Cache

again :-P

& Linking

Bin Liu
Section I

Agenda

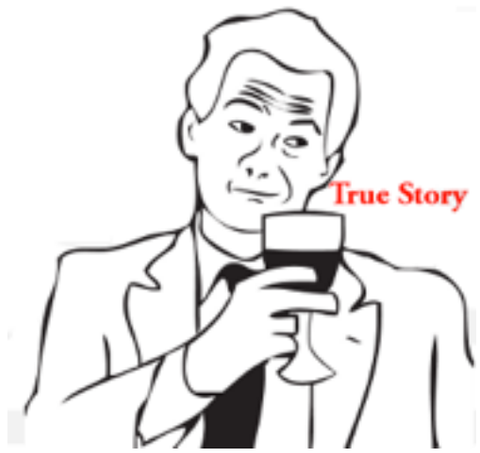
- Yo, I'm a cache simulator
- Linking stuff
- Er.. that's enough for 50min :-P

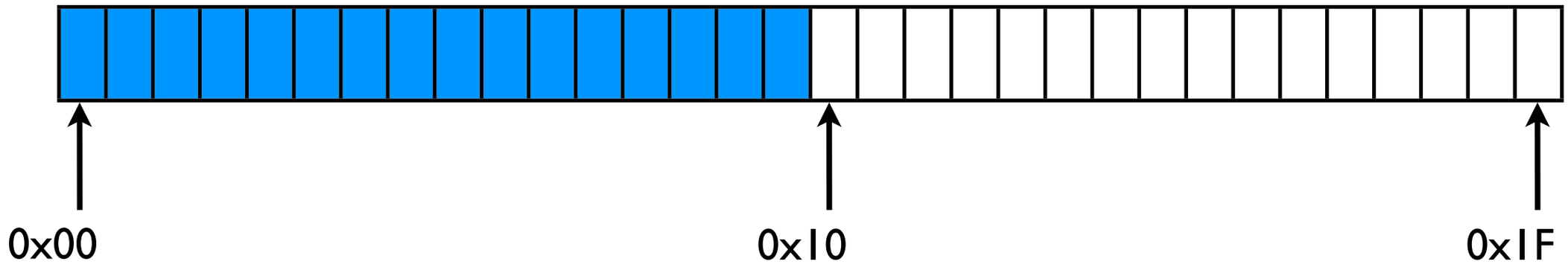
Cache Walkthrough

A piece of Memory!
Addresses are 8 bits

Cache Walkthrough

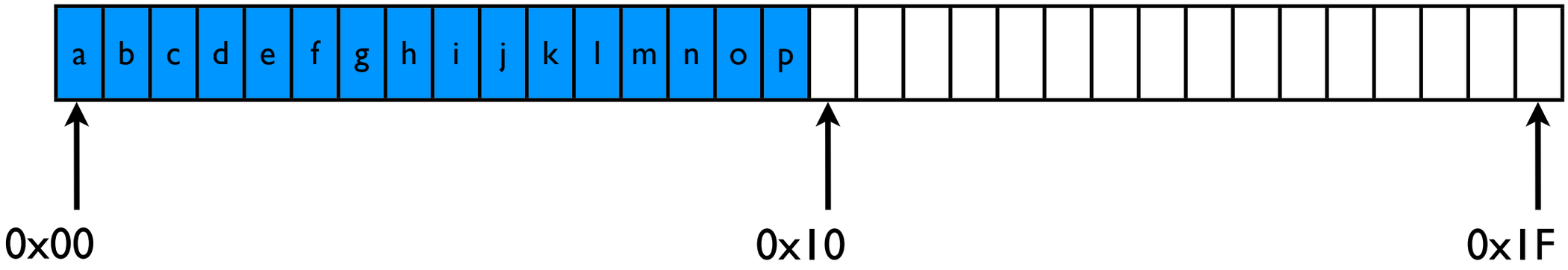
We have an 2D char array stored here!






```
#define ROWS 4
#define COLS 4

char char_arr[ROWS][COLS];
```

```
char char_arr[ROWS][COLS];
```

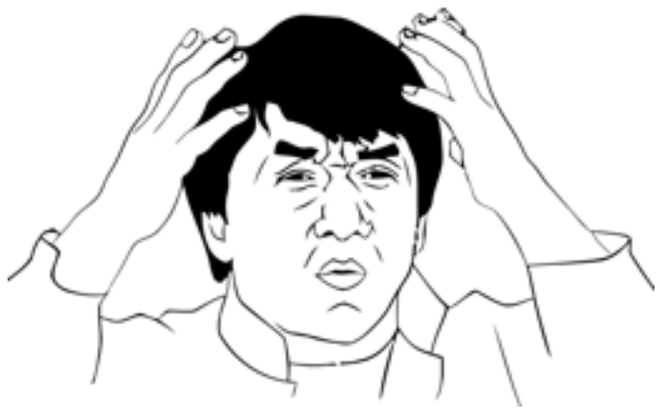


Scenario:

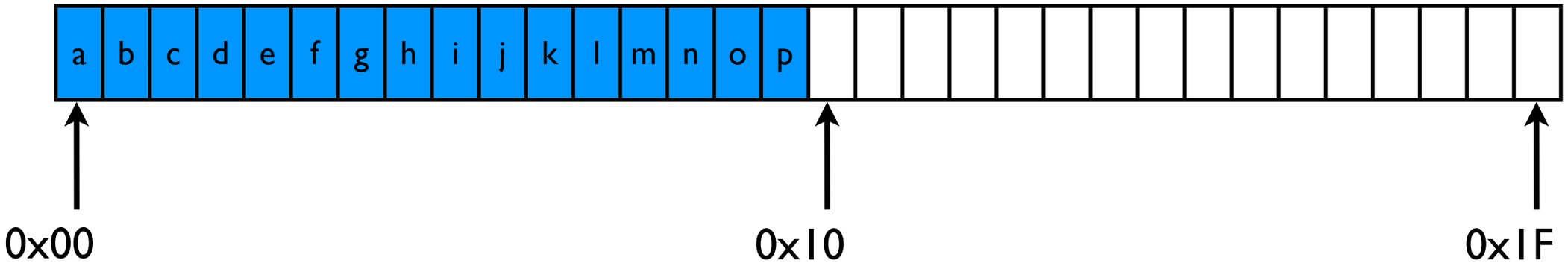
Someone who can't even read binary memory content wanna know whats in the array, so we have to print every character for him.

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```

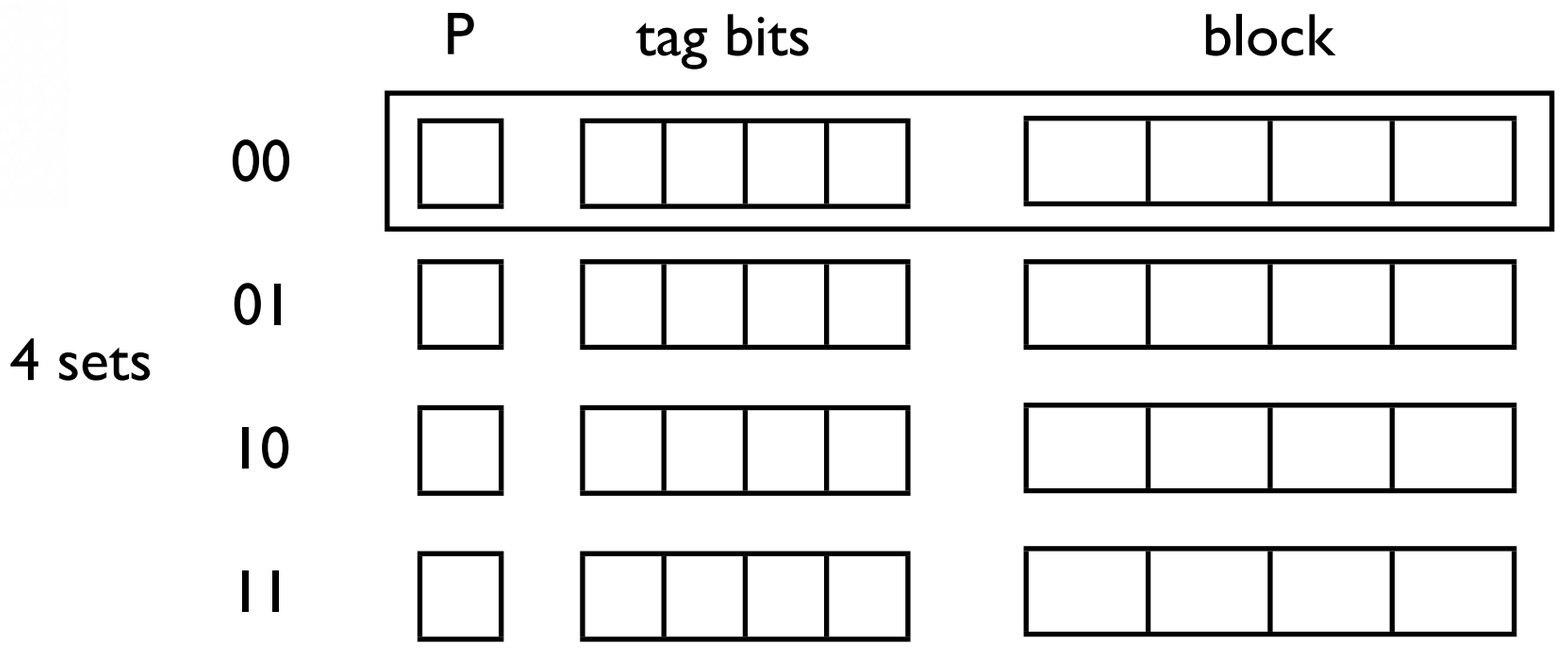
16 memory accesses!



