



Аллокация памяти

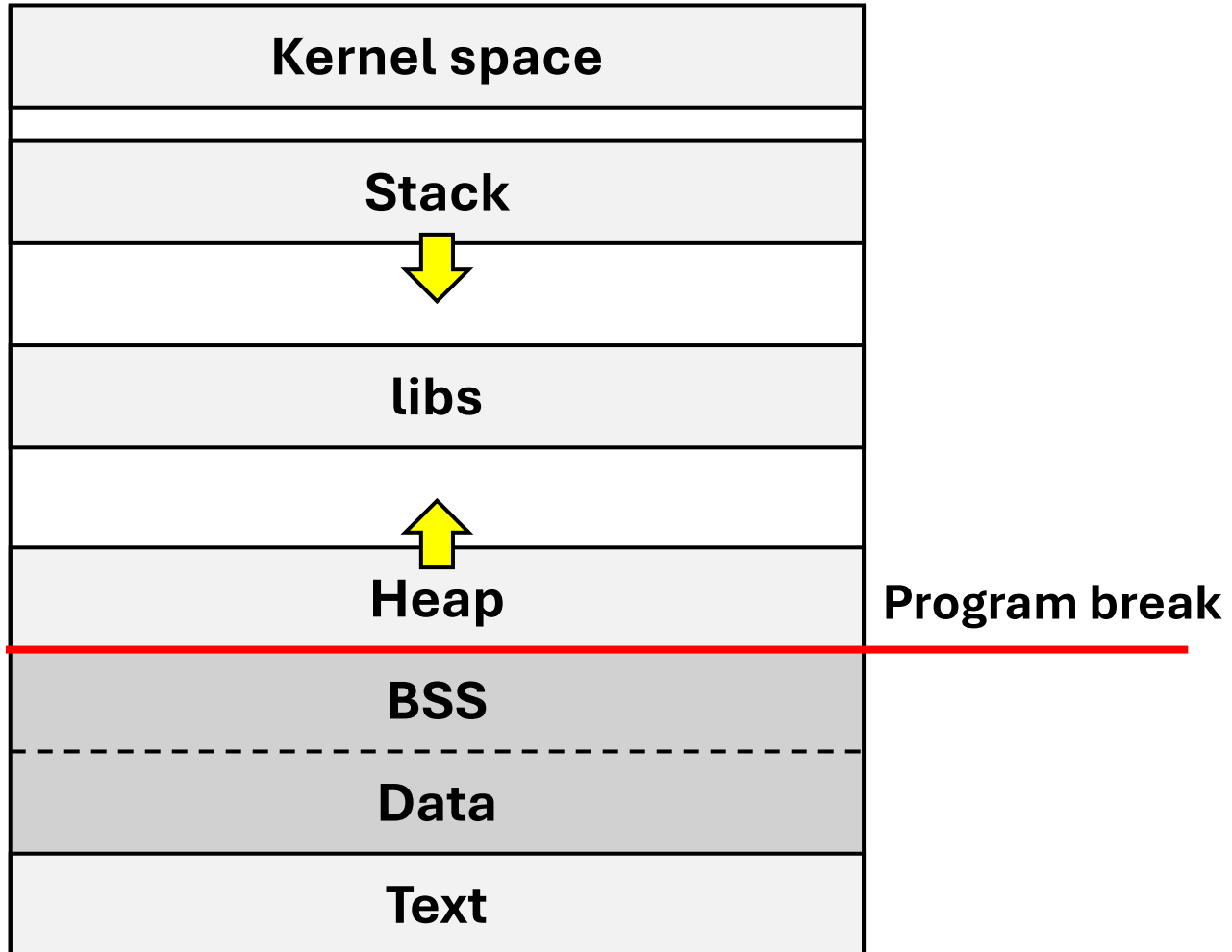
Системные вызовы ОС *Linux* для работы с памятью:

brk()/sbrk() — используется для изменения объема памяти, выделенной процессу.

mmap() – позволяет отображать память из любого места процесса



Аллокация памяти



С помощью системного
вызова ***sbrk()*** возможно
изменять положение
Program break

