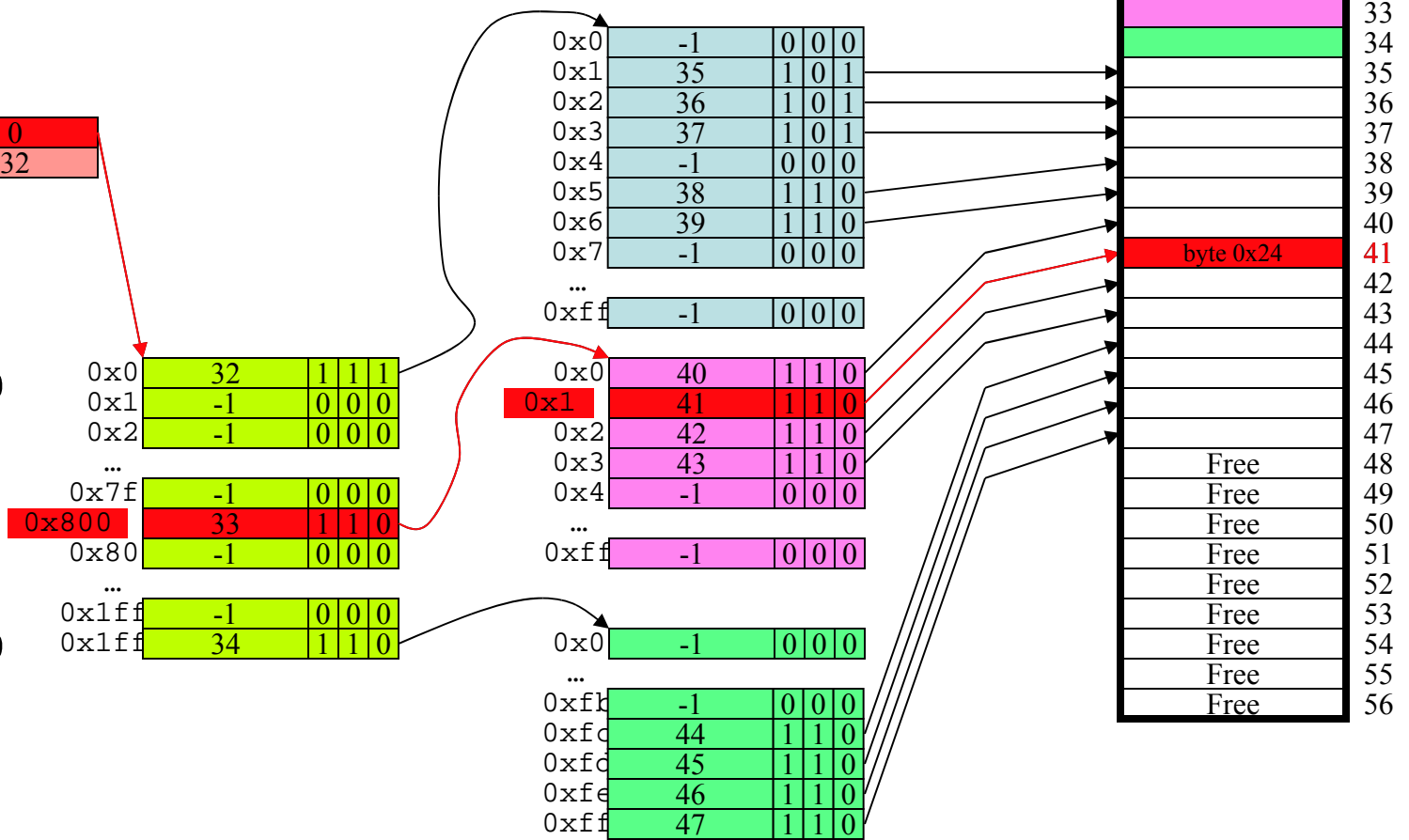
0x20000424 = 0010 0000 0000 0000 0000 0000 0100 0010 0100  
 = 0010000000000000 00000001 0000100100  
 = 0x800 0x1 0x24

## Main Memory

Pages 0 - 31

PTBR 0  
 PTLR 32

8192/256 = 32 Pages

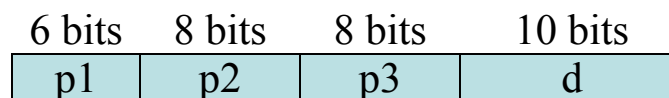


New Example - Our machine has:

- 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

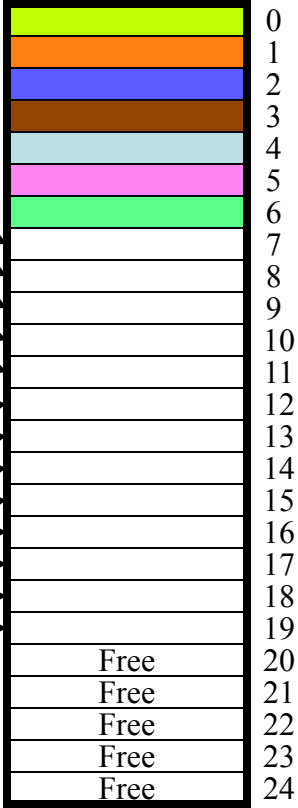
There are  $2^5 = 32$  first-level PTE's.

|            |   |            |   |         |   |                           |
|------------|---|------------|---|---------|---|---------------------------|
| 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 | - | 0x7ffffeff | - | NULL    | - | Pages 0x80004 - 0x1ffffb  |
| 0x7ffff000 | - | 0x7fffffff | - | Stack   | - | Pages 0x1ffffc - 0x1fffff |