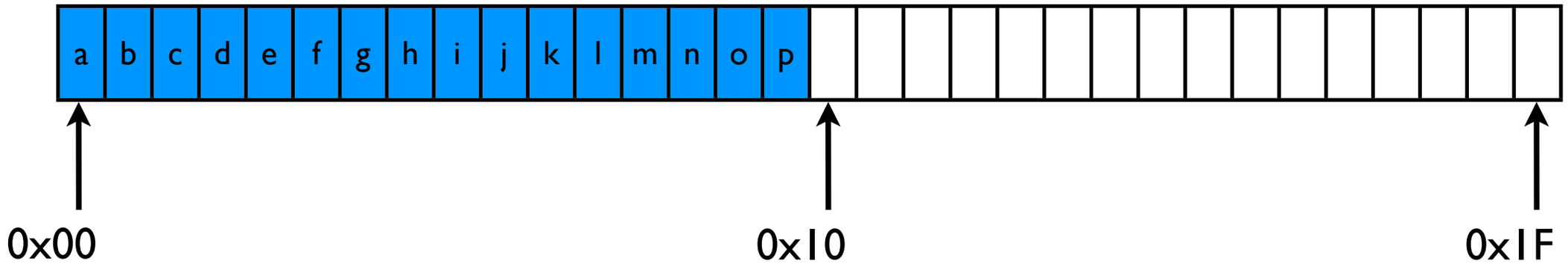
```
char char_arr[ROWS][COLS];
```



We have a direct-mapped cache!

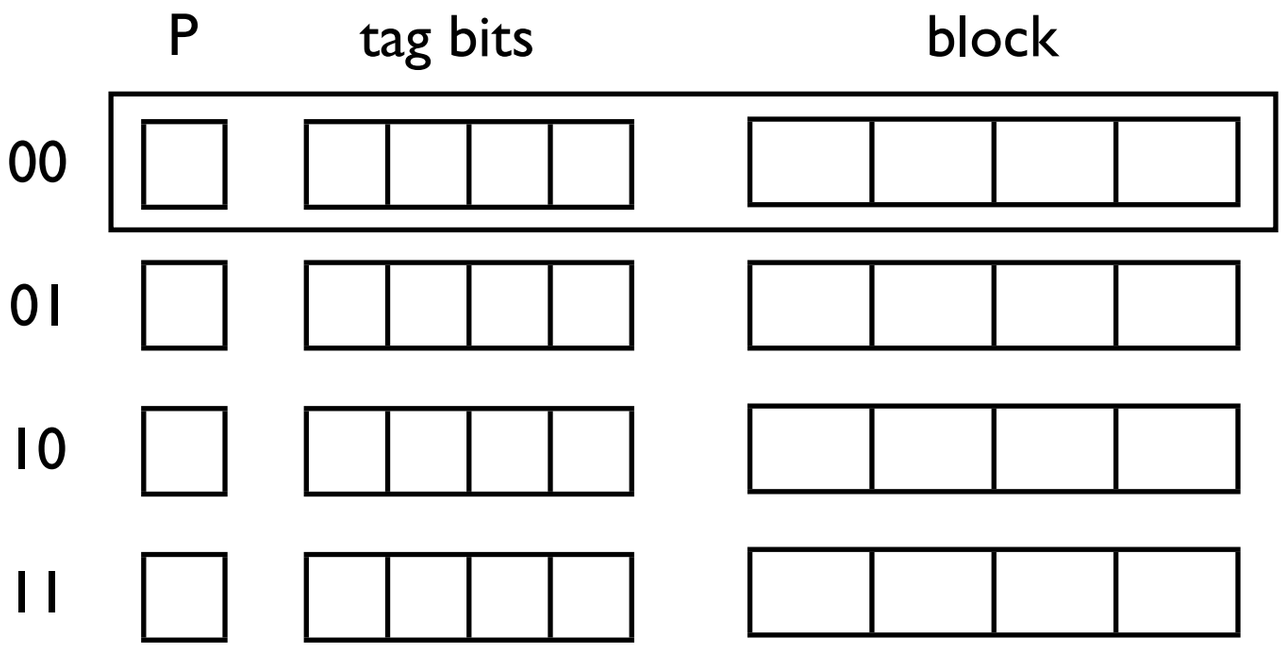


```
char char_arr[ROWS][COLS];
```

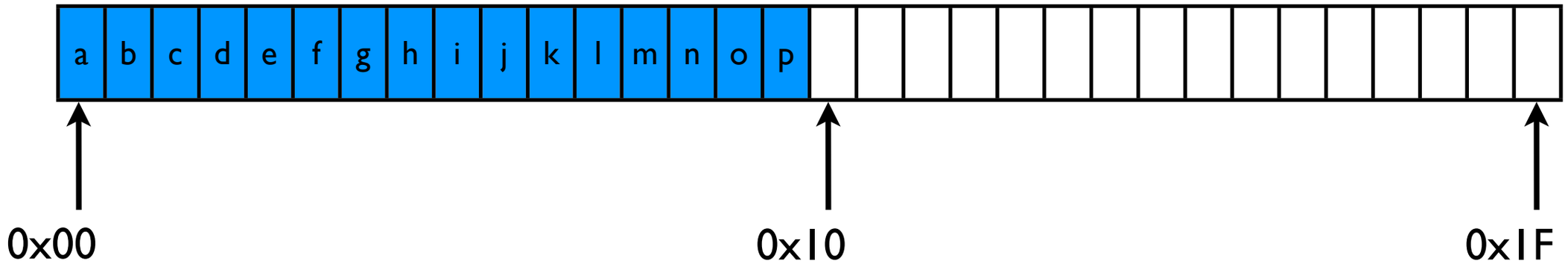


Access char_arr[0][0], addr: 0000 0000_2

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```



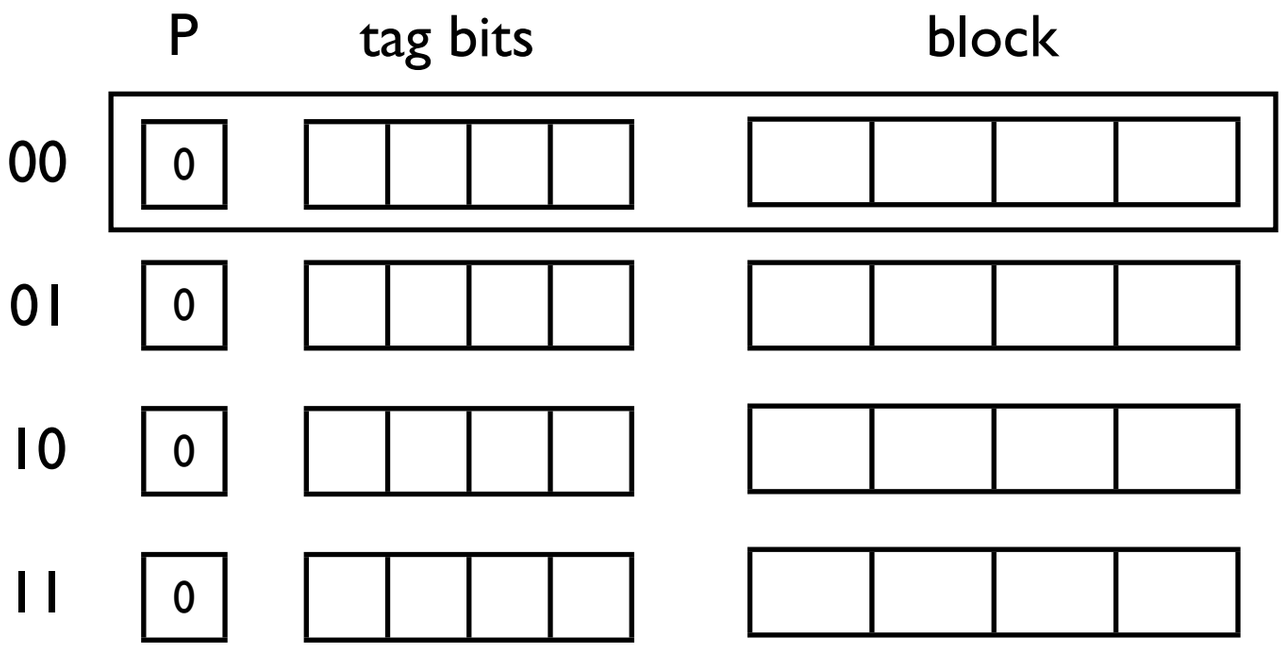
```
char char_arr[ROWS][COLS];
```



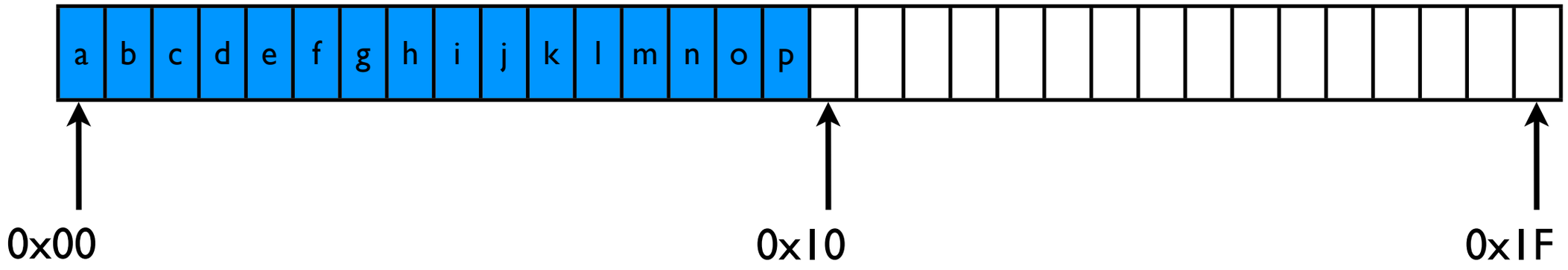
Access char_arr[0][0], addr: 0000 0000_2

