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:

Addresses: 0x0020 - 0x006f and 0x0080 - 0x0fff are illegal.

Addresses: 0x0040 - 0xffff are illegal.
Suppose process P1 is running:

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

Addresses:
0x0020 - 0x006f
and
0x0080 - 0xffff
are illegal.
Suppose process P1 is running:

Addresses:

\[
\begin{align*}
0x0020 & \rightarrow 0x006f \\
0x0080 & \rightarrow 0xfffe
\end{align*}
\]

are illegal.

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

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

Memory:
Suppose process P1 is running:

Addresses:
0x0020 - 0x006f and
0x0080 - 0x0fff are illegal.

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

```
mv #4 -> %r0
st %r0 -> j
```
Suppose process P1 is running:

Addresses:
- 0x0020 - 0x006f
- 0x0080 - 0xffff

are illegal.

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:

Addresses:
0x0020 – 0x006f and 0x0080 – 0xffff are illegal.

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

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

0x0018 = 00000000 00011000

Offset 8
Suppose process P1 is running:

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

Addresses:
0x0020 - 0x006f
and
0x0080 - 0xffff
are illegal.

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

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

0x0018 =

```
00000000 00011000
```

Offset 8
To run process $P2$, you need to switch the PTBR/PTLR:

Addresses: $0x0040 - 0xffff$ are illegal.
What is user address 0x003e?

Addresses: 0x0040 - 0xffff are illegal.

Physical address 0x009e.
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;
Now, suppose you have a process whose address space is as follows:

0x0800-0x1fff: Code
0x2000-0x234c: Globals
0x7ff000-0x7ffffff: Stack

(sp = starting page)
typedef struct {
    PTE **table;
    int    tableSize;
} PageTable;

defined 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:

| 22 bits | 10 bits |
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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Pages</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000</td>
<td>NULL</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>0x00000400</td>
<td>Code</td>
<td>0x1 - 0x3</td>
<td></td>
</tr>
<tr>
<td>0x00001000</td>
<td>NULL</td>
<td>0x4</td>
<td></td>
</tr>
<tr>
<td>0x00014000</td>
<td>Globals</td>
<td>0x5 - 0x6</td>
<td></td>
</tr>
<tr>
<td>0x0001c00</td>
<td>NULL</td>
<td>0x7 - 0x7fff</td>
<td></td>
</tr>
<tr>
<td>0x20000000</td>
<td>Heap</td>
<td>0x8000 - 0x80003</td>
<td></td>
</tr>
<tr>
<td>0x20010000</td>
<td>NULL</td>
<td>0x80004 - 0x1fffffb</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff</td>
<td>Stack</td>
<td>0x1fffffc - 0x1fffffff</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Type</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x000003ff</td>
<td>NULL</td>
<td>Page: 0x0</td>
</tr>
<tr>
<td>0x00000400</td>
<td>0x00000fff</td>
<td>Code</td>
<td>Pages 0x1 - 0x3</td>
</tr>
<tr>
<td>0x00001000</td>
<td>0x000013ff</td>
<td>NULL</td>
<td>Page: 0x4</td>
</tr>
<tr>
<td>0x00001400</td>
<td>0x00001bff</td>
<td>_globals</td>
<td>Pages 0x5 - 0x6</td>
</tr>
<tr>
<td>0x00001c00</td>
<td>0x1fffffff</td>
<td>NULL</td>
<td>Pages 0x7 - 0x7fff</td>
</tr>
<tr>
<td>0x20000000</td>
<td>0x20000fff</td>
<td>Heap</td>
<td>Pages 0x80000 - 0x80003</td>
</tr>
<tr>
<td>0x20001000</td>
<td>0x7fffffff</td>
<td>NULL</td>
<td>Pages 0x80004 - 0x1fffffff</td>
</tr>
<tr>
<td>0x7fffffff000</td>
<td>0x7fffffff</td>
<td>Stack</td>
<td>Pages 0x1fffffff - 0x1fffffff</td>
</tr>
</tbody>
</table>
Single-Level Page Table