# Three-Level Page Table

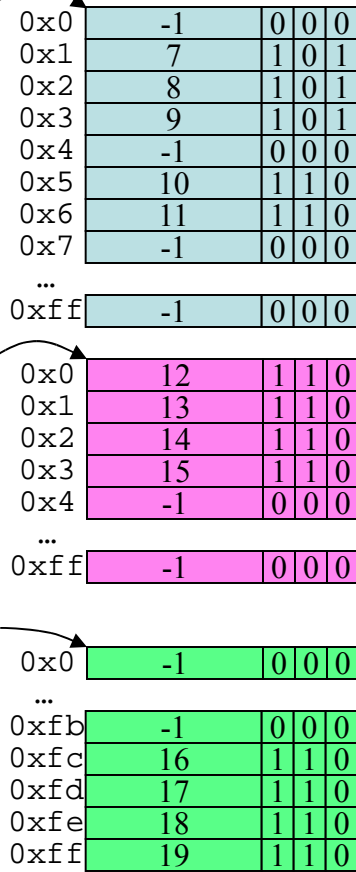
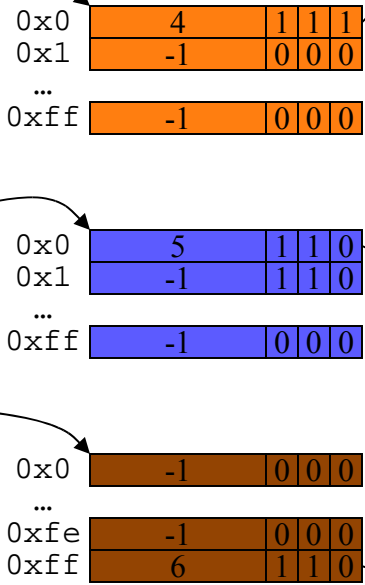
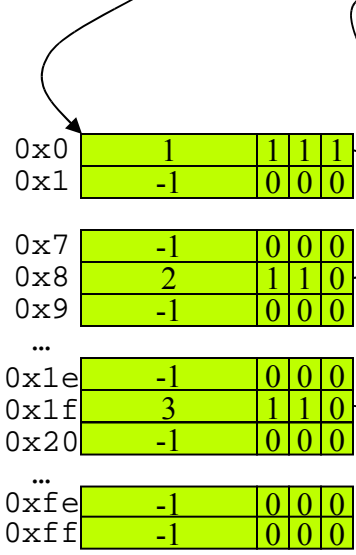
Memory usage: 20 KB: Excellent!

## Main 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 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

|      |   |
|------|---|
| PTBR | 0 |
| PTLR | 1 |





# Three-Level Page Table

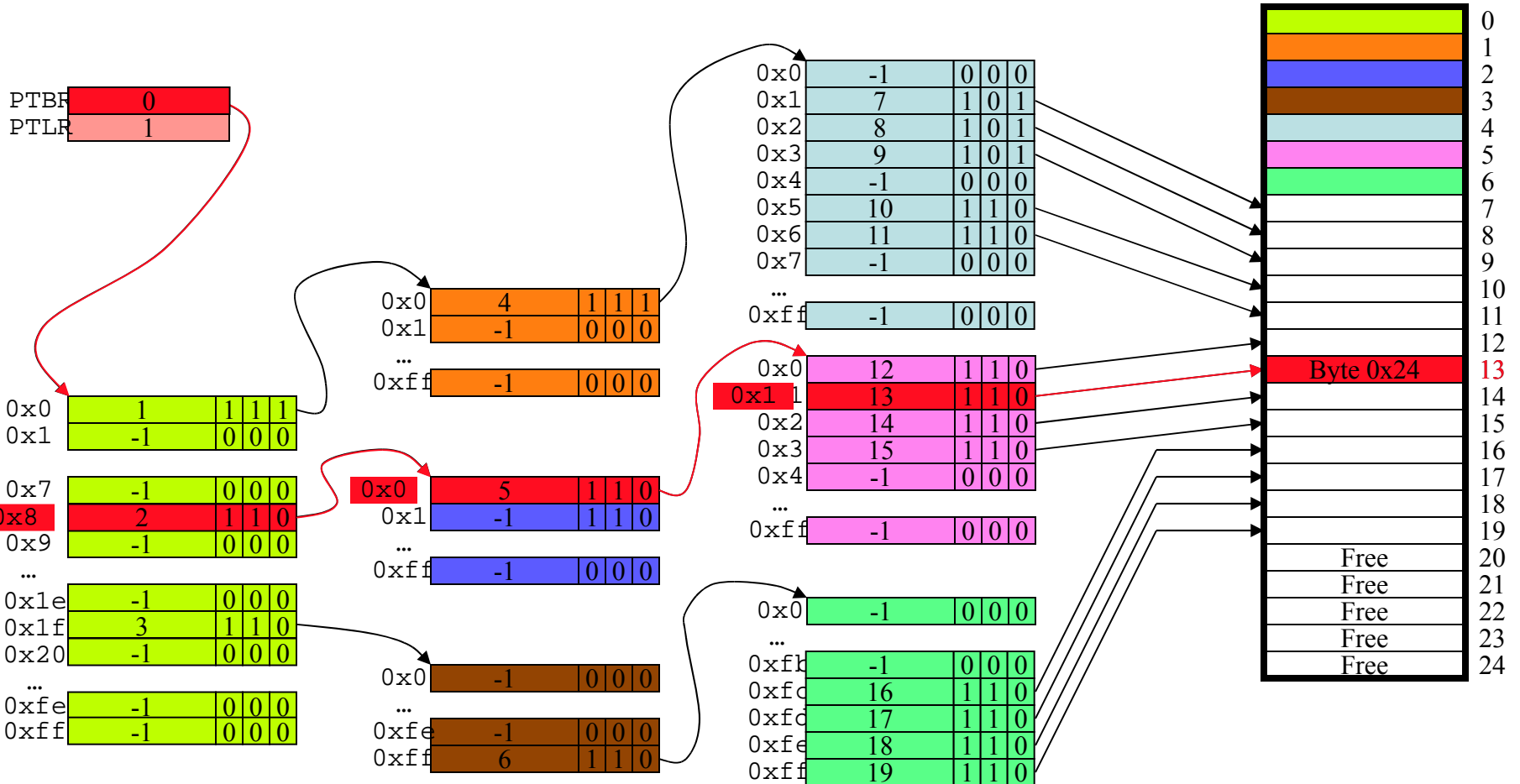
Physical page 13, offset 0x24:

0x0d 0x24 = .. 0000 0000 1101 00 0010 0100 =  
 0000 0000 0000 0000 0011 0100 0010 0100 =  
 0x00003424

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

## Main Memory

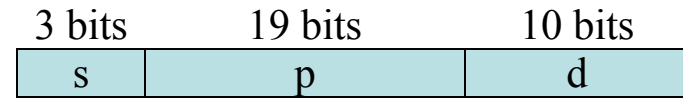


New Example - Our machine has:

- 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.

