In summary, good things to do,

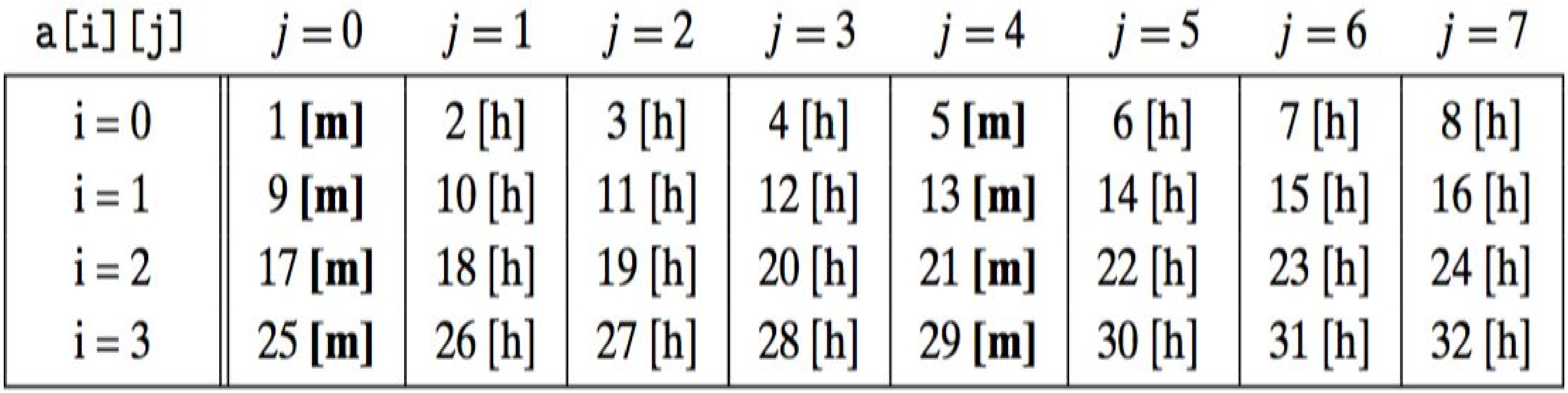
- Repeated references to the local variables are good (temporal locality). Cache them in the registers!
- Stride-1-reference patterns are also good because all caches store data sequentially as contiguous blocks.

Now consider this program

```
int sumarrayrows(int arr[4][8]) {
    int sum = 0;

    for (int i = 0; i < 4; i++)
        for(int j = 0; j < 8; j++)
            sum += arr[i][j];

    return sum;
}
```

- C stores arrays in a row-major order.
- Again, stride-1-reference pattern.

```
int sumarrayrows(int arr[4][8]) {
    int sum = 0;

    for (int i = 0; i < 4; i++)
        for(int j = 0; j < 8; j++)
            sum += arr[i][j];

    return sum;
}
```



a[i] [j]	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
i = 0	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
i = 1	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
i = 2	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
i = 3	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

```
int sumarrayrows(int arr[4][8]) {
    int sum = 0;

    for (int i = 0; i < 4; i++)
        for(int j = 0; j < 8; j++)
            sum += arr[i][j];

    return sum;
}
```

Miss Ratio = 8/32 = 0.25

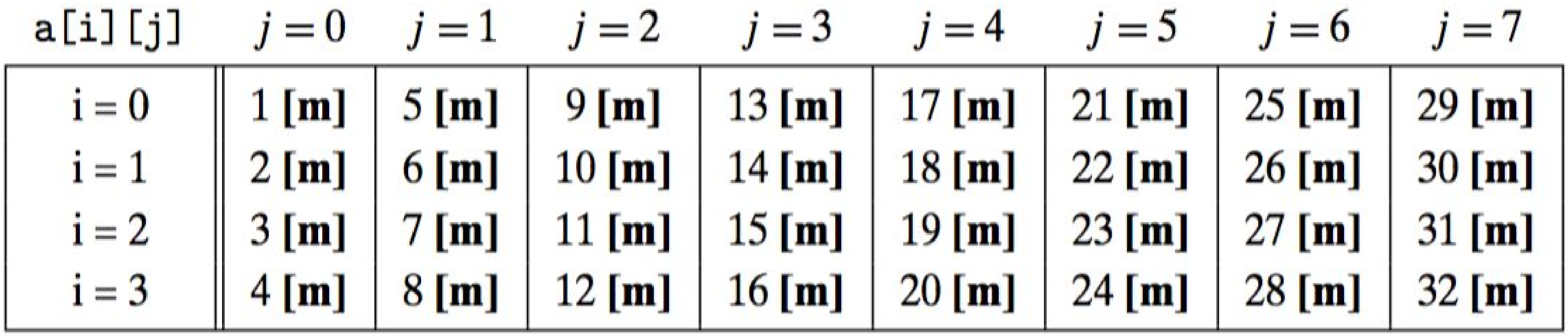
Now what if we reference the array in column-major order?

```
int sumarraycols(int arr[4][8]) {
    int sum = 0;

    for(int j = 0; j < 8; j++)
        for (int i = 0; i < 4; i++)
            sum += arr[i][j];

    return sum;
}
```