0x00000000 - 0x0000003ff - NULL - Page: 0x0
0x00000040 - 0x00000fff - Code - Pages 0x1 - 0x3
0x00000100 - 0x000013ff - NULL - Page: 0x4
0x000001400 - 0x00001bff - Globals - Pages 0x5 - 0x6
0x00001c00 - 0x1fffffff - NULL - Pages 0x7 - 0x7ffff
0x20000000 - 0x20000fff - Heap - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffffff - Stack - Pages 0x1ffffc - 0x1fffff

0x2000 = 8192 Pages
Memory usage: 8205 KB: Horrible!

Single-Level Page Table

- **PTBR**: 0
- **PTLR**: 8192

**Main Memory**

- Pages 0 - 8191

**0x2000 = 8192 Pages**

- **0x00000000 - 0x000003ff**: NULL - Page: 0x0
- **0x00000400 - 0x00000fff**: Code - Pages 0x1 - 0x3
- **0x00001000 - 0x000013ff**: NULL - Page: 0x4
- **0x00001400 - 0x0001bff**: Globals - Pages 0x5 - 0x6
- **0x0001c00 - 0x1fffffff**: NULL - Pages 0x7 - 0x7fff
- **0x20000000 - 0x20000fff**: Heap - Pages 0x80000 - 0x80003
- **0x20001000 - 0x7ffefff**: NULL - Pages 0x80004 - 0x1ffffb
- **0x7ffff000 - 0x7fffffff**: Stack - Pages 0x1ffffc - 0x1ffffff
Let’s find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 0010000000000000000001 0000100100
= 0x80001 0x24

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

0x2000 = 8192 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.

Now, try a two-level page table:

<table>
<thead>
<tr>
<th>14 bits</th>
<th>8 bits</th>
<th>10 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>d</td>
</tr>
</tbody>
</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.
Two-Level Page Table

- 0x00000000 - 0x0000003ff - NULL - Page: 0x0
- 0x00000400 - 0x00000fff - Code - Pages 0x1 - 0x3
- 0x00001000 - 0x000013ff - NULL - Page: 0x4
- 0x00001400 - 0x00001bff -Globals - Pages 0x5 - 0x6
- 0x00001c00 - 0xfffffff - NULL - Pages 0x7 - 0x7fff
- 0x20000000 - 0x20000fff - Heap - Pages 0x80000 - 0x80003
- 0x20001000 - 0x7fffefff - NULL - Pages 0x80004 - 0x1ffffb
- 0x7ffffff0 - 0x7fffffff - Stack - Pages 0x1ffffc - 0x1fffffff

Main Memory

- Pages 0 - 31

8192/256 = 32 Pages

PTBR 0
PTLR 32
Let's find byte 0x20000424, which is in the second page of the heap.

Physical page 41, offset 0x24:
0x29 0x24 = .. 0000 0010 1001 00 0010 0100 
0x800 0x1 0x24

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 010000000000 00000001 000100100
= 0x800 0x1 0x24

8192/256 = 32 Pages

0x0 32 1 1 1
0x1 -1 0 0 0
0x2 -1 0 0 0
... 0x7f -1 0 0 0
0x800 33 1 1 0
0x801 -1 0 0 0
... 0x1ff -1 0 0 0
0x20000424 = 0x800 0x1 0x24

0x0 -1 0 0 0
0x1 35 1 0 1
0x2 36 1 0 1
0x3 37 1 0 1
0x4 -1 0 0 0
0x5 38 1 1 0
0x6 39 1 1 0
0x7 -1 0 0 0
... 
0xff -1 0 0 0

0x0 40 1 1 0
0x1 41 1 1 0
0x2 42 1 1 0
0x3 43 1 1 0
0x4 -1 0 0 0
... 
0xff -1 0 0 0