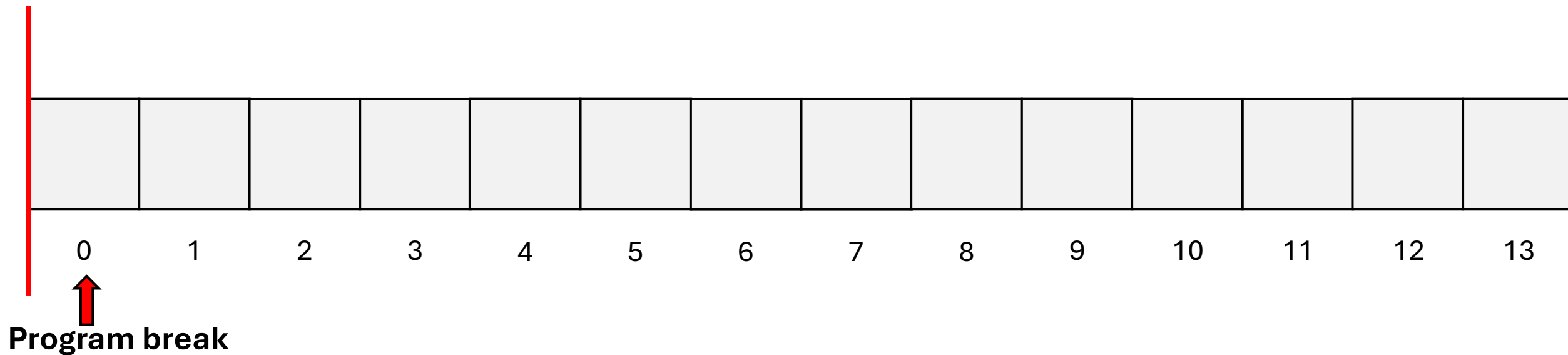


A horizontal array of 14 light gray rectangular cells, each representing an element in memory. Below each cell is its corresponding index, ranging from 0 to 13. A red vertical line is drawn at the left edge of the first cell (index 0). A red arrow points upwards from the text 'Program break' to the red line.

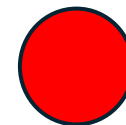


Аллокация памяти

```
int* x = (int*)malloc(N);
```



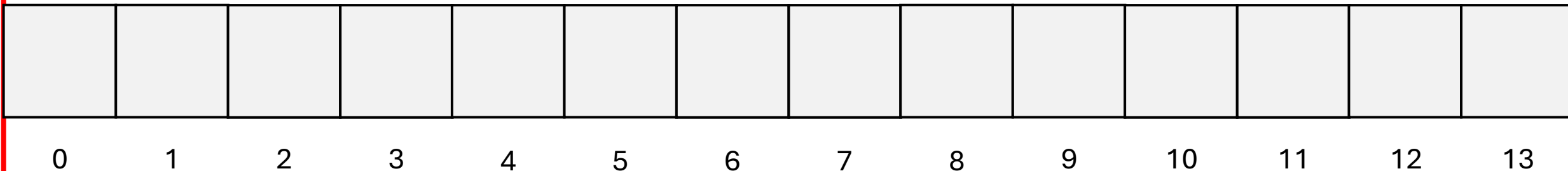
```
struct mem_control_block {  
    int is_available;  
    long size;  
};
```





Аллокация памяти

```
int* x = (int*)malloc(3);
```



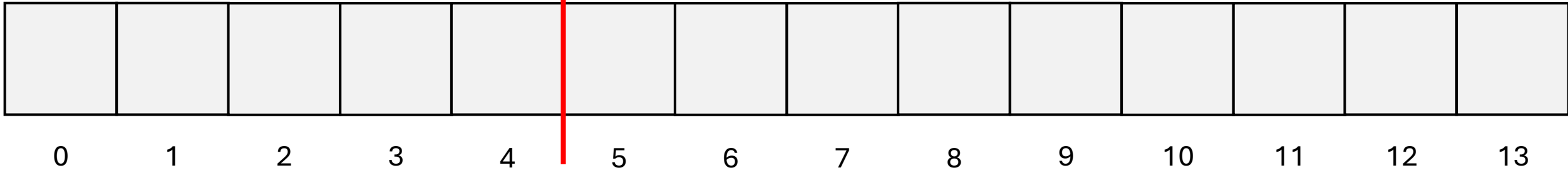
Program break
current_location
managed_memory_start
last_valid_address

<code>last_valid_address = sbrk(0);</code>	<code>last_valid_address = 0</code>
<code>managed_memory_start = last_valid_address;</code>	<code>managed_memory_start = 0</code>
<code>current_location = managed_memory_start;</code>	<code>current_location = 0</code>
<code>numbytes = numbytes + sizeof(struct mem_control_block);</code>	<code>numbytes = 3 + 2 = 5</code>



