Suppose:

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

Then:

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

Processes can consume 4096 pages (were there enough memory).
There are two processes in memory:

**P1**
- Addresses: 0x0020 - 0x006f and 0x0080 - 0xfff are illegal.

**P2**
- Addresses: 0x0040 - 0xfff are illegal.
Suppose process P1 is running:

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

The PTBR and PTLR are set so that the hardware will find the right words.
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
Suppose process P1 is running:

Addresses:
- $0x0000 - 0x006f$
- $0x0070 - 0x007c$
- $0x0080 - 0x0fff$

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

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

```
0x0000 0x0004 0x0008 0x000c 0x0010 0x0014 0x0018 0x001c
0x0020 0x0024 0x0028 0x002c 0x0030 0x0034 0x0038 0x003c
0x0040 0x0044 0x0048 0x004c 0x0050 0x0054 0x0058 0x005c
0x0060 0x0064 0x0068 0x006c 0x0070 0x0074 0x0078 0x007c
0x0080 0x0084 0x0088 0x008c 0x0090 0x0094 0x0098 0x009c
```

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
```

```
0x0018 = 00000000 00011000
```

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: 

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
</tr>
<tr>
<td>0x0004</td>
</tr>
<tr>
<td>0x0008</td>
</tr>
<tr>
<td>0x0010</td>
</tr>
<tr>
<td>0x0014</td>
</tr>
<tr>
<td>0x0018</td>
</tr>
<tr>
<td>0x001c</td>
</tr>
<tr>
<td>0x0020</td>
</tr>
<tr>
<td>0x0024</td>
</tr>
<tr>
<td>0x0028</td>
</tr>
<tr>
<td>0x002c</td>
</tr>
<tr>
<td>0x0030</td>
</tr>
<tr>
<td>0x0034</td>
</tr>
<tr>
<td>0x0038</td>
</tr>
<tr>
<td>0x003c</td>
</tr>
<tr>
<td>0x0040</td>
</tr>
<tr>
<td>0x0044</td>
</tr>
<tr>
<td>0x0048</td>
</tr>
<tr>
<td>0x004c</td>
</tr>
<tr>
<td>0x0050</td>
</tr>
<tr>
<td>0x0054</td>
</tr>
<tr>
<td>0x0058</td>
</tr>
<tr>
<td>0x005c</td>
</tr>
<tr>
<td>0x0060</td>
</tr>
<tr>
<td>0x0064</td>
</tr>
<tr>
<td>0x0068</td>
</tr>
<tr>
<td>0x006c</td>
</tr>
<tr>
<td>0x0070</td>
</tr>
<tr>
<td>0x0074</td>
</tr>
<tr>
<td>0x0078</td>
</tr>
<tr>
<td>0x007c</td>
</tr>
</tbody>
</table>

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

- PTBR: 0x003
- PTLR: 0x02

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

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.

Memory: 0x0000 0x0004 0x0008 0x000c 0x0010 0x0014 0x0018 0x001c 0x0020 0x0024 0x0028 0x002c 0x0030 0x0034 0x0038 0x003c 0x0040 0x0044 0x0048 0x004c 0x0050 0x0054 0x0058 0x005c 0x0060 0x0064 0x0068 0x006c 0x0070 0x0074 0x0078 0x007c 0x0080 0x0084 0x0088 0x008c 0x0090 0x0094 0x0098 0x009c
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)

```c
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
- 0x7ffe000-0x7fffffff: Stack

```c
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 —— 12 pages: SP 4
- 0x2000-0x234c: Globals —— 2 pages: SP 16
- 0x7ff000-0x7fffff: Stack — 8 pages: SP 16376

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

typedef struct {
    int physicalFrame;
    bool valid;
    bool dirty;
    bool use;
} PTE;
```
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
0x7fd000-0x7ffffff: Stack —— 8 pages: SP 16376
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.
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.

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.