set bits: 00, block bits: 00

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```



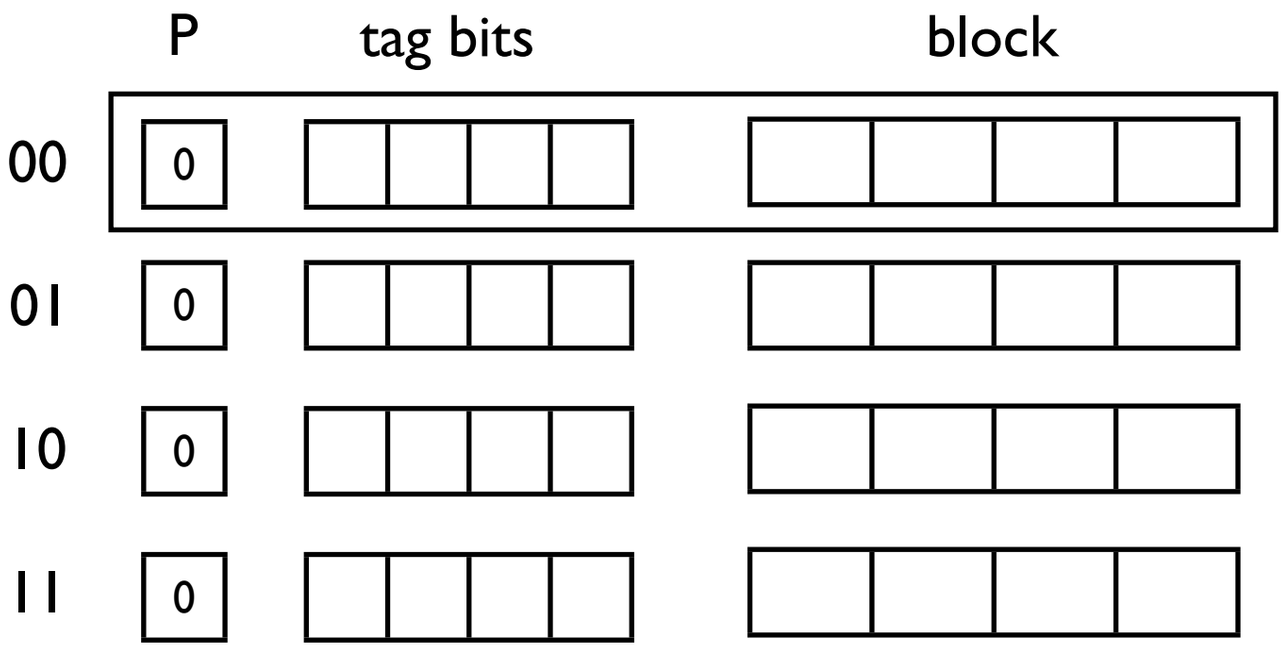
```
char char_arr[ROWS][COLS];
```



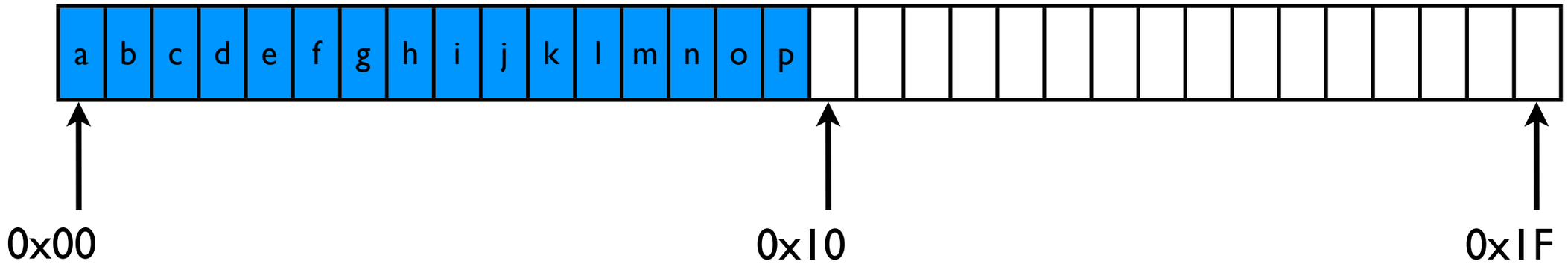
Access char_arr[0][0], addr: 0000 0000_2

set bits: 00, block bits: 00

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```



```
char char_arr[ROWS][COLS];
```