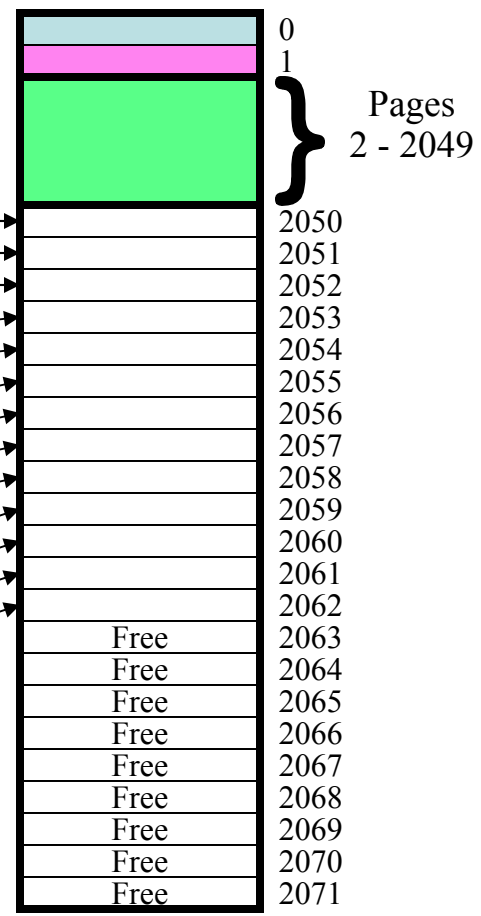
|            |              |           |                              |
|------------|--------------|-----------|------------------------------|
| 0x00000000 | - 0x000003ff | - NULL    | - Page: 0x0                  |
| 0x00000400 | - 0x00000fff | - Code    | - Pages 0x1 - 0x3            |
| 0x00001000 | - 0x000013ff | - NULL    | - Page: 0x4                  |
| 0x00001400 | - 0x00001bff | - Globals | - Pages 0x5 - 0x6            |
| 0x00001c00 | - 0x1fffffff | - NULL    | - Pages 0x7 - 0x7ffff        |
| 0x20000000 | - 0x20000fff | - Heap    | - Pages 0x80000 - 0x80003    |
| 0x20001000 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 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 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

## Main Memory



### Segment Registers

|        |      |      |
|--------|------|------|
| STBR00 | 0    | 1111 |
| STLR00 | 1    |      |
| STBR01 | 1    | 1110 |
| STLR01 | 1    |      |
| STBR10 | -1   | 0000 |
| STLR10 | -1   |      |
| STBR11 | 2    | 1110 |
| STLR11 | 2048 |      |

|          |      |      |
|----------|------|------|
| 0x0      | -1   | 0000 |
| 0x1      | 2050 | 1001 |
| 0x2      | 2051 | 1001 |
| 0x3      | 2052 | 1001 |
| 0x4      | -1   | 0000 |
| 0x5      | 2053 | 1110 |
| 0x6      | 2054 | 1110 |
| 0x7      | -1   | 0000 |
| ...      |      |      |
| 0xff     | -1   | 0000 |
| ...      |      |      |
| 0x0      | 2055 | 1110 |
| 0x1      | 2056 | 1110 |
| 0x2      | 2057 | 1110 |
| 0x3      | 2058 | 1110 |
| 0x4      | -1   | 0000 |
| ...      |      |      |
| 0xff     | -1   | 0000 |
| ...      |      |      |
| 0x0      | -1   | 0000 |
| ...      |      |      |
| 0x7ffffb | -1   | 0000 |
| 0x7ffffc | 2059 | 1110 |
| 0x7ffffd | 2060 | 1110 |
| 0x7ffffe | 2061 | 1110 |
| 0x7fffff | 2062 | 1110 |

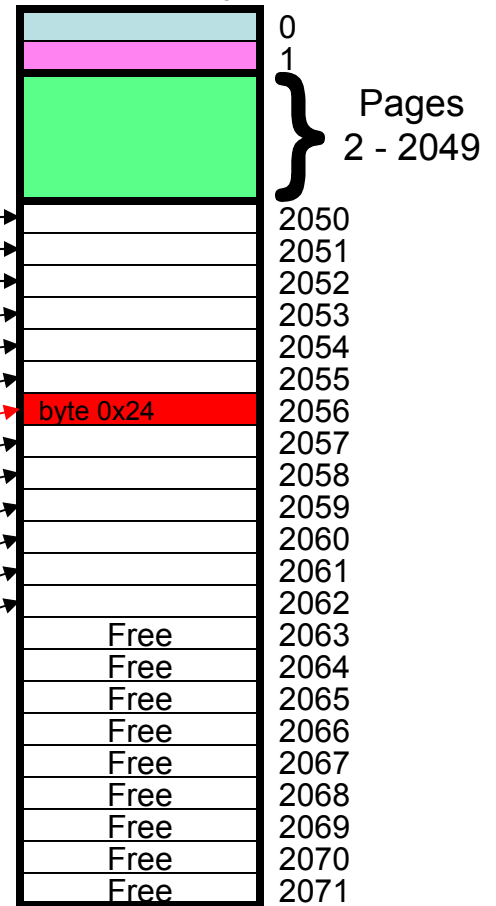
# Segmented Page Table

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

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

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100  
 = 001 00000000000000000001 00000100100  
 = 0x1 0x1 0x24