If the cache is large enough (to hold the entire array) we may get away with this.

But it’s highly unlikely.

So…



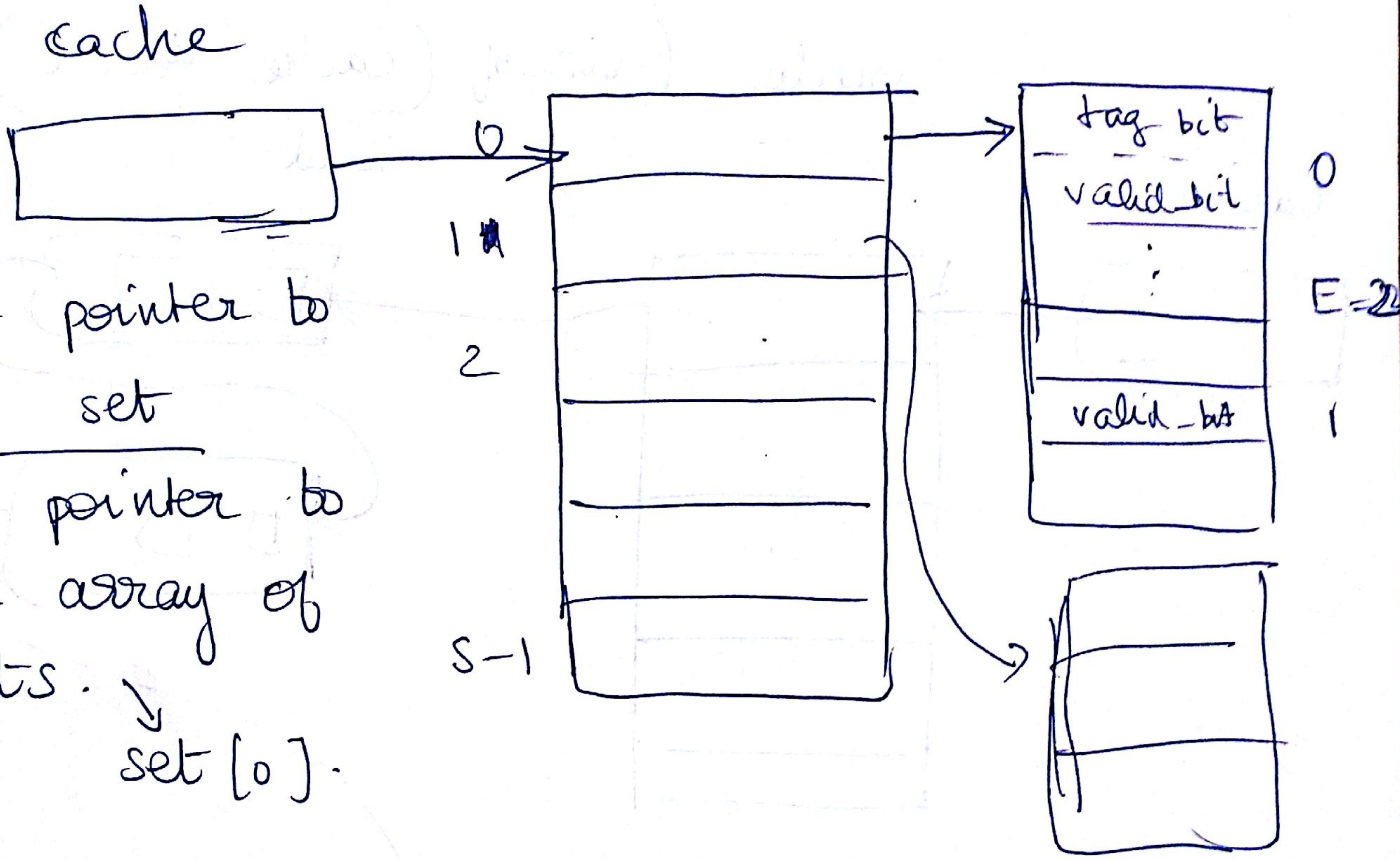
`a[i][j]`	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
i = 0	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
i = 1	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
i = 2	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
i = 3	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