Physical page 41, offset 0x24:
0x29 0x24 = .. 0000 0010 1001 00 0010 0100 
0x800 0x1 0x24

0x00000a424

8192/256 = 32 Pages

0x0 32 1 1 1
0x1 -1 0 0 0
0x2 -1 0 0 0
... 0x7f -1 0 0 0
0x800 33 1 1 0
0x801 -1 0 0 0
... 0x1ff -1 0 0 0
0x20000424 = 0x800 0x1 0x24

0x0 -1 0 0 0
0x1 35 1 0 1
0x2 36 1 0 1
0x3 37 1 0 1
0x4 -1 0 0 0
0x5 38 1 1 0
0x6 39 1 1 0
0x7 -1 0 0 0
... 
0xff -1 0 0 0

0x0 40 1 1 0
0x1 41 1 1 0
0x2 42 1 1 0
0x3 43 1 1 0
0x4 -1 0 0 0
... 
0xff -1 0 0 0

Physical page 41, offset 0x24:
0x29 0x24 = .. 0000 0010 1001 00 0010 0100 
0x800 0x1 0x24

0x00000a424
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:

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>8 bits</th>
<th>8 bits</th>
<th>10 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</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.

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0x000003ff</td>
<td>NULL</td>
<td>Page: 0x0</td>
</tr>
<tr>
<td>0x00000400 - 0x00000fff</td>
<td>Code</td>
<td>Pages 0x1 - 0x3</td>
</tr>
<tr>
<td>0x00001000 - 0x000013ff</td>
<td>NULL</td>
<td>Page: 0x4</td>
</tr>
<tr>
<td>0x00001400 - 0x00001bff</td>
<td>Globals</td>
<td>Pages 0x5 - 0x6</td>
</tr>
<tr>
<td>0x00001c00 - 0x1fffffff</td>
<td>NULL</td>
<td>Pages 0x7 - 0x7ffff</td>
</tr>
<tr>
<td>0x20000000 - 0x20000fff</td>
<td>Heap</td>
<td>Pages 0x80000 - 0x80003</td>
</tr>
<tr>
<td>0x20001000 - 0x7fffffff</td>
<td>NULL</td>
<td>Pages 0x80004 - 0x1fffffff</td>
</tr>
<tr>
<td>0x7fffffff000 - 0x7ffffffff</td>
<td>Stack</td>
<td>Pages 0x1ffffffff - 0x1fffffff</td>
</tr>
</tbody>
</table>
Three-Level Page Table

Memory usage: 20 KB: Excellent!
Let’s find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 001000 0000000 00000001 0000100100
= 0x8 0x00 0x1 0x24

Physical page 13, offset 0x24:
0xd0 0x24 = .. 0000 0000 1101 00 0010 0100 =
0000 0000 0000 0000 0001 0100 0010 0100 =
0x00003424

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

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:

<table>
<thead>
<tr>
<th>s</th>
<th>p</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bits</td>
<td>19 bits</td>
<td>10 bits</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0x000003ff</td>
<td>NULL</td>
</tr>
<tr>
<td>0x00000400 - 0x00000fff</td>
<td>Code</td>
</tr>
<tr>
<td>0x00001000 - 0x000013ff</td>
<td>NULL</td>
</tr>
<tr>
<td>0x00001400 - 0x00001bff</td>
<td>Globals</td>
</tr>
<tr>
<td>0x00001c00 - 0x1fffffff</td>
<td>NULL</td>
</tr>
<tr>
<td>0x20000000 - 0x20000fff</td>
<td>Heap</td>
</tr>
<tr>
<td>0x20001000 - 0x7fffefff</td>
<td>NULL</td>
</tr>
<tr>
<td>0x7fffffff000 - 0x7ffffffff</td>
<td>Stack</td>
</tr>
</tbody>
</table>
### Segmented Page Table

