Virtual address

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00002</td>
<td>0x0858A</td>
</tr>
<tr>
<td>0x03001</td>
<td>0x02C72</td>
</tr>
<tr>
<td>0x00001</td>
<td>0x94525</td>
</tr>
<tr>
<td>0xABCD</td>
<td>0x741AC</td>
</tr>
</tbody>
</table>

Page table entry

<table>
<thead>
<tr>
<th>Physical page number</th>
<th>Valid</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x813B8</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0x858A</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0x9A525</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0xB0BB1</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Physical Memory (shown as 4-byte words)

- 2-level page table
  - 256 entry L2 page table
  - 256 entry L1 page table
  - 64 KB pages

Cache

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Word 0</th>
<th>Word 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x858A479</td>
<td>71.2</td>
<td>13.9</td>
</tr>
<tr>
<td>1</td>
<td>0x858A400</td>
<td>16.7</td>
<td>56.0</td>
</tr>
<tr>
<td>1</td>
<td>0x02C72000</td>
<td>0x2DB007B4</td>
<td>0x000023790</td>
</tr>
<tr>
<td>1</td>
<td>0x9A525000</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- CR3 register: 0x7A2300000
- MMU
- PA 0: 0x9A525000
- PA 1: 0x02C72000
- PA 2: 0x02C72000
- PA 3: 0x
Managing the heap!

heap for **dynamic memory allocation**

C ➔ malloc
Java ➔ new
C++ ➔ new

C/C++ we use **explicit memory management**

explicit ➔ programmer is required to explicitly allocate & deallocate memory

C ➔ malloc + free

Java: implicit ➔ memory is **garbage collected**

malloc (size) + free (void+)

To get memory ➔ malloc asks the OS

more virtual memory
Sbrk (pronounced ess-break)

System call
Sbrk (incr)

⇒ allocating new Virtual memory
Sbrk (4096)
Sbrk (-10,000)
⇒ freeing memory

When does OS update your page table?
Sbrk is called + OS allocates pages in the page table + physical memory

Goals for memory allocator
- track all allocated blocks
- handle arbitrary requests
- can't changes to old allocations
- it can only use the heap
- alignment restrictions
More about fragmentation

- **External** fragmentation of physical memory

  By there is enough free memory to satisfy the request but not a single free block large enough

```c
int *a = malloc(100);
int *b = malloc(100);
int *c = NULL;
int *d = NULL;
int *e = NULL;
free(b);
free(d);
int *f = malloc(150);
```

- There is 200B free but no block is 150B or larger

```
int x[y[0]] = 3;
```

More addresses

- Why we use *a

```c
int i;
for (i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```
Internal fragmentation within an allocated block we don't use all the space usually occurs because of alignment restrictions.

```
*a = malloc(1)
```