- Access a[0][0]. Cache miss!
- So we load block containing a[0][0] along with a[0][1], a[0][2], a[0][3] onto the cache.
- In the second iteration of the innermost loop, we access a[1][0]. Cache miss again!
- Now load block containing a[1][0] along with a[1][1], a[1][2], a[1][3] onto the cache.
- In the third iteration, we access a[2][0]...

Try working on this example for the following params,
m = 8, S = 4, B = 16 and E = 1.

How many misses do you get?

cache = (cache_set_t *) malloc
(size of (___) * S);



A pointer to a set

A pointer to an array of sets. → set[0].

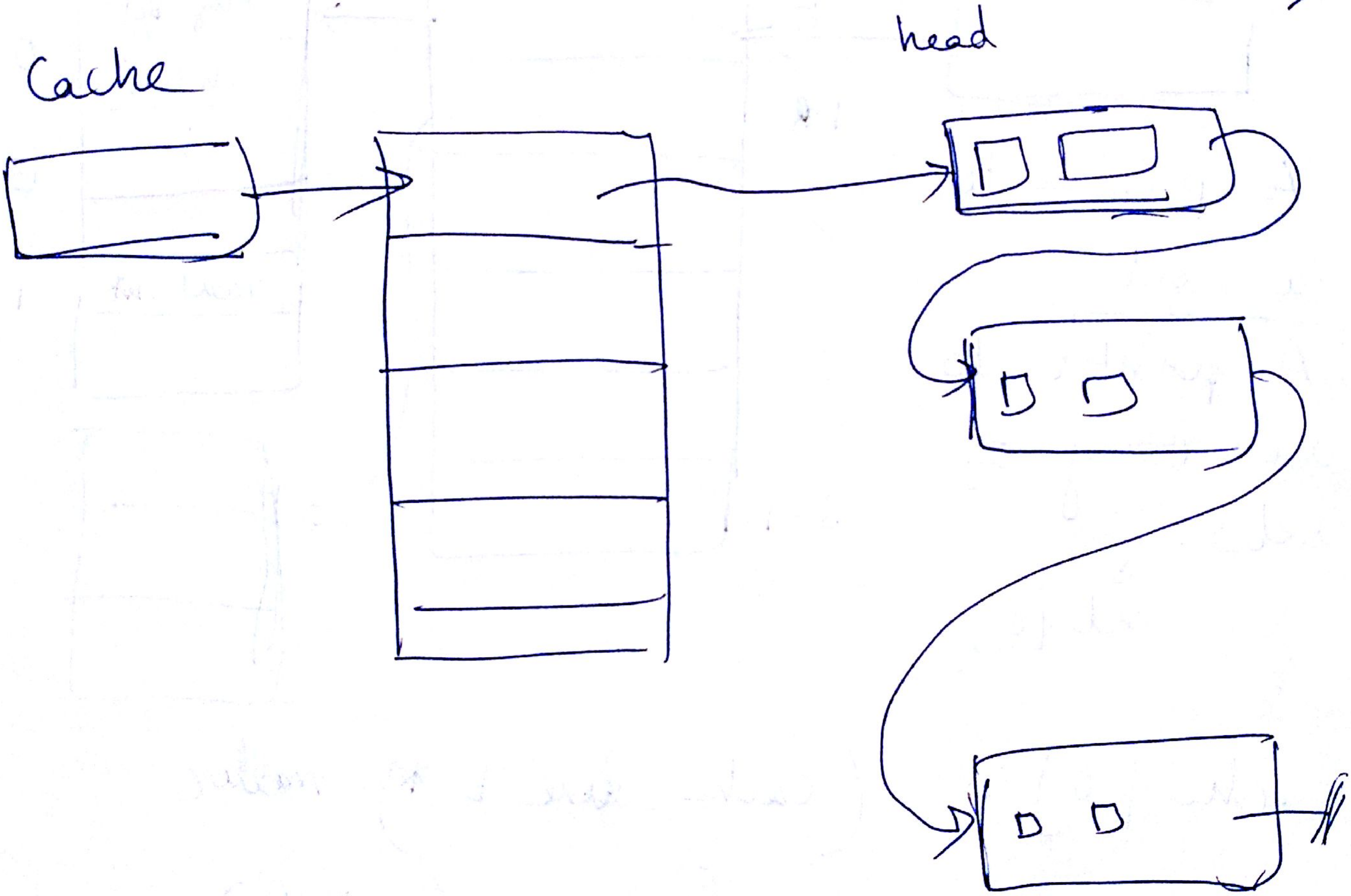
cache[0] = (cache_line_t *) malloc
(size of (c_l_t) * E)