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Type</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000000 - 0x0000003ff</td>
<td>NULL</td>
<td>Page: 0x0</td>
</tr>
<tr>
<td>0x000000400 - 0x00000fff</td>
<td>Code</td>
<td>Pages: 0x1 - 0x3</td>
</tr>
<tr>
<td>0x000001000 - 0x0000013ff</td>
<td>NULL</td>
<td>Page: 0x4</td>
</tr>
<tr>
<td>0x000014000 - 0x00001bff</td>
<td>Globals</td>
<td>Pages: 0x5 - 0x6</td>
</tr>
<tr>
<td>0x00001c000 - 0x1fffffff</td>
<td>NULL</td>
<td>Pages: 0x7 - 0x7ffff</td>
</tr>
<tr>
<td>0x200000000 - 0x2000003ff</td>
<td>Heap</td>
<td>Pages: 0x80000 - 0x80003</td>
</tr>
<tr>
<td>0x200010000 - 0x7ffffffff</td>
<td>NULL</td>
<td>Pages: 0x80004 - 0x1ffffffff</td>
</tr>
<tr>
<td>0x7ffffffff000 - 0x7ffffffff</td>
<td>Stack</td>
<td>Pages: 0x1ffffffffc - 0x1ffffffff</td>
</tr>
</tbody>
</table>

### Main Memory

#### Segment Registers

<table>
<thead>
<tr>
<th>Segment Register</th>
<th>Base</th>
<th>Limit</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBR00</td>
<td>0</td>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>STLR00</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STBR01</td>
<td>1</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>STLR01</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STBR10</td>
<td>-1</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>STLR10</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STBR11</td>
<td>2</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>STLR11</td>
<td>2048</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Memory Usage

- **Free** memory locations: 0x00000000 - 0x0000003ff, 0x200000000 - 0x2000003ff
- **Used** memory locations: 0x000014000 - 0x00001bff, 0x00001c000 - 0x1fffffff

**Memory usage:** 2063 KB: Bad!
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

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:

<table>
<thead>
<tr>
<th>3 bits</th>
<th>11 bits</th>
<th>8 bits</th>
<th>10 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>p1</td>
<td>p2</td>
<td>d</td>
</tr>
</tbody>
</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 - 0x7fffff
0x20000000 - 0x20000fff - Heap - Pages 0x80000 - 0x80003
0x20001000 - 0x7fffffff - NULL - Pages 0x80004 - 0x1fffffff
0x7fffffff000 - 0x7fffffff - Stack - Pages 0x1fffffff - 0x1fffffff
```
Two-Level Segments

0x00000000 - 0x0000003ff - 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 - 0x1ffffb
0x7fffffff - 0x7fffffff - Stack - Pages 0x1ffffc - 0x1fffff

Memory usage: 27 KB: Good!

Segment Registers

Segment Registers

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

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
    = 001 0000000000 00000001 00000100100
    = 0x1 0x0 0x1 0x24
Two-Level Segments

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

0x20000424 = 0010 0000 0000 0000 0100 0110 0100
= 001 0000000000 00000001 00000100100
= 0x1 0x0 0x1 0x24

Segment Registers

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

Main Memory

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

0x20000424 = 0010 0000 0000 0000 0100 0010 0100
= 00100000000000000000001 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
With a TLB & Cache

TLB: Translation Lookaside Buffer

Lookup

Segment Registers

Main Memory

Associative Memory: Small: 128-512 entries
With a TLB & Cache

Segment Registers

Main Memory

TLB: Translation Lookaside Buffer

Lookup

Not found: Find & reissue
With a TLB & Cache

Cache: Set-associative, bigger

TLB: Translation Lookaside Buffer

Segment Registers

Main Memory

Lookup

Segment Registers

Main Memory

Lookup
With a TLB & Cache

TLB: Translation Lookaside Buffer

Cache: Set-associative, bigger

Segment Registers

Main Memory

Note:
Caches store physical addresses.