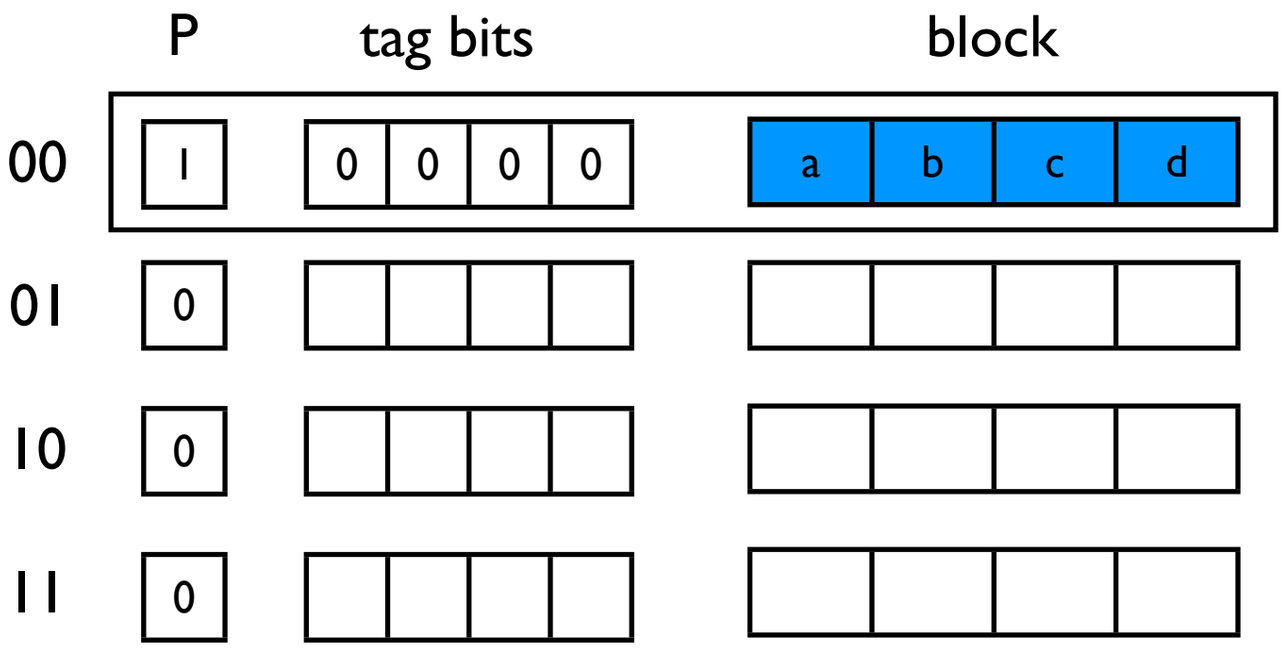


Access char_arr[0][0], addr: 0000 0000_2

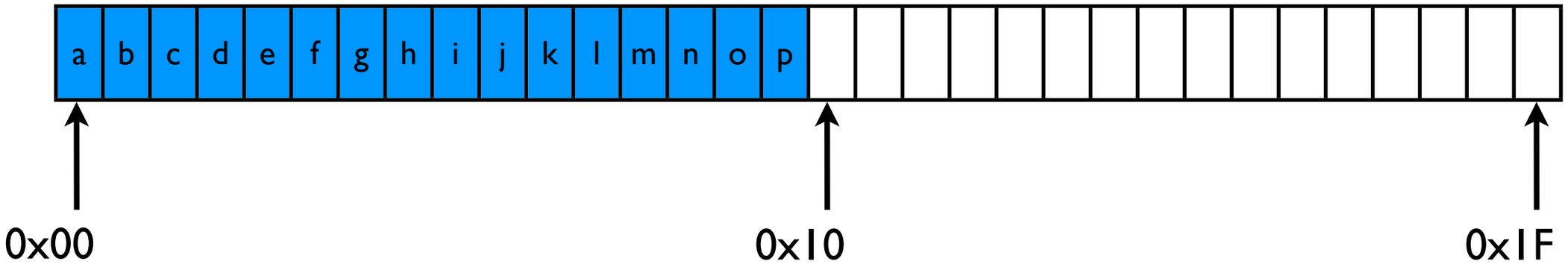
set bits: 00, block bits: 00

Cache Miss

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```

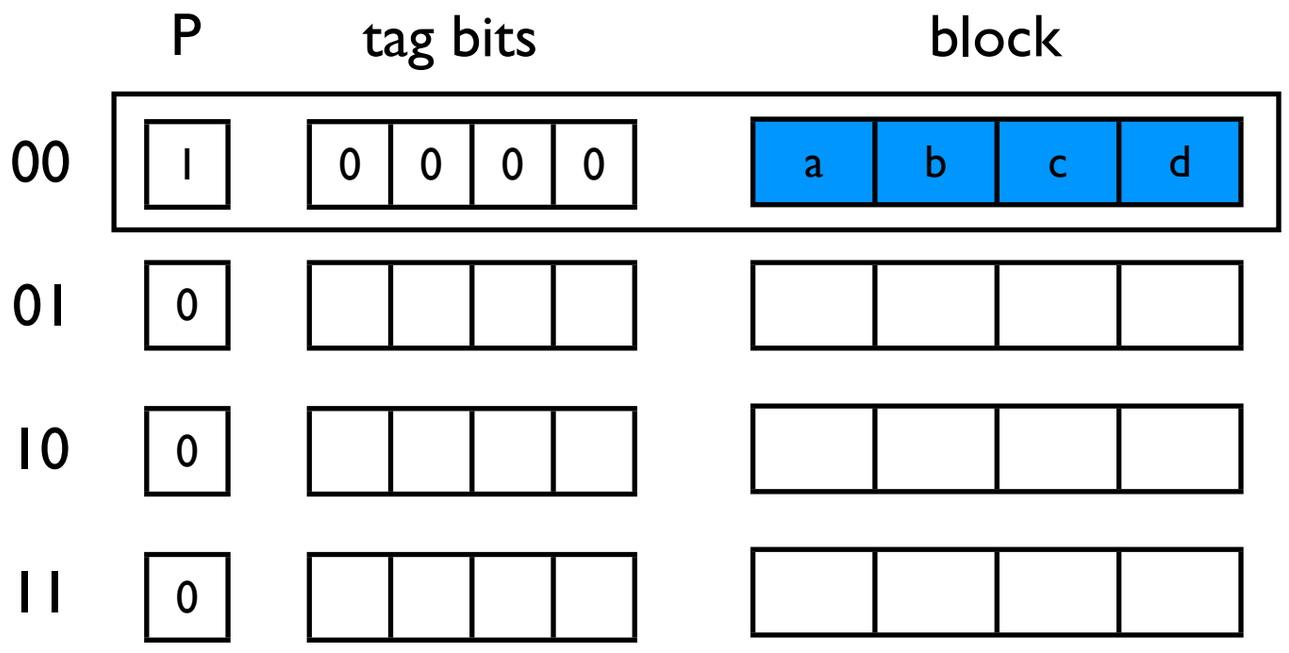


```
char char_arr[ROWS][COLS];
```

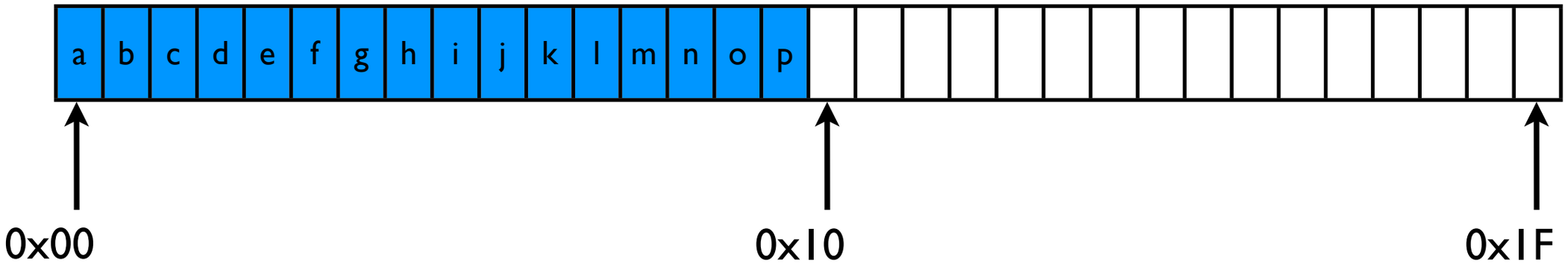


Access char_arr[0][1], addr: 0000 0001$_2$

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```



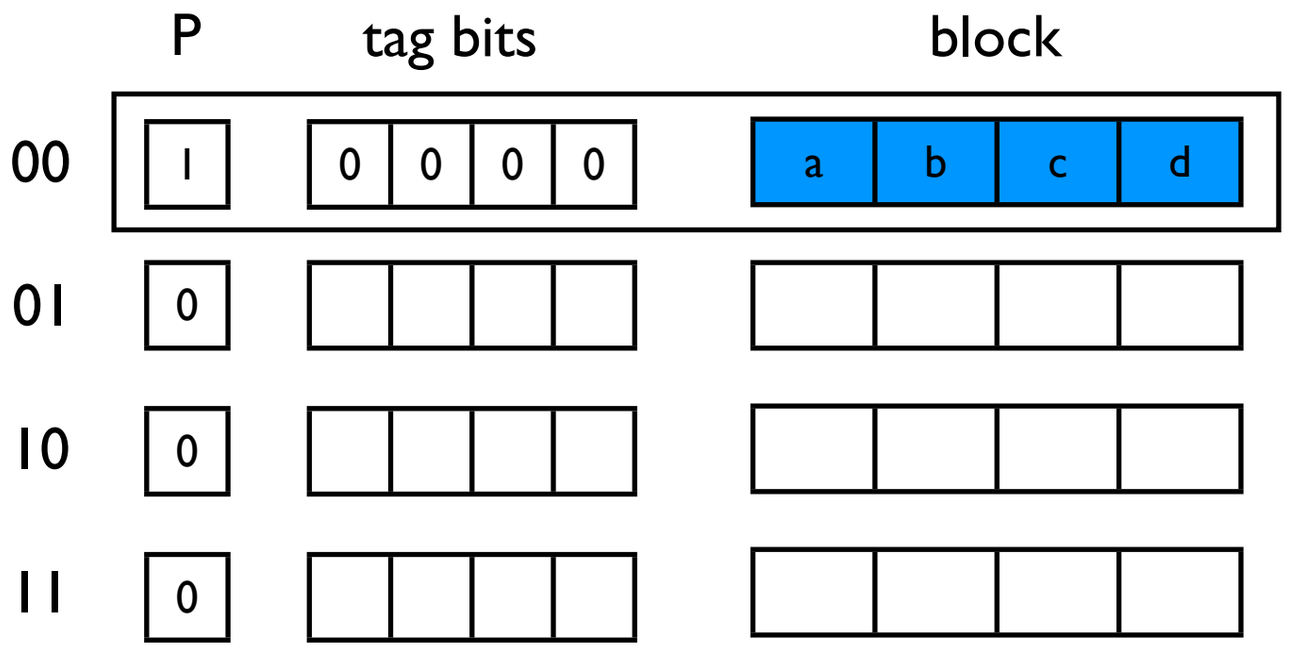
```
char char_arr[ROWS][COLS];
```



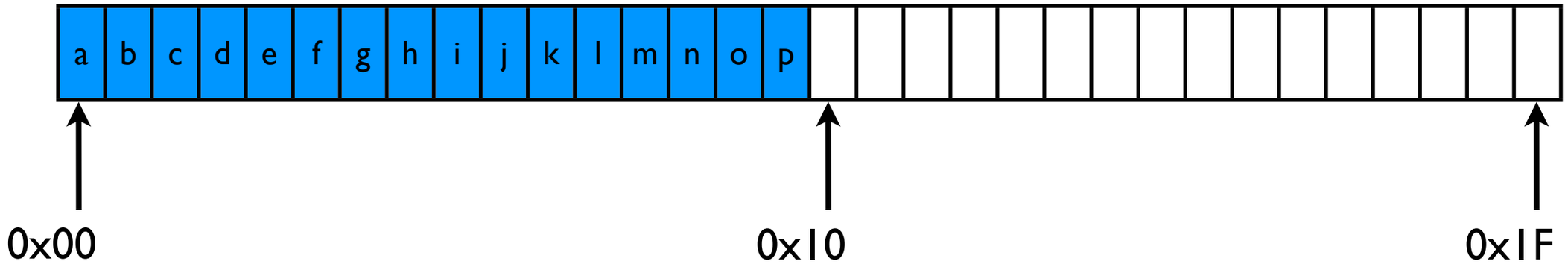
Access char_arr[0][1], addr: 0000 0001$_2$

set bits: 00, block bits: 01

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```



```
char char_arr[ROWS][COLS];
```

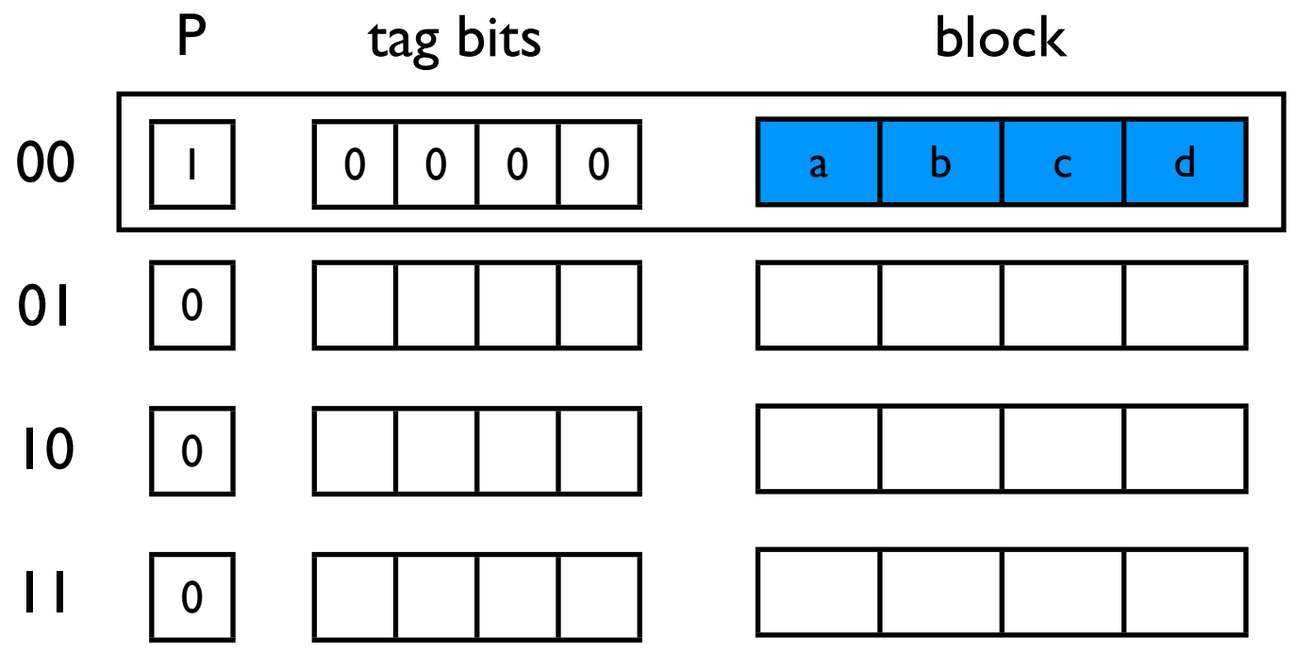


Access char_arr[0][1], addr: 0000 0001$_2$

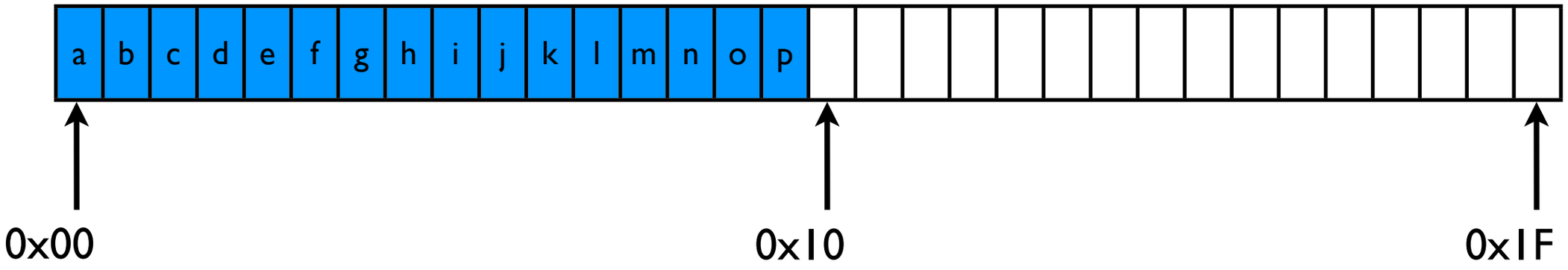
set bits: 00, block bits: 01

Cache Hit!