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>Contents</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>NULL</td>
<td>0x000003ff</td>
</tr>
<tr>
<td>0x00000400</td>
<td>Code</td>
<td>0x00000fff</td>
</tr>
<tr>
<td>0x00001000</td>
<td>NULL</td>
<td>0x000013ff</td>
</tr>
<tr>
<td>0x00001400</td>
<td>Globals</td>
<td>0x00001bff</td>
</tr>
<tr>
<td>0x00001c00</td>
<td>NULL</td>
<td>0x1fffffff</td>
</tr>
<tr>
<td>0x20000000</td>
<td>Heap</td>
<td>0x20000fff</td>
</tr>
<tr>
<td>0x20001000</td>
<td>NULL</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>0x7fffffff</td>
<td>Stack</td>
<td>0x7fffffff</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.

Now, suppose we have a single-level page table. Draw memory.
Single-Level Page Table

0x00000000 – 0x0000003ff – NULL – Page: 0x0
0x00000040 – 0x00000fff – Code – Pages 0x1 – 0x3
0x00001000 – 0x000013ff – NULL – Page: 0x4
0x00001400 – 0x00001bff – Globals – Pages 0x5 – 0x6
0x0001c00 – 0x1ffffff – NULL – Pages 0x7 – 0x7ffff
0x20000000 – 0x20000fff – Heap – Pages 0x80000 – 0x80003
0x20001000 – 0x7fffffff – NULL – Pages 0x80004 – 0x1ffffff
0x7fffffff000 – 0x7fffffff – Stack – Pages 0x1ffffffc – 0x1ffffff

PTBR
8192
PTLR

0x2000 = 8192 Pages

Main Memory

Pages 0 - 8191
0x00000000 – 0x000003ff – NULL – Page: 0x0
0x00000400 – 0x00000fff – Code – Pages 0x1 – 0x3
0x000001000 – 0x000013ff – NULL – Page: 0x4
0x000001400 – 0x00001bff – Globals – Pages 0x5 – 0x6
0x000001c00 – 0x1fffffff – NULL – Pages 0x7 – 0x7fff
0x20000000 – 0x20000fff – Heap – Pages 0x80000 – 0x80003
0x20001000 – 0x7fffffff – NULL – Pages 0x80004 – 0x1fffe
0x7fffffff000 – 0x7fffffff – Stack – Pages 0x1fffffff – 0x1fffffff

Memory usage: 8204 KB: Horrible!

Single-Level Page Table

Main Memory

Pages 0 - 8191

0x2000 = 8192 Pages

PTBR
8192

PTLR
8192
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.

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>

\[ d = 10 \text{ bits}: 1K \text{ pages} \]
\[ p2 = 8 \text{ bits}: 256 \text{ PTEs/Page} \]
\[ p1 = 14 \text{ bits}: \text{Remainder} \]

There are \(2^{13} = 8K\) first-level PTE’s. Why? The first bit is always 0.
Two-Level Page Table

Memory usage: 47 KB: Better!

Main Memory

8192/256 = 32 Pages

PTBR
PTLR

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 – 0x20001bff – NULL – Pages 0x80004 – 0x1ffffb
0x7ffffff0 – 0x7fffffff – Stack – Pages 0x1ffffc – 0x1fffff
Let’s find byte 0x20000424, which is in the second page of the heap.

0x20000424 = 0010 0000 0000 0000 0000 0100 0010 0100
= 001000000000000000000001 00001000100
= 0x800 0x1 0x24

8192/256 = 32 Pages

8192/256/256 = 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.

How about a 3-level page table:

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

\[ d = 10 \text{ bits: 1K pages} \]
\[ p2/3 = 8 \text{ bits: 256 PTEs/Page} \]
\[ p1 = 6 \text{ bits: Remainder} \]

There are \( 2^5 = 32 \) first-level PTE’s.