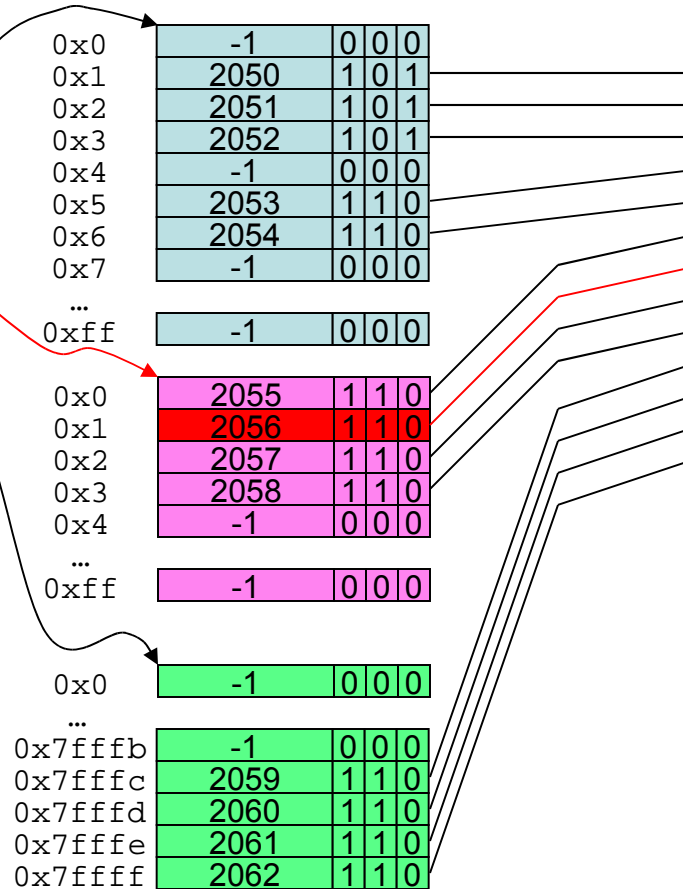
## Main Memory



## Segment Registers

|        |      |      |
|--------|------|------|
| STBR00 | 0    | 1111 |
| STLR00 | 1    |      |
| STBR01 | 1    | 1110 |
| STLR01 | 1    |      |
| STBR10 | -1   | 0000 |
| STLR10 | -1   |      |
| STBR11 | 2    | 1110 |
| STLR11 | 2048 |      |

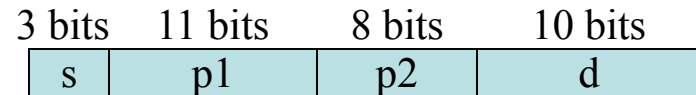
|         |      |     |
|---------|------|-----|
| 0x0     | -1   | 000 |
| 0x1     | 2050 | 101 |
| 0x2     | 2051 | 101 |
| 0x3     | 2052 | 101 |
| 0x4     | -1   | 000 |
| 0x5     | 2053 | 110 |
| 0x6     | 2054 | 110 |
| 0x7     | -1   | 000 |
| ...     |      |     |
| 0xff    | -1   | 000 |
| 0x0     | 2055 | 110 |
| 0x1     | 2056 | 110 |
| 0x2     | 2057 | 110 |
| 0x3     | 2058 | 110 |
| 0x4     | -1   | 000 |
| ...     |      |     |
| 0xff    | -1   | 000 |
| 0x0     | -1   | 000 |
| ...     |      |     |
| 0x7fffb | -1   | 000 |
| 0x7fffc | 2059 | 110 |
| 0x7fffd | 2060 | 110 |
| 0x7fffe | 2061 | 110 |
| 0x7ffff | 2062 | 110 |



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:

## Finally, let's add another layer of paging:



- 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.

|            |              |           |                             |
|------------|--------------|-----------|-----------------------------|
| 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 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1ffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1ffffc - 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 | - 0x7ffffeff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff | - Stack   | - Pages 0x1fffffc - 0x1fffff |

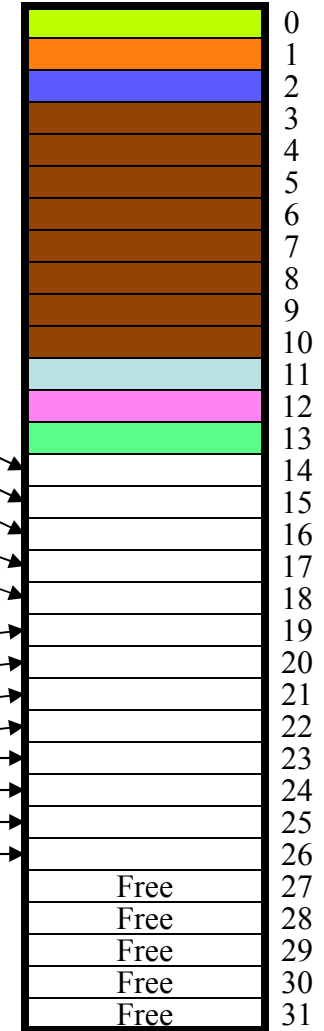
## Segment Registers

|        |    |      |
|--------|----|------|
| STBR00 | 0  | 1111 |
| STLR00 | 1  |      |
| STBR01 | 1  | 1110 |
| STLR01 | 1  |      |
| STBR10 | -1 | 0000 |
| STLR10 | -1 |      |
| STBR11 | 2  | 1110 |
| STLR11 | 8  |      |

|       |    |      |
|-------|----|------|
| 0x0   | 11 | 1111 |
| 0x1   | -1 | 0000 |
| ...   |    |      |
| 0xff  | -1 | 0000 |
| ...   |    |      |
| 0x0   | 12 | 1110 |
| 0x1   | -1 | 1110 |
| ...   |    |      |
| 0xff  | -1 | 0000 |
| ...   |    |      |
| 0x0   | -1 | 0000 |
| ...   |    |      |
| 0x7ef | -1 | 0000 |
| 0x7ff | 13 | 1110 |

|      |    |      |
|------|----|------|
| 0x0  | -1 | 0000 |
| 0x1  | 14 | 1001 |
| 0x2  | 15 | 1001 |
| 0x3  | 16 | 1001 |
| 0x4  | -1 | 0000 |
| 0x5  | 17 | 1110 |
| 0x6  | 18 | 1110 |
| 0x7  | -1 | 0000 |
| ...  |    |      |
| 0xff | -1 | 0000 |
| ...  |    |      |
| 0x0  | 19 | 1110 |
| 0x1  | 20 | 1110 |
| 0x2  | 21 | 1110 |
| 0x3  | 22 | 1110 |
| 0x4  | -1 | 0000 |
| ...  |    |      |
| 0xff | -1 | 0000 |
| ...  |    |      |
| 0x0  | -1 | 0000 |
| ...  |    |      |
| 0xfb | -1 | 0000 |
| 0xfc | 23 | 1110 |
| 0xfd | 24 | 1110 |
| 0xfe | 25 | 1110 |
| 0xff | 26 | 1110 |

## Main Memory



# Two-Level Segments

Physical page 20, offset 0x24:

0x14 0x24 = .. 0000 0001 1000 00 0010 0100 =  
 0000 0000 0000 0000 0110 0000 0010 0100 =  
 0x00006024