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```

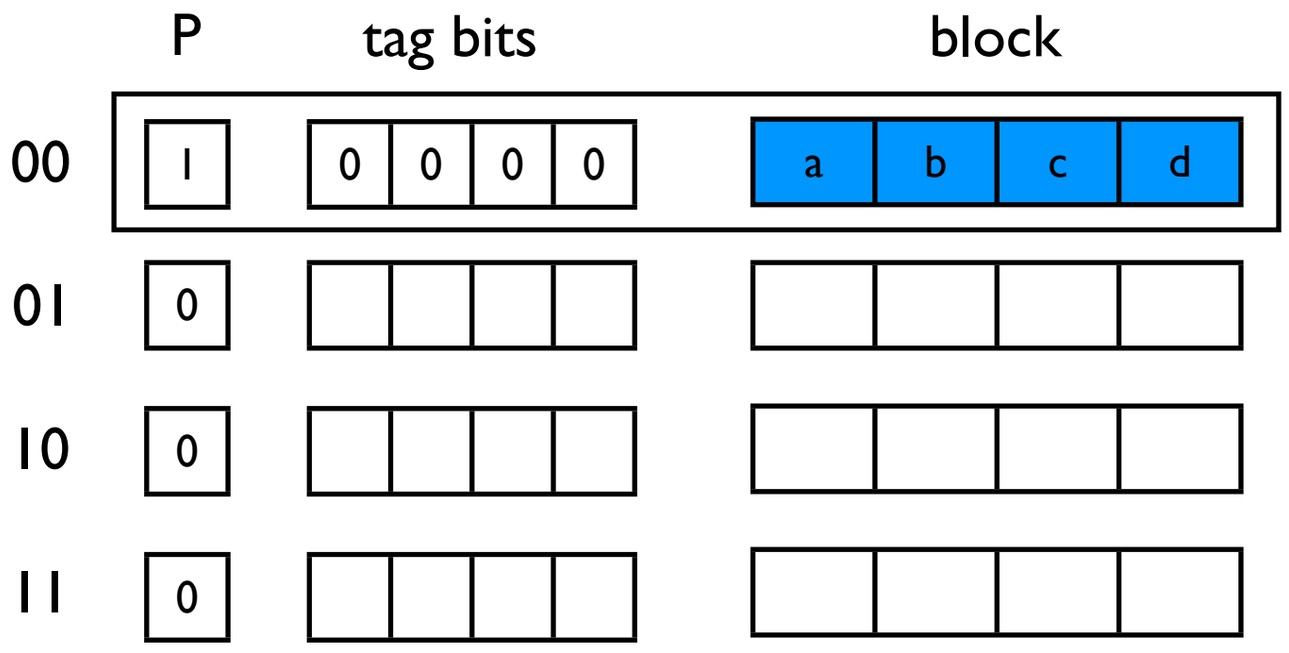


```
char char_arr[ROWS][COLS];
```

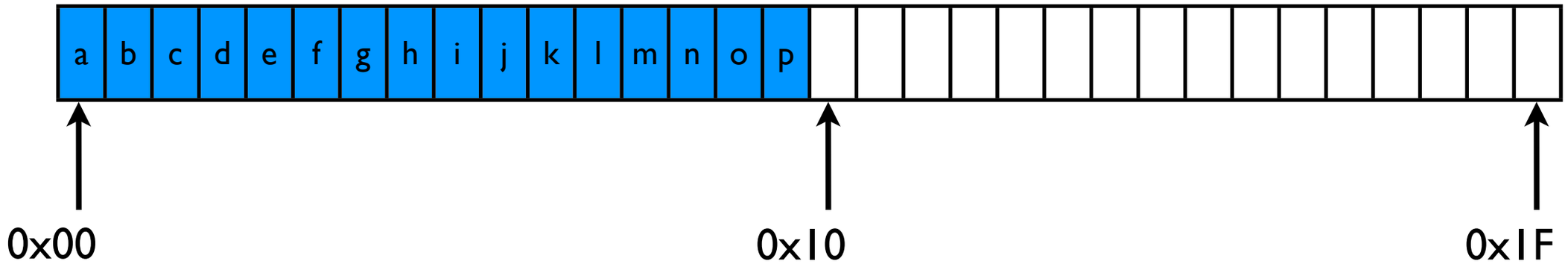


You should be able to do this on your own

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```



```
char char_arr[ROWS][COLS];
```

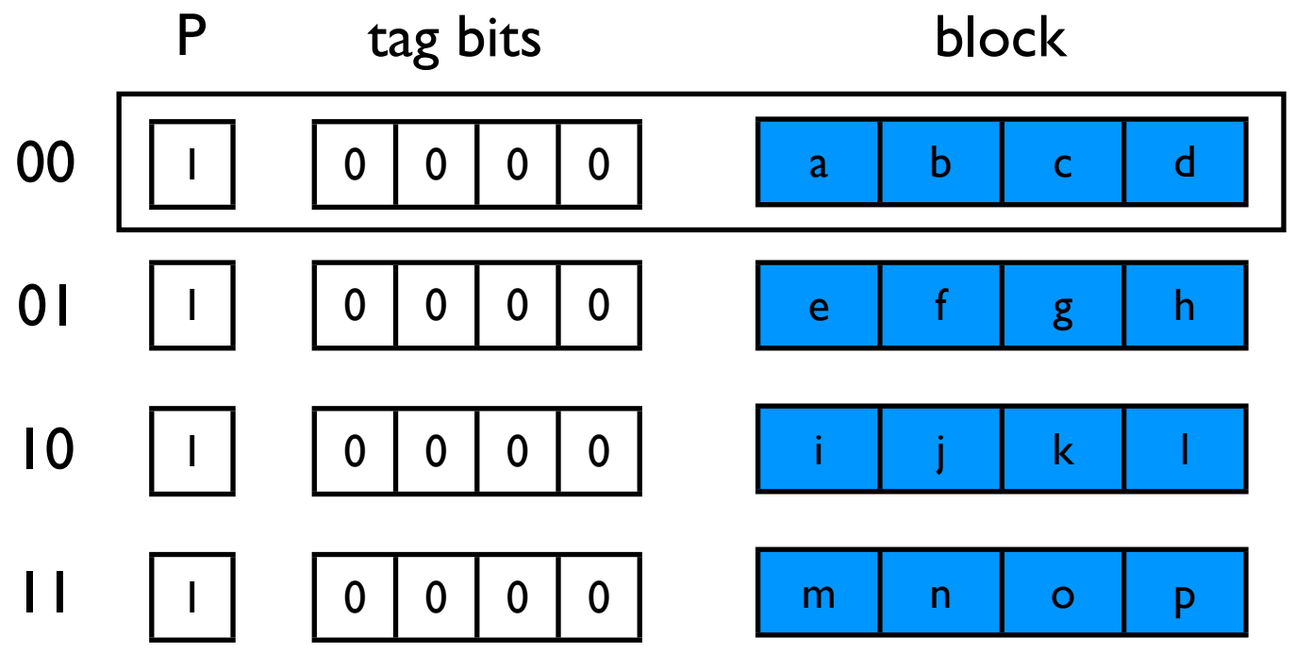


Access char_arr[3][3], addr: 0000 1111$_2$

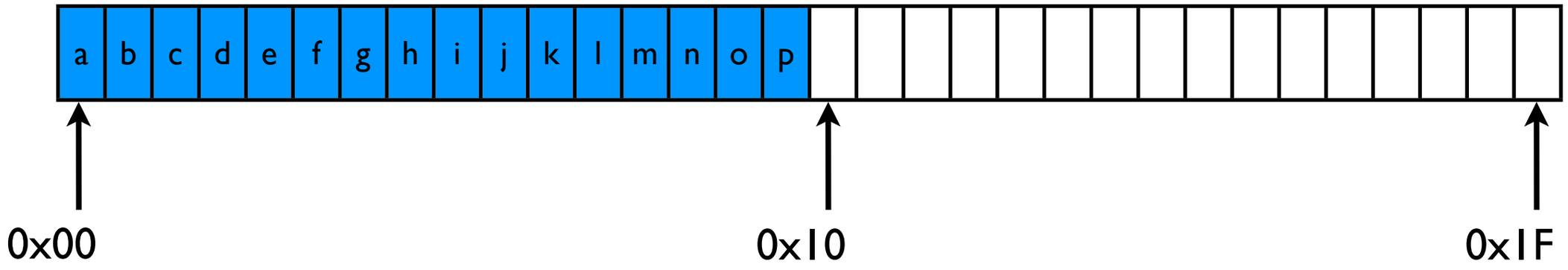
set bits: 11, block bits: 11

Cache Hit!