cache[0][1]. valid_bit

Linked List

cache_line_t * newLine = (   )

malloc (sizeof (cache_line_t)));

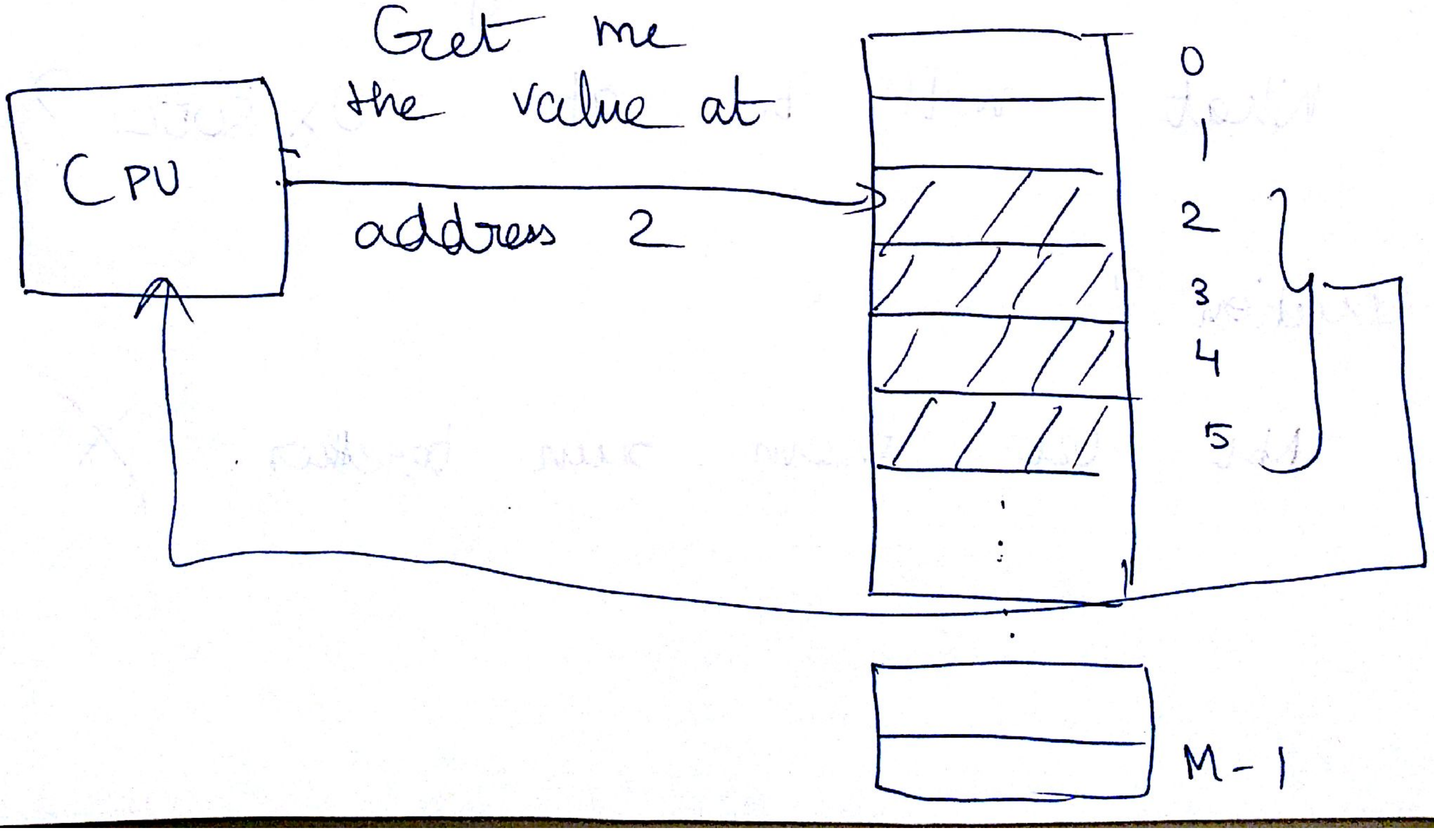


Memory (Physical)

Memory → organized as an array of M contiguous byte-sized cells.

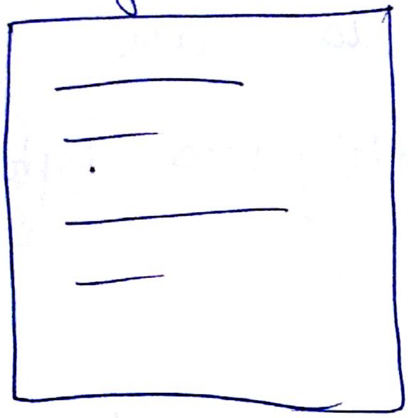
Each byte → has a unique address.

CPU → when it wants to access something from the main memory, it uses its physical address.



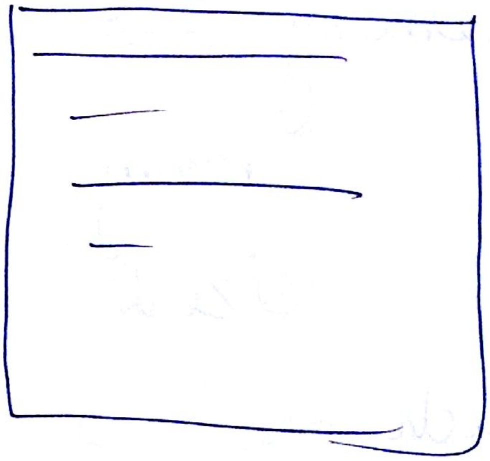
Problems.

① Program A