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

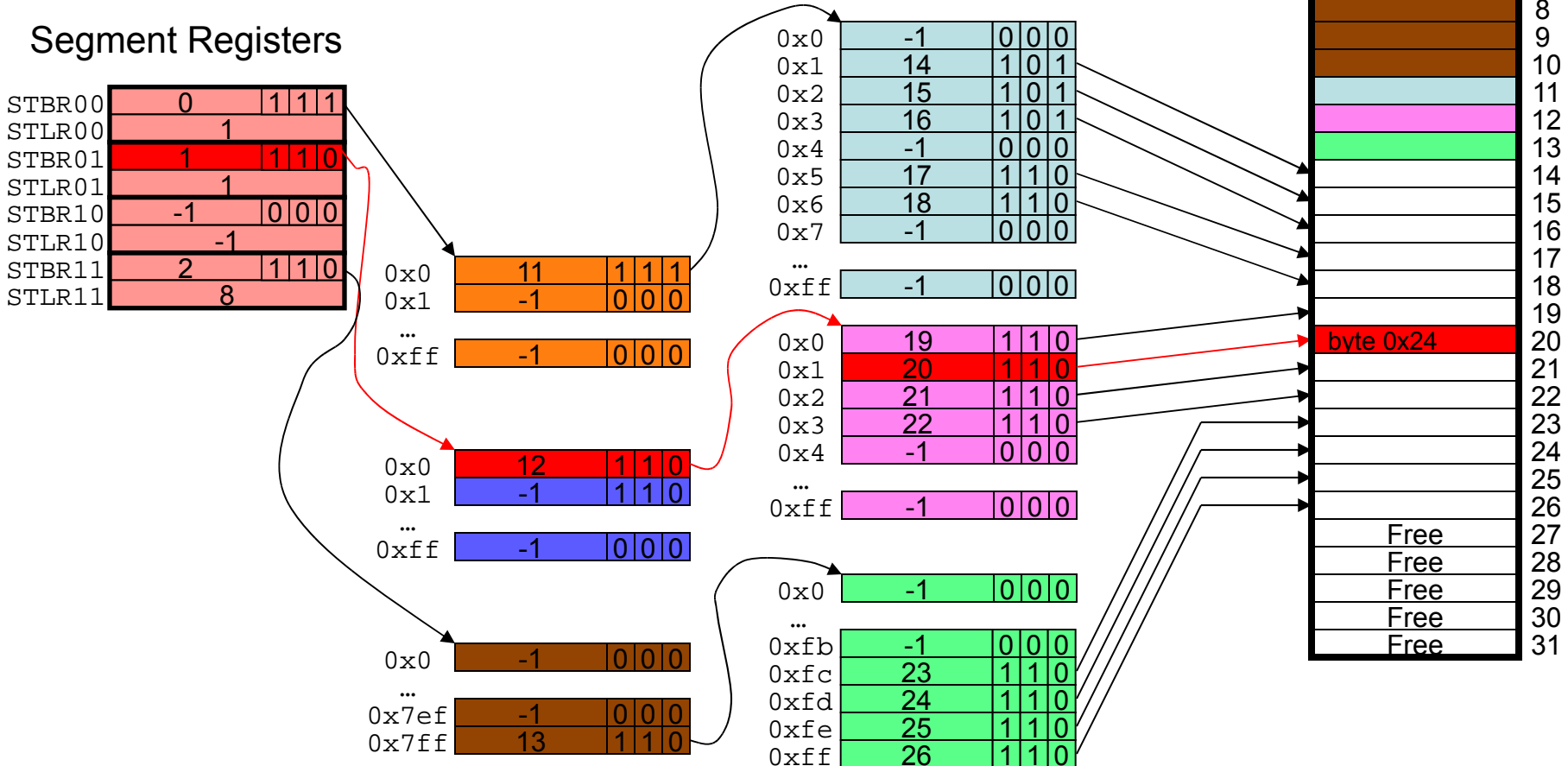
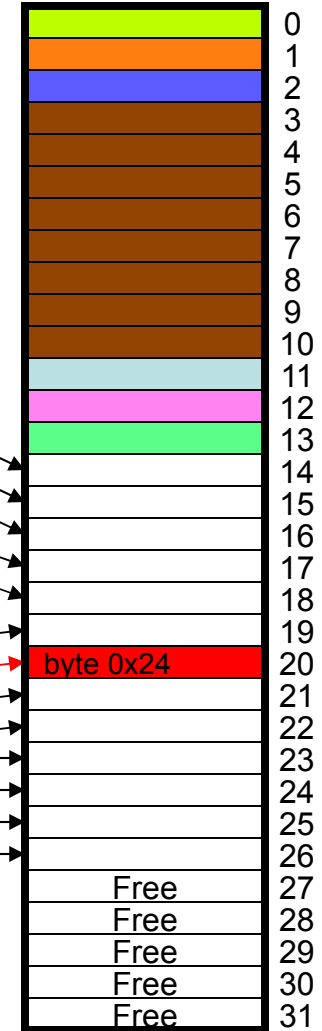
## Segment Registers

|        |    |      |
|--------|----|------|
| STBR00 | 0  | 1111 |
| STLR00 | 1  |      |
| STBR01 | 1  | 1110 |
| STLR01 | 1  |      |
| STBR10 | -1 | 0000 |
| STLR10 | -1 |      |
| STBR11 | 2  | 1110 |
| STLR11 | 8  |      |

|       |    |      |
|-------|----|------|
| 0x0   | 11 | 1111 |
| 0x1   | -1 | 0000 |
| ...   |    |      |
| 0xff  | -1 | 0000 |
| ...   |    |      |
| 0x0   | 12 | 1110 |
| 0x1   | -1 | 1110 |
| ...   |    |      |
| 0xff  | -1 | 0000 |
| ...   |    |      |
| 0x0   | -1 | 0000 |
| ...   |    |      |
| 0x7ef | -1 | 0000 |
| 0x7ff | 13 | 1110 |

|      |    |      |
|------|----|------|
| 0x0  | -1 | 0000 |
| 0x1  | 14 | 1001 |
| 0x2  | 15 | 1001 |
| 0x3  | 16 | 1001 |
| 0x4  | -1 | 0000 |
| 0x5  | 17 | 1110 |
| 0x6  | 18 | 1110 |
| 0x7  | -1 | 0000 |
| ...  |    |      |
| 0xff | -1 | 0000 |
| ...  |    |      |
| 0x0  | 19 | 1110 |
| 0x1  | 20 | 1110 |
| 0x2  | 21 | 1110 |
| 0x3  | 22 | 1110 |
| 0x4  | -1 | 0000 |
| ...  |    |      |
| 0xff | -1 | 0000 |
| ...  |    |      |
| 0x0  | -1 | 0000 |
| ...  |    |      |
| 0xfb | -1 | 0000 |
| 0xfc | 23 | 1110 |
| 0xfd | 24 | 1110 |
| 0xfe | 25 | 1110 |
| 0xff | 26 | 1110 |