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```

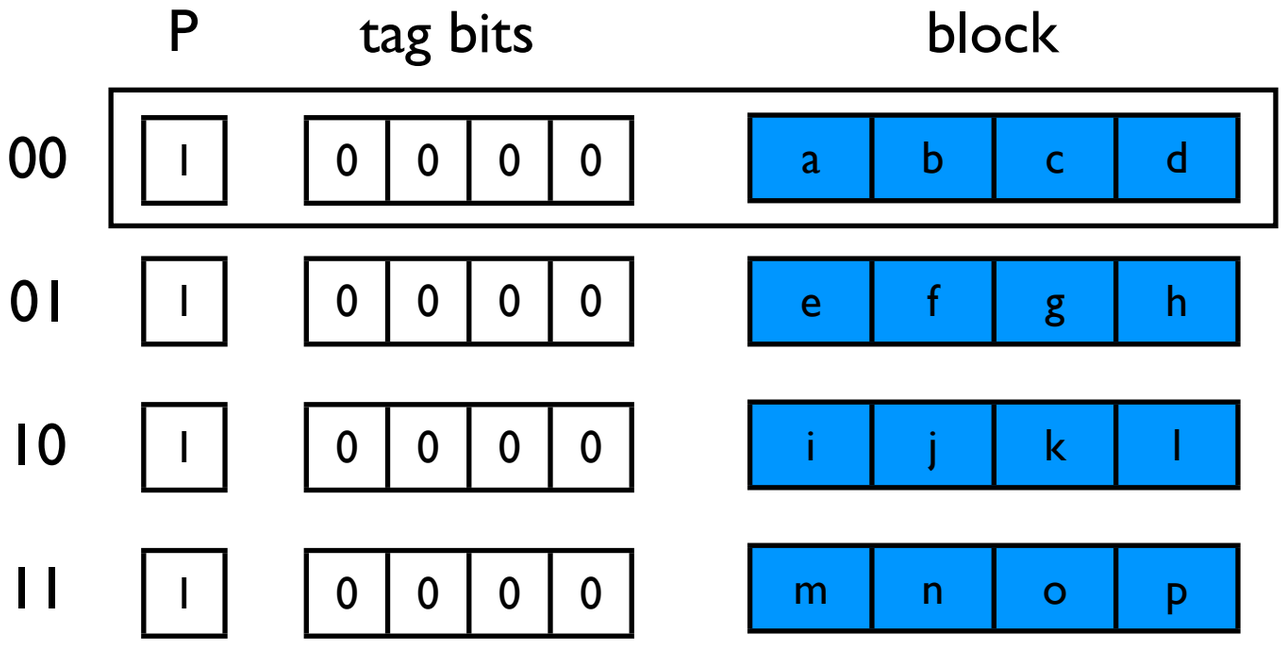


```
char char_arr[ROWS][COLS];
```

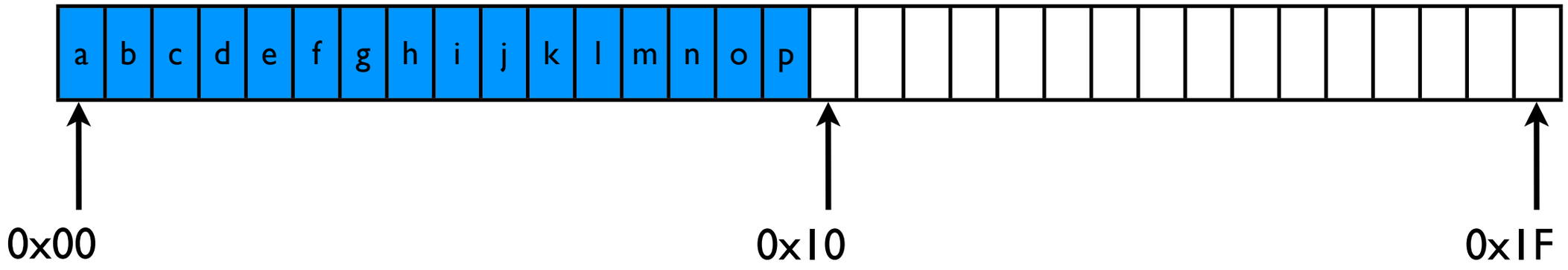


What's our cache miss rate?

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```

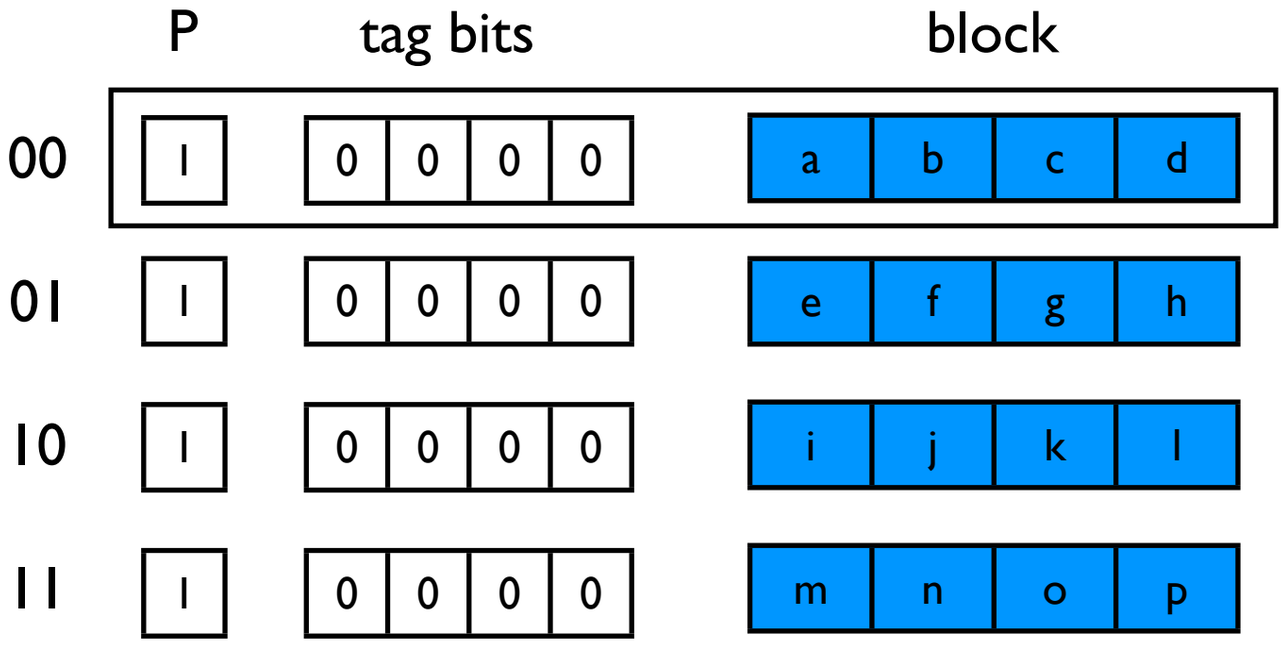


```
char char_arr[ROWS][COLS];
```

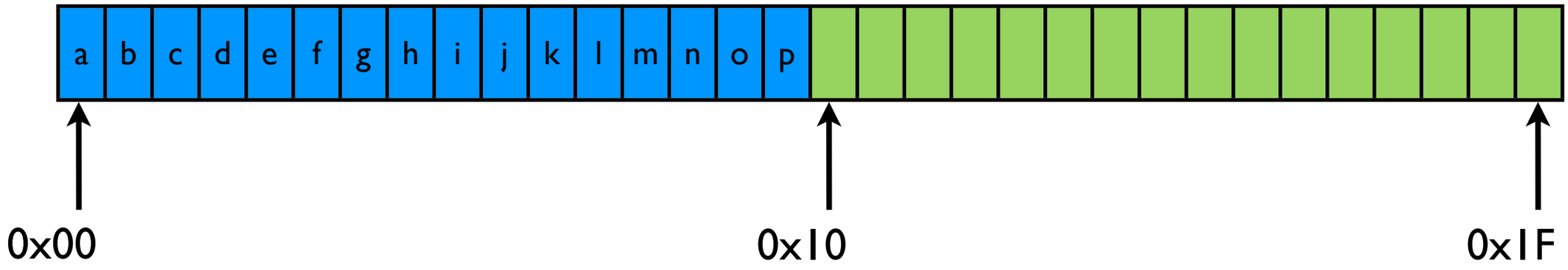


What's our cache miss rate? 1/4

```
char char_arr[ROWS][COLS];
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
      puts(char_arr[i][j]);
   }
}
```

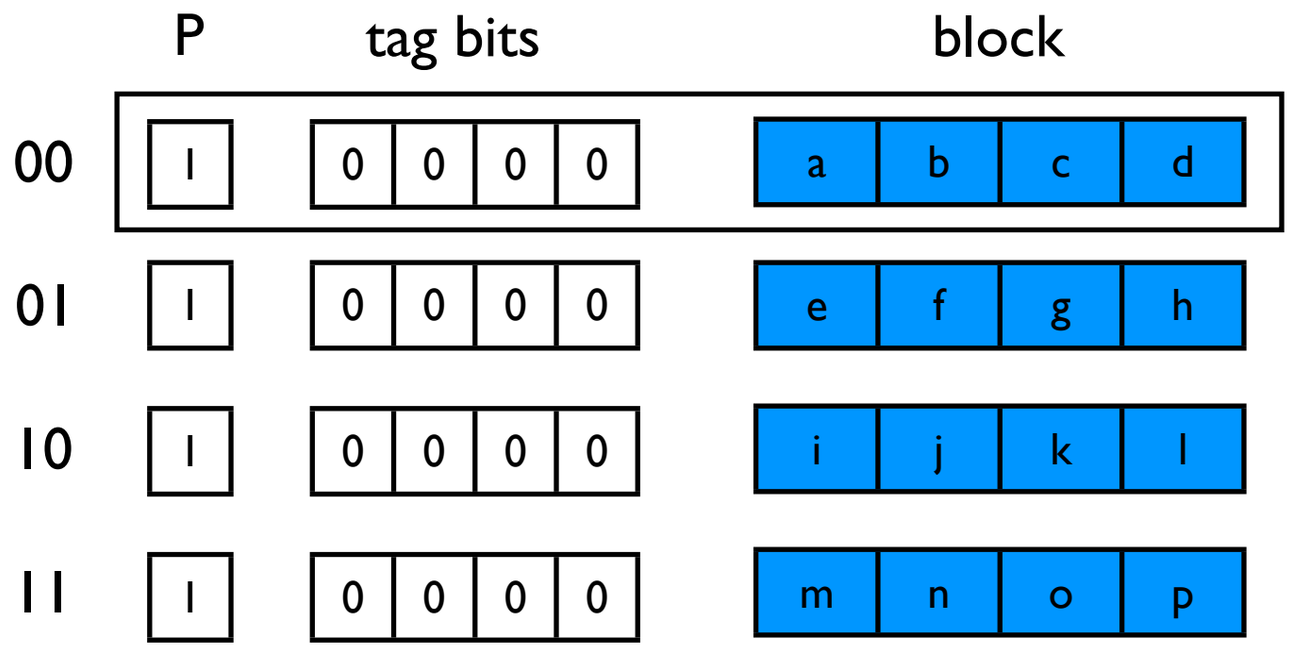


```
char char_arr[ROWS][COLS];
char to_arr[ROWS][COLS];
```

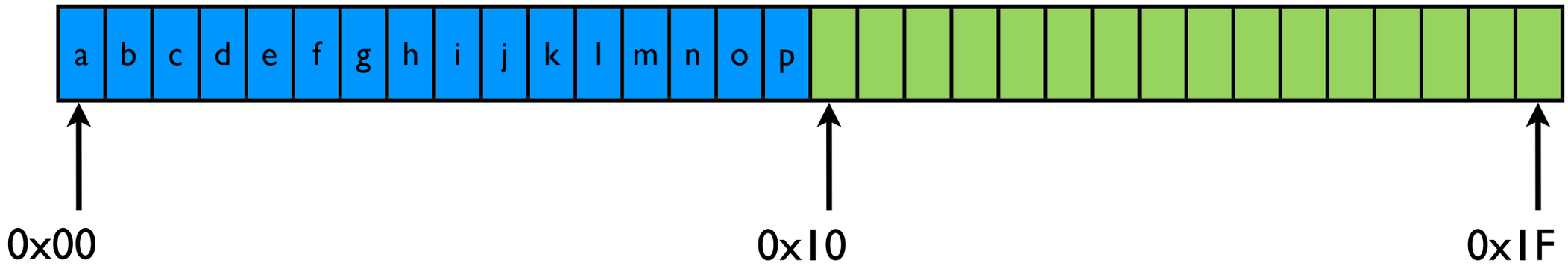


What if we were to copy the char_arr to to_arr?

```
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
       to_arr[i][j] = char_arr[i][j];
   }
}
```