Need to store a variable X at 0x 805C.

Program B



Need to store a function Y at 0x 805C.

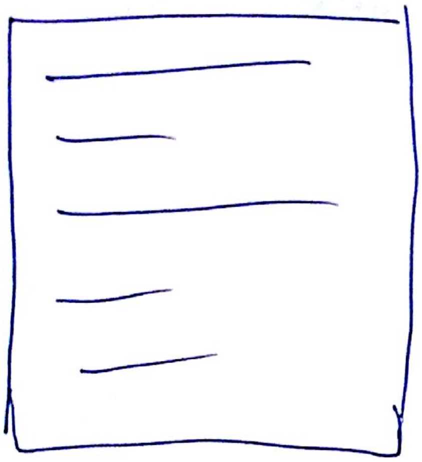
Let them run together.

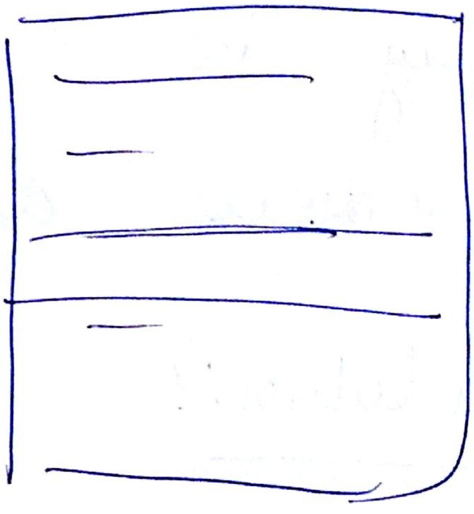
What will be at 0x 805C ?

Solution ?

Not let them run together. ✗

② Prg A   Prg B





"not_a_virus..exe".

Access the address space allocated to Prg B //.

Chrome process running your UWCU account.

③ Prg A   Prg B

Has a bad pointer → modifies a value in A.