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 – 0x000003ff</td>
<td>NULL – Page: 0x0</td>
</tr>
<tr>
<td>0x00000400 – 0x00000fff</td>
<td>Code – Pages 0x1 – 0x3</td>
</tr>
<tr>
<td>0x00001000 – 0x000013ff</td>
<td>NULL – Page: 0x4</td>
</tr>
<tr>
<td>0x00001400 – 0x00001bff</td>
<td>Globals – Pages 0x5 – 0x6</td>
</tr>
<tr>
<td>0x00001c00 – 0x1fffffff</td>
<td>NULL – Pages 0x7 – 0x7fffffff</td>
</tr>
<tr>
<td>0x20000000 – 0x20000fff</td>
<td>Heap – Pages 0x80000 – 0x80003</td>
</tr>
<tr>
<td>0x20001000 – 0x7fffffff</td>
<td>NULL – Pages 0x80004 – 0x1fffffff</td>
</tr>
<tr>
<td>0x7fffffff000 – 0x7ffffffff</td>
<td>Stack – Pages 0x1fffffff – 0x1ffffffff</td>
</tr>
</tbody>
</table>
Three-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 – 0x7ffeefff – NULL – Pages 0x80004 – 0x1ffffb
0x7ffff000 – 0x7fffffff – Stack – Pages 0x1ffffc – 0x1fffff

Memory usage: 19 KB: Excellent!
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 00001000100
= 0x8 0x00 0x1 0x24

Main Memory

Byte 0x24

PTBR 0x00000000
PTLR 32
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
- 1 NULL page between code & globals
- Heap starts 512M from beginning of address space
- Stack starts at the last address & grows up.

Now, how about a segmented scheme as follows:

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

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

<table>
<thead>
<tr>
<th>Address Range</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 – 0x7fffffff</td>
<td>NULL</td>
</tr>
<tr>
<td>0x7fffffff000 – 0x7fffffff</td>
<td>Stack</td>
</tr>
</tbody>
</table>
The image shows a diagram of a segmented page table and main memory. The segmented page table includes entries for various segments like Code, Globals, Heap, and Stack. The diagram also illustrates the segment registers and memory usage. The main memory section indicates free space and used memory blocks.
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.

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 – 0x1ffffff – NULL – Pages 0x7 – 0x7ffff
0x20000000 – 0x20000fff – Heap – Pages 0x80000 – 0x80003
0x20001000 – 0x7ffe000 – NULL – Pages 0x80004 – 0x1ffffffb
0x7fff0000 – 0x7fffffff – Stack – Pages 0x1ffffffc – 0x1fffffff
```
Two-Level Segments

Memory usage: 26 KB: Good!

Segment Registers

```
<table>
<thead>
<tr>
<th>Segment</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBR00</td>
<td>0</td>
</tr>
<tr>
<td>STLR00</td>
<td>1</td>
</tr>
<tr>
<td>STBR01</td>
<td>1</td>
</tr>
<tr>
<td>STLR01</td>
<td>1</td>
</tr>
<tr>
<td>STBR10</td>
<td>-1</td>
</tr>
<tr>
<td>STLR10</td>
<td>0</td>
</tr>
<tr>
<td>STBR11</td>
<td>2</td>
</tr>
<tr>
<td>STLR11</td>
<td>0</td>
</tr>
</tbody>
</table>
```

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 - 0x7fffffff - NULL - Pages 0x80004 - 0x1ffffb
0x7fffffff - 0x7fffffff - Stack - Pages 0x1ffffc - 0x1fffff
```

```
0x0     11  111
0x1     11  111
0xff    11  111
```

```
0x0     -1  0  0  0
0x1     14  1  0  1
0x2     15  1  0  1
0x3     16  1  0  1
0x4     -1  0  0  0
0x5     17  1  1  0
0x6     18  1  1  0
0x7     -1  0  0  0
```

```
0x0     19  1  1  0
0x1     20  1  1  0
0x2     21  1  1  0
0x3     22  1  1  0
0x4     -1  0  0  0
```

```
0x0     -1  0  0  0
0xfb    -1  0  0  0
0xfc    23  1  1  0
0xfd    24  1  1  0
0xfe    25  1  1  0
0xff    26  1  1  0
```