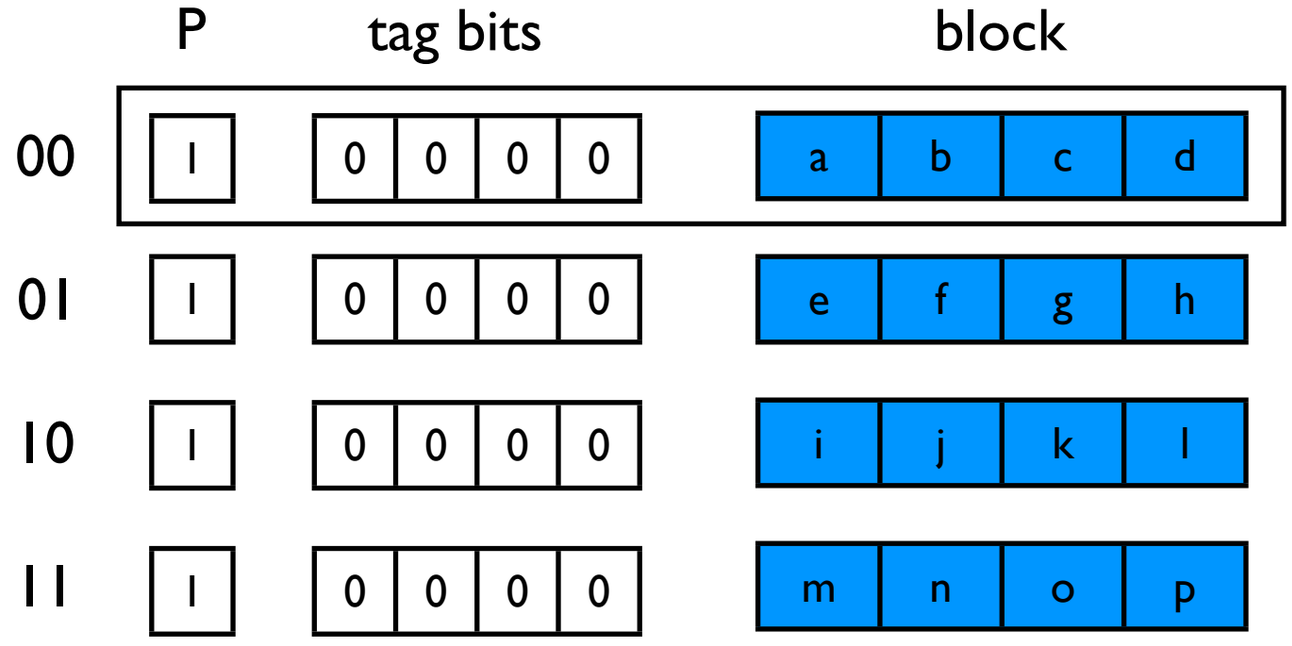
```
char char_arr[ROWS][COLS];
char to_arr[ROWS][COLS];
```



What if we were to copy the char_arr to to_arr?

There'll be tons of evictions if we use this code!

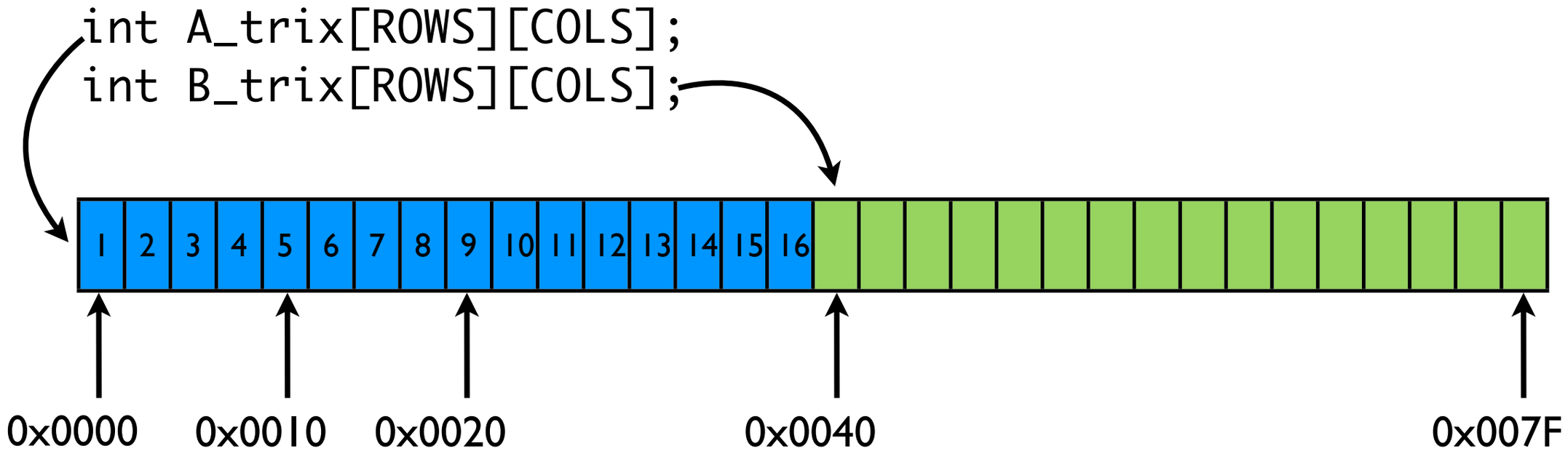
```
int i, j;

for(i = 0; i < ROWS; i++ ) {
   for(j = 0; j < COLS; j++ ) {
       to_arr[i][j] = char_arr[i][j];
   }
}
```



Use cache wisely

- Avoid Thrashing!
- Registers?
- Blocking!