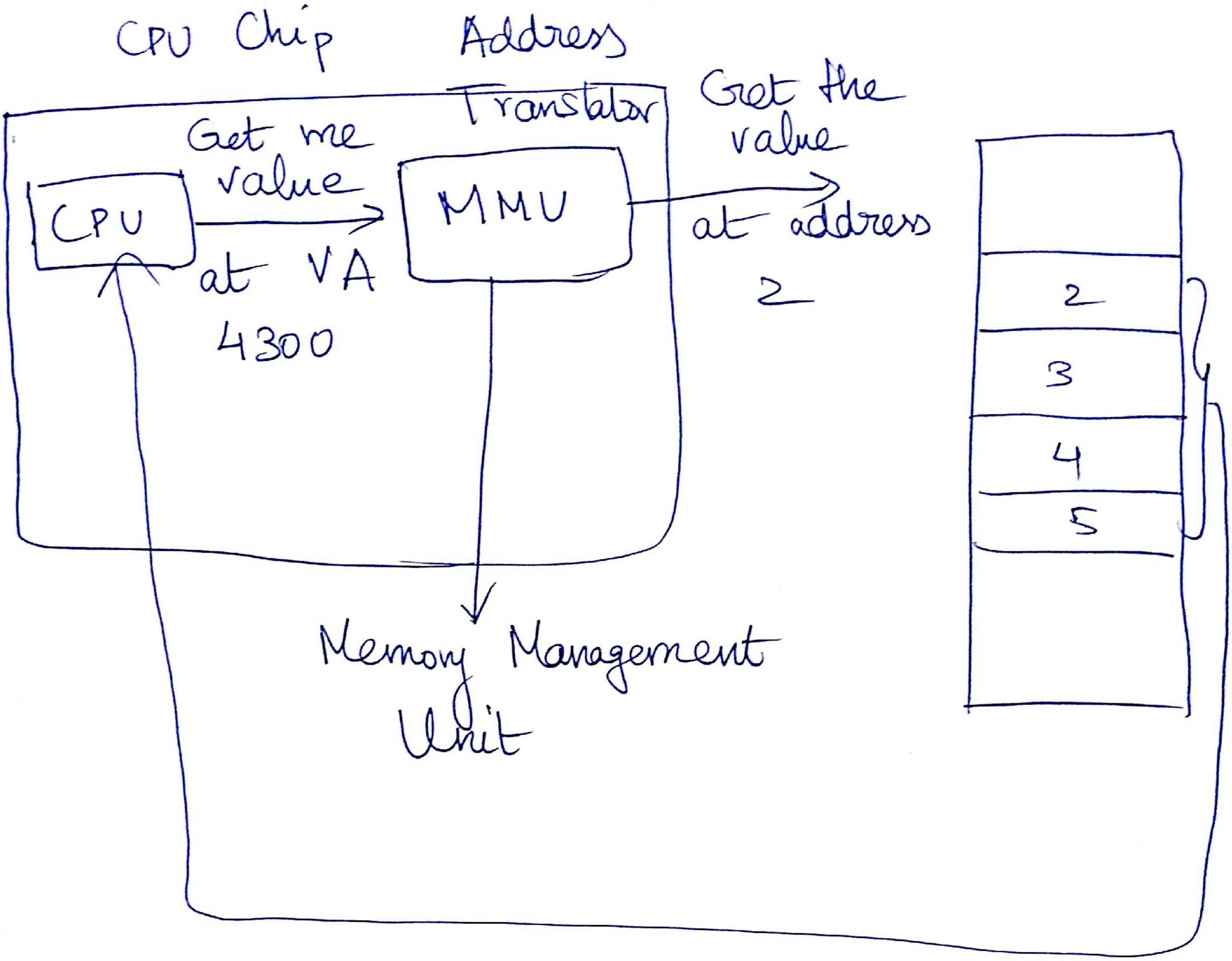
④ · Some parts of the memory may be allocated to the OS, hardware devices.

<u>Solution?</u>

<u>Virtual Addressing</u>.

CPU accesses main memory by generating a <u>virtual address</u>. that gets converted to a <u>physical address</u>.

virtual address $\xrightarrow[\text{Address translator.}]{}$ physical address


CPU Chip
Address Translator
Get me value
at VA 4300
CPU
MMU
Get the value
at address 2
2
3
4
5
Memory Management Unit