## Main Memory



# Two-Level Segments

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

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

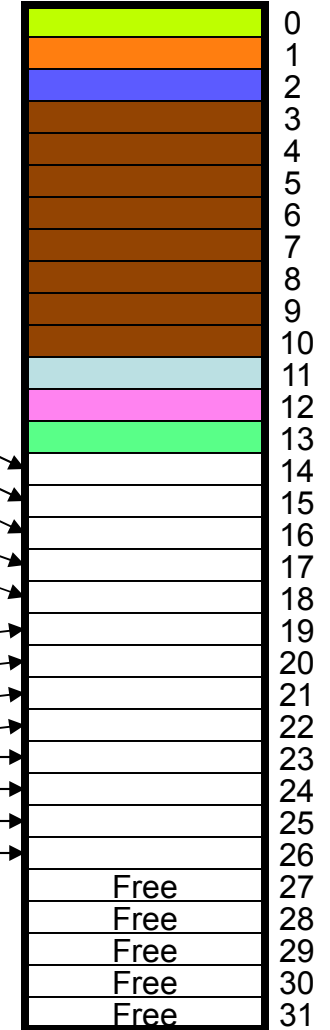
## Segment Registers

|        |    |      |
|--------|----|------|
| STBR00 | 0  | 1111 |
| STLR00 | 1  |      |
| STBR01 | 1  | 1110 |
| STLR01 | 1  |      |
| STBR10 | -1 | 0000 |
| STLR10 | -1 |      |
| STBR11 | 2  | 1110 |
| STLR11 | 8  |      |

|       |    |      |
|-------|----|------|
| 0x0   | 11 | 1111 |
| 0x1   | -1 | 0000 |
| ...   |    |      |
| 0xff  | -1 | 0000 |
| ...   |    |      |
| 0x0   | 12 | 1110 |
| 0x1   | -1 | 1110 |
| ...   |    |      |
| 0xff  | -1 | 0000 |
| ...   |    |      |
| 0x0   | -1 | 0000 |
| ...   |    |      |
| 0x7ef | -1 | 0000 |
| 0x7ff | 13 | 1110 |

|      |    |      |
|------|----|------|
| 0x0  | -1 | 0000 |
| 0x1  | 14 | 1001 |
| 0x2  | 15 | 1001 |
| 0x3  | 16 | 1001 |
| 0x4  | -1 | 0000 |
| 0x5  | 17 | 1110 |
| 0x6  | 18 | 1110 |
| 0x7  | -1 | 0000 |
| ...  |    |      |
| 0xff | -1 | 0000 |
| ...  |    |      |
| 0x0  | 19 | 1110 |
| 0x1  | 20 | 1110 |
| 0x2  | 21 | 1110 |
| 0x3  | 22 | 1110 |
| 0x4  | -1 | 0000 |
| ...  |    |      |
| 0xff | -1 | 0000 |
| ...  |    |      |
| 0x0  | -1 | 0000 |
| ...  |    |      |
| 0xfb | -1 | 0000 |
| 0xfc | 23 | 1110 |
| 0xfd | 24 | 1110 |
| 0xfe | 25 | 1110 |
| 0xff | 26 | 1110 |

## Main Memory

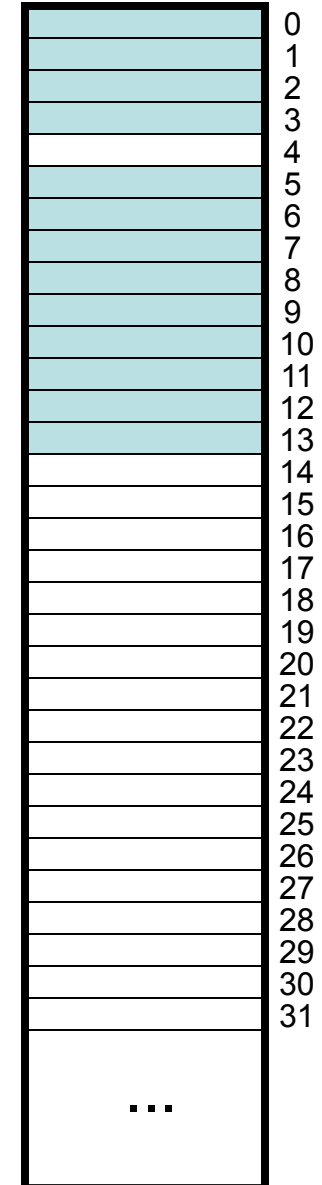




# Inverted Page Table

|            |               |           |                              |
|------------|---------------|-----------|------------------------------|
| 0x00000000 | - 0x000003ff  | - NULL    | - Page: 0x0                  |
| 0x00000400 | - 0x00000fff  | - Code    | - Pages 0x1 - 0x3            |
| 0x00001000 | - 0x000013ff  | - NULL    | - Page: 0x4                  |
| 0x00001400 | - 0x00001bfff | - Globals | - Pages 0x5 - 0x6            |
| 0x00001c00 | - 0x1fffffff  | - NULL    | - Pages 0x7 - 0x7ffff        |
| 0x20000000 | - 0x20000fff  | - Heap    | - Pages 0x80000 - 0x80003    |
| 0x20001000 | - 0x7ffffefff | - NULL    | - Pages 0x80004 - 0x1fffffb  |
| 0x7ffff000 | - 0x7fffffff  | - Stack   | - Pages 0x1fffffc - 0x1fffff |

## Main Memory



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.

|         | 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  |

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.