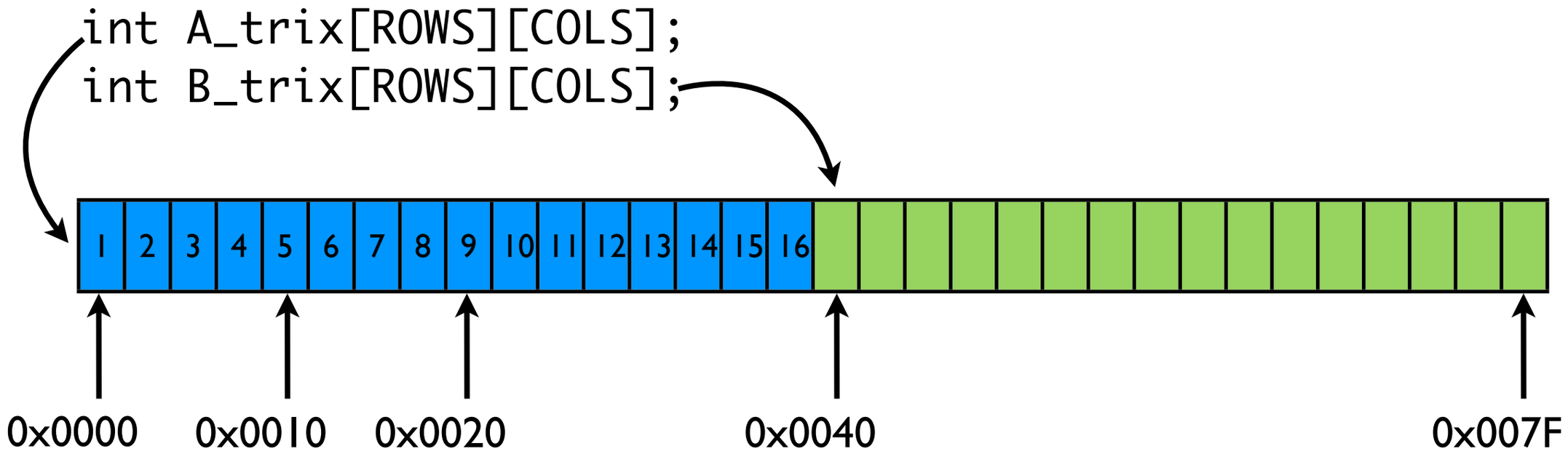
Efficient Matrix Transpose

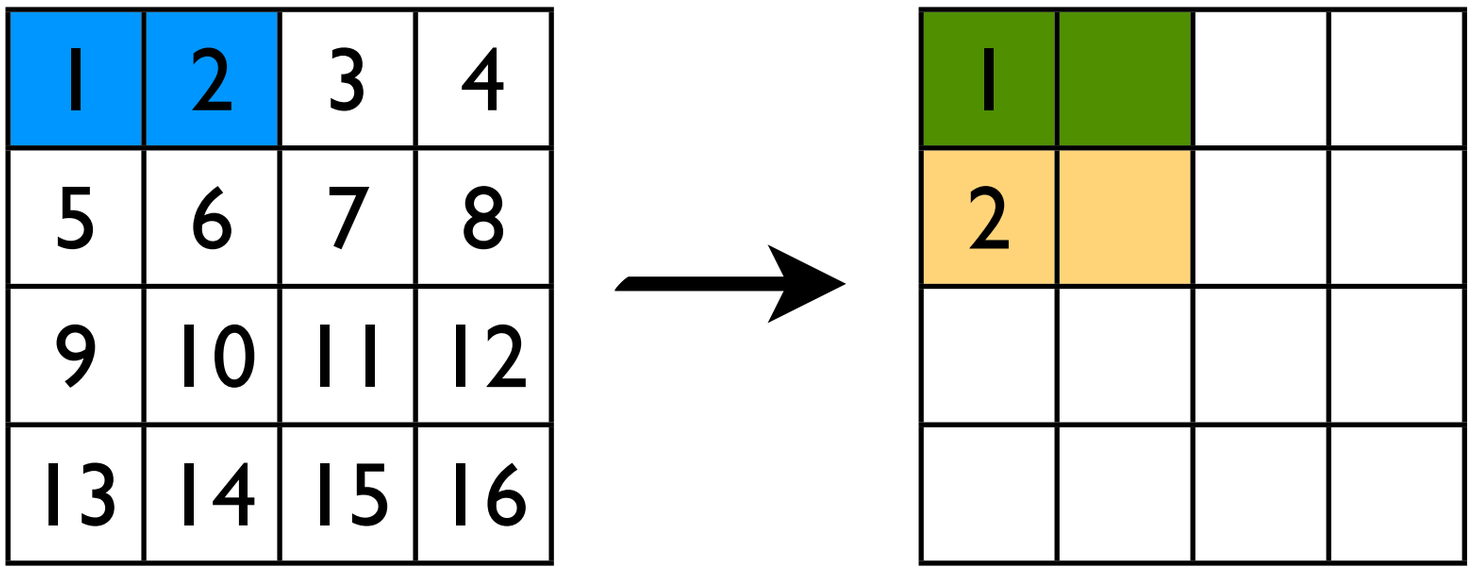


1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

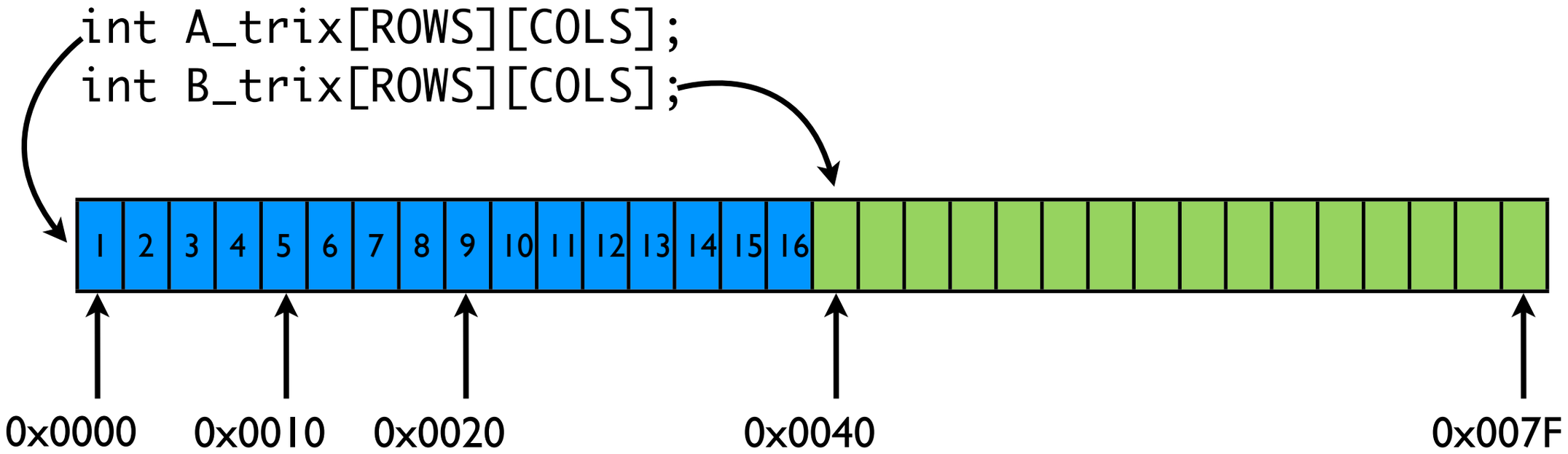
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Efficient Matrix Transpose





Efficient Matrix Transpose



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1			
2			

What do we do next? Read 3 & 4? or 5 & 6?

Blocking

- What inspiration do you get from previous slide?
  - Divide matrix into sub-matrices
  - This is called blocking (CSAPP2e p.629)
  - Size of sub-matrix depends on cache block size, cache size, input matrix size
  - Try different sub-matrix sizes
- We hope you invent more tricks to reduce the number of misses!

Linking Stuff

- Where did your code go?
- Resolving global symbols
- Global Variables

Relocating Code and Data

Global

```
int buf[2] = {1, 2};

int main()
{
   swap();
   return 0;
}
```

main.c

External

Linker knows nothing of temp

Global

External

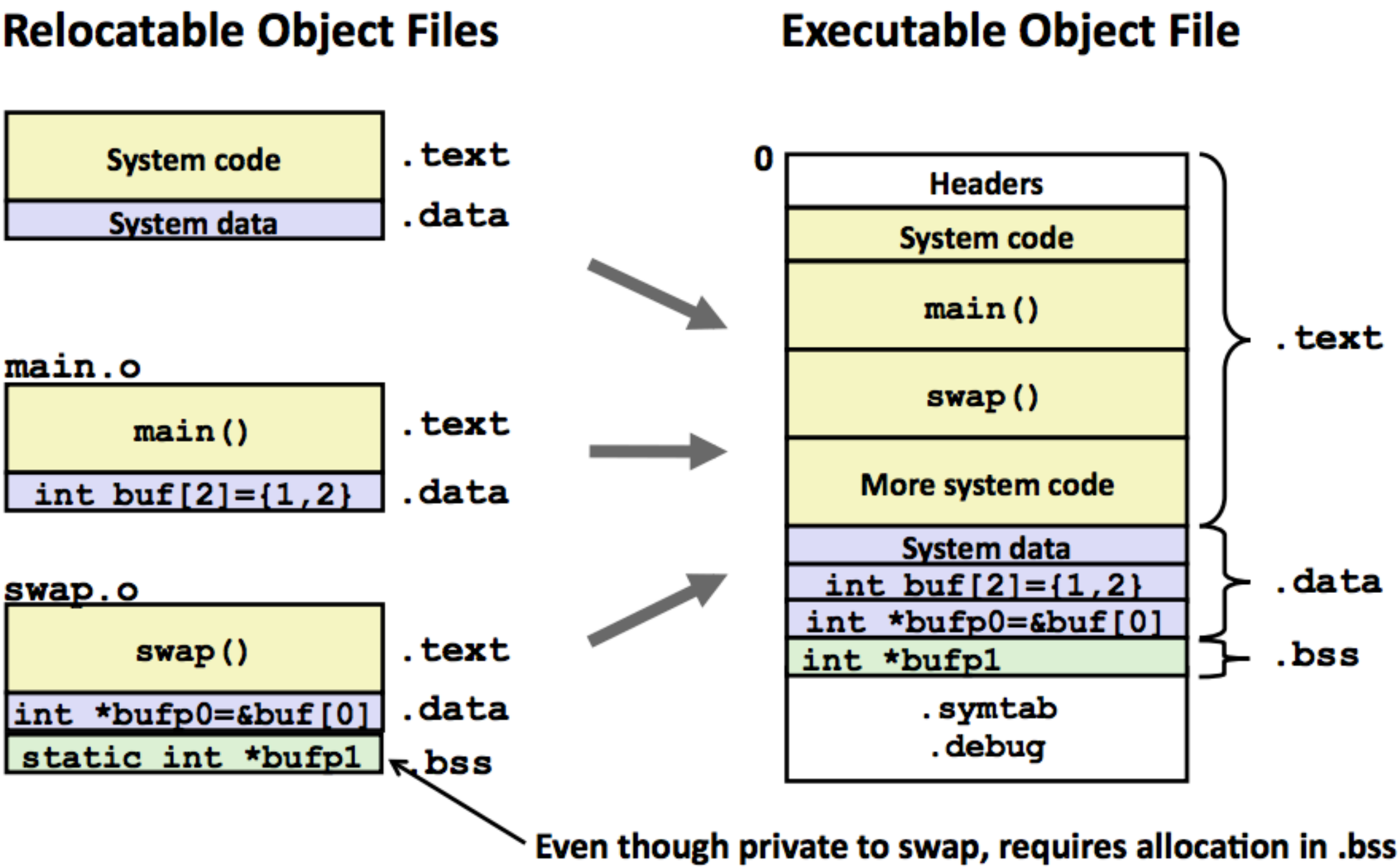
Local

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

Global

swap.c

Relocating Code and Data



Relocating Code and Data

In the main.o file:

```
6:    e8 fc ff ff ff          call   7 <main+0x7>    swap();
7: R_386_PC32 swap            relocation entry
```

Relocation Entry:

r consists of three fields:

```
r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

Relocating Code and Data

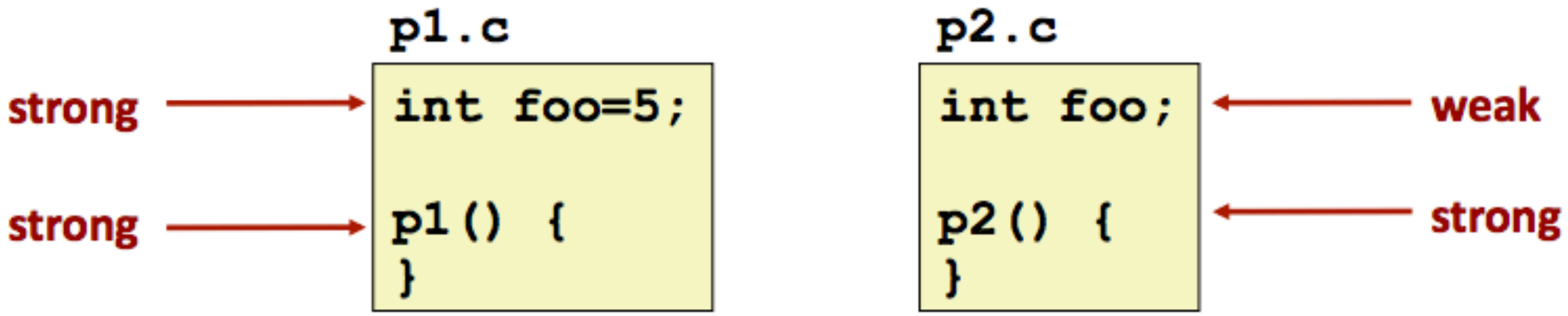
```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_386_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_386_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14      }
15  }
```

Figure 7.9 **Relocation algorithm.**

```
80483ba:   e8 09 00 00 00          call   80483c8 <swap>
```

Strong and Weak Symbols

- **Program symbols are either strong or weak**
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



Linker's symbol rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error

- **Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol

- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`

Global Variables

- Avoid if you can
- Otherwise
  - Use static if you can
  - Initialize if you define a global variable
  - Use extern if you use external global variable

Reference

- Some materials are shamelessly stolen from previous year's lecture slides

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f11/www/lectures/12-linking.pdf

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f11/www/recitations/rec7.pdf

Questions?

p.s. Go design better algorithms for your cache lab!
It's always awesome to be the best :-)