# 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
            = 001000000000000000000001 00000100100
            = 0x80001 0x24
```

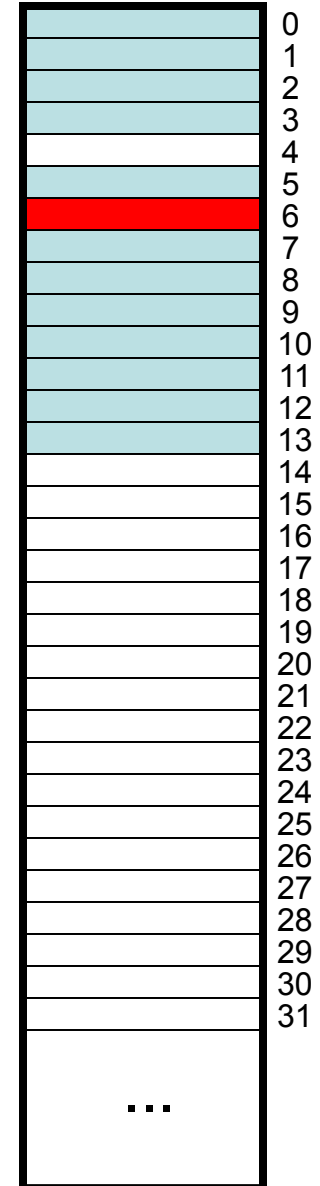
We perform a linear search of the inverted page table until we find page 0x80001 and PID 551. That is page 0x6.

So the address is 0x6 0x24:

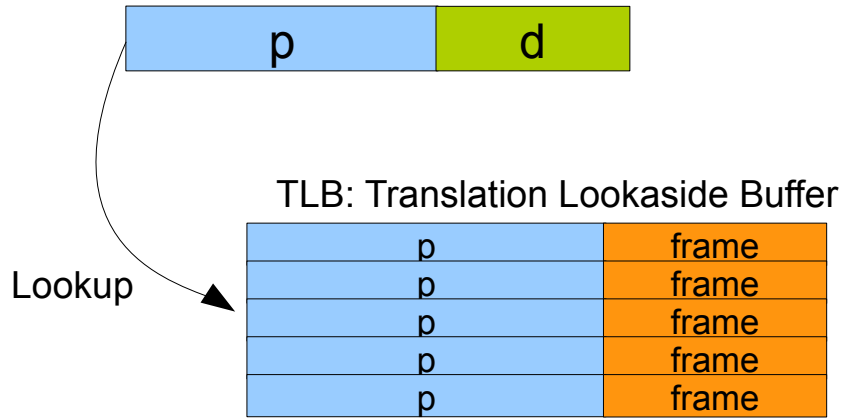
```
110 0000100100 = 0x1824
```

|         | 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     | 0x1ffff  | 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  |

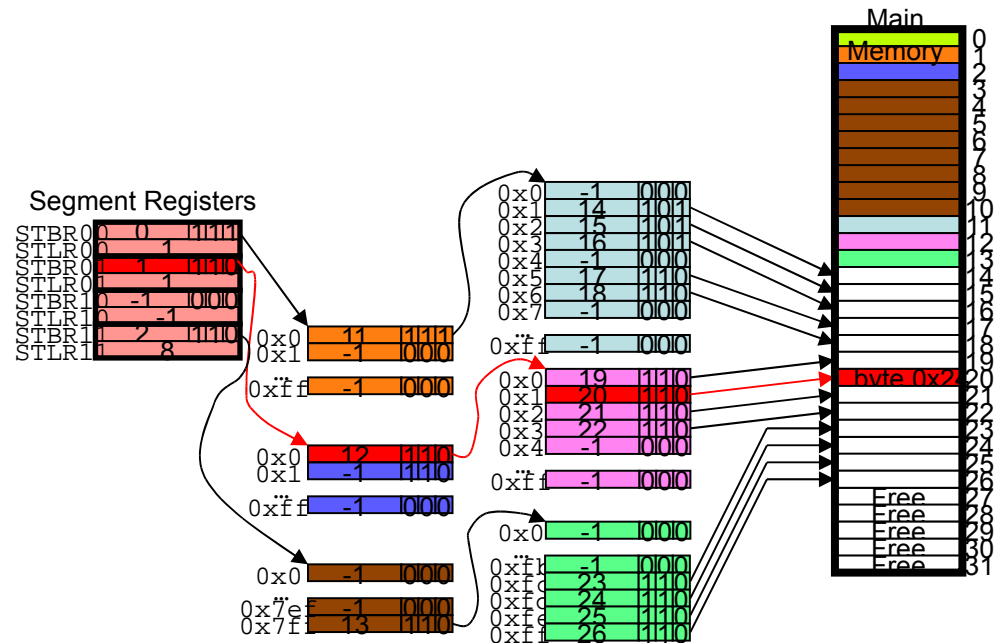
## Main Memory



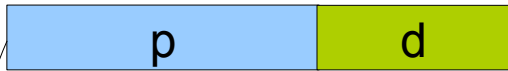
# With a TLB & Cache



Associative Memory:  
Small:  
128-512 entries



# With a TLB & Cache

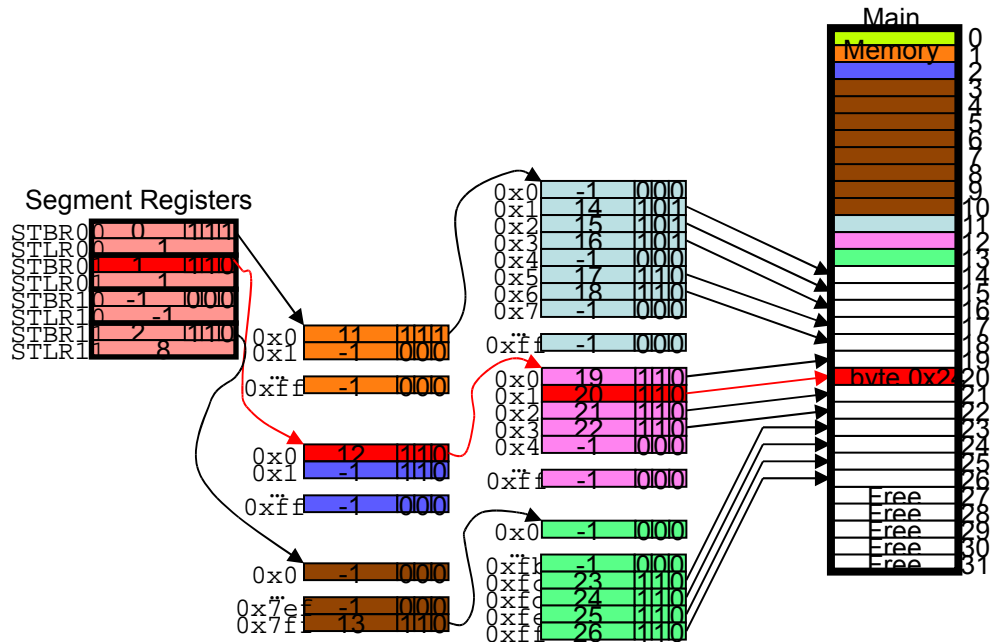


Lookup

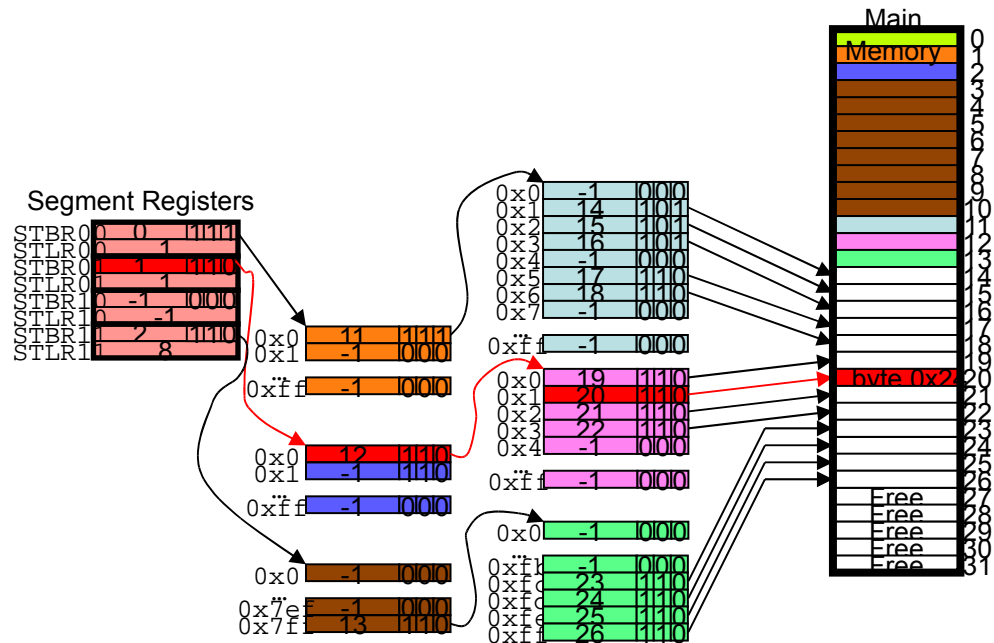
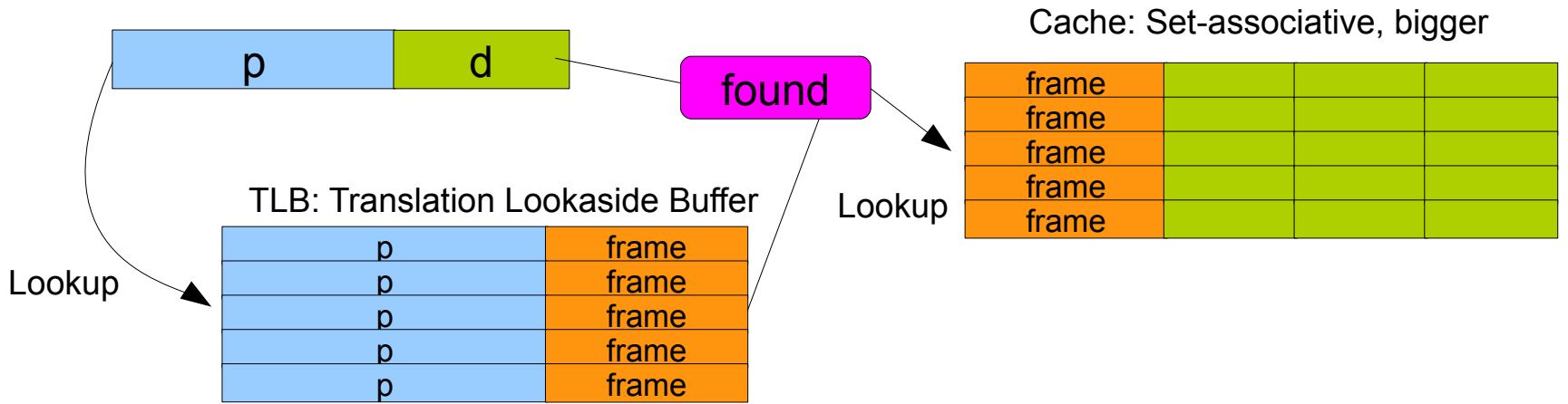
TLB: Translation Lookaside Buffer

|   |       |
|---|-------|
| p | frame |
| p | frame |
| p | frame |
| p | frame |
| p | frame |

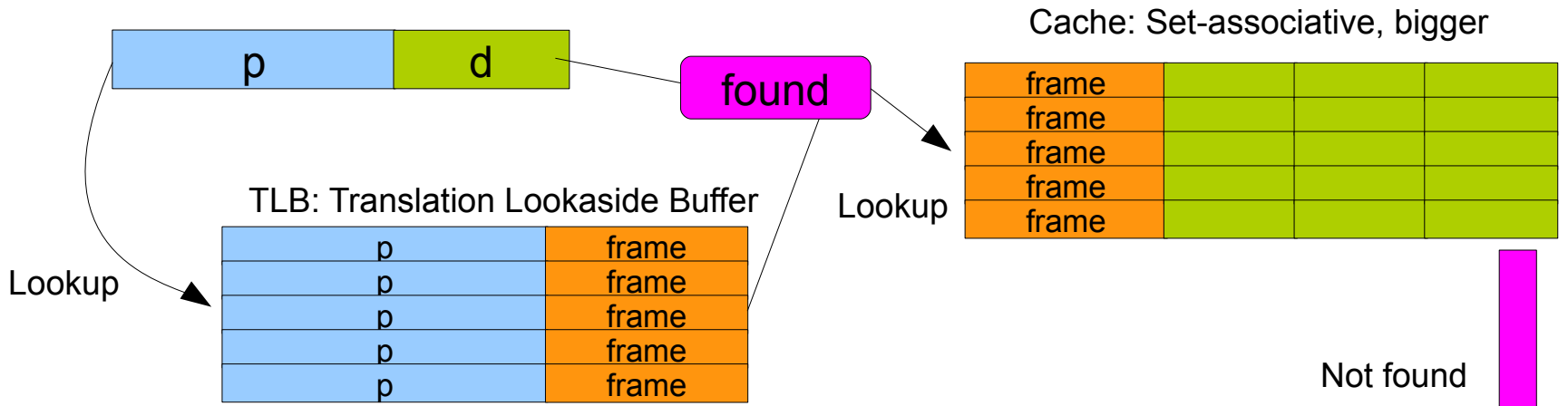
Not found:  
Find & reissue



# With a TLB & Cache



# With a TLB & Cache



Note:  
Caches store  
physical addresses.