Аллокация памяти

```
int* x = (int*)malloc(3);
```



current_location

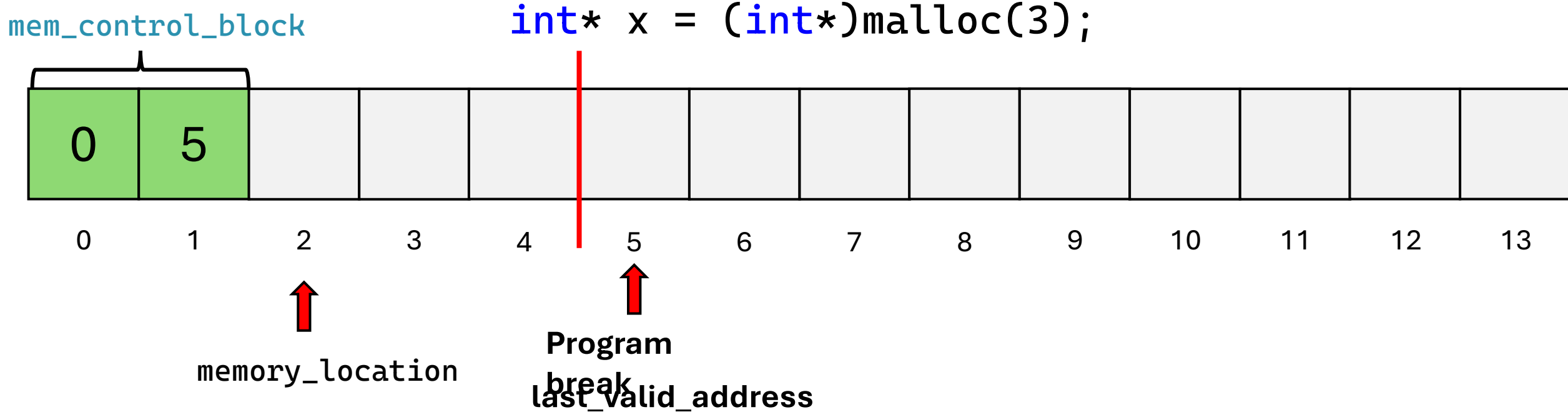
managed_memory_start

Program
break
last_valid_address

<code>sbrk(numbytes);</code>	<code>sbrk(5);</code>
<code>memory_location = last_valid_address;</code>	<code>memory_location = 0</code>
<code>last_valid_address = last_valid_address + numbytes;</code>	<code>last_valid_address = 0 + 5</code>
<code>current_location_mcb = memory_location;</code>	<code>current_location_mcb = 0</code>
<code>current_location_mcb->is_available = 0;</code>	<code>clm->is_available = 0</code>
<code>current_location_mcb->size = numbytes;</code>	<code>clm->size = 5</code>



Аллокация памяти



```
memory_location = memory_location + sizeof(struct mem_control_block);
```

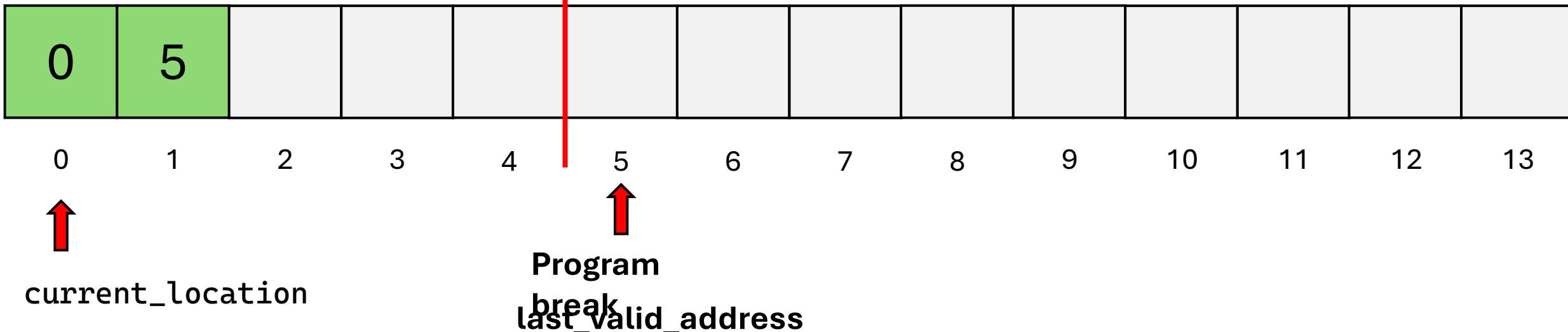
```
memory_location = 0 + 2 = 2;
```

```
return memory_location;
```



Аллокация памяти

```
int* y = (int*)malloc(5);
```



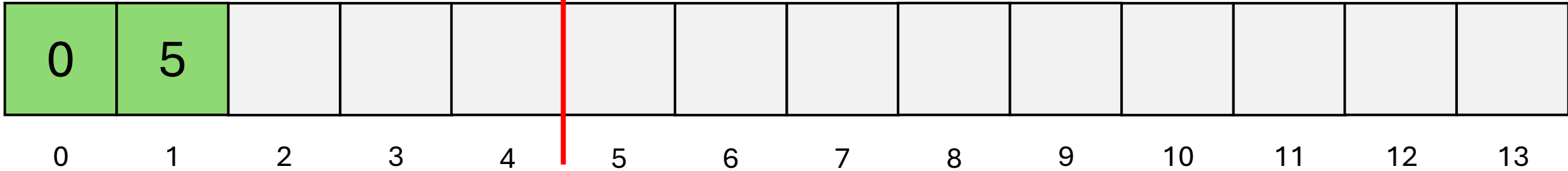
```
while (current_location != last_valid_address){  
    if (current_location_mcb->is_available) {//выделяем память}  
        current_location = current_location + current_location_mcb->size;  
    }  
}
```

Ячейка занята, поэтому двигаемся дальше $\text{current_location} = 0 + 5 = 5$



Аллокация памяти

```
int* y = (int*)malloc(5);
```



Program
break
last_valid_address
current_location
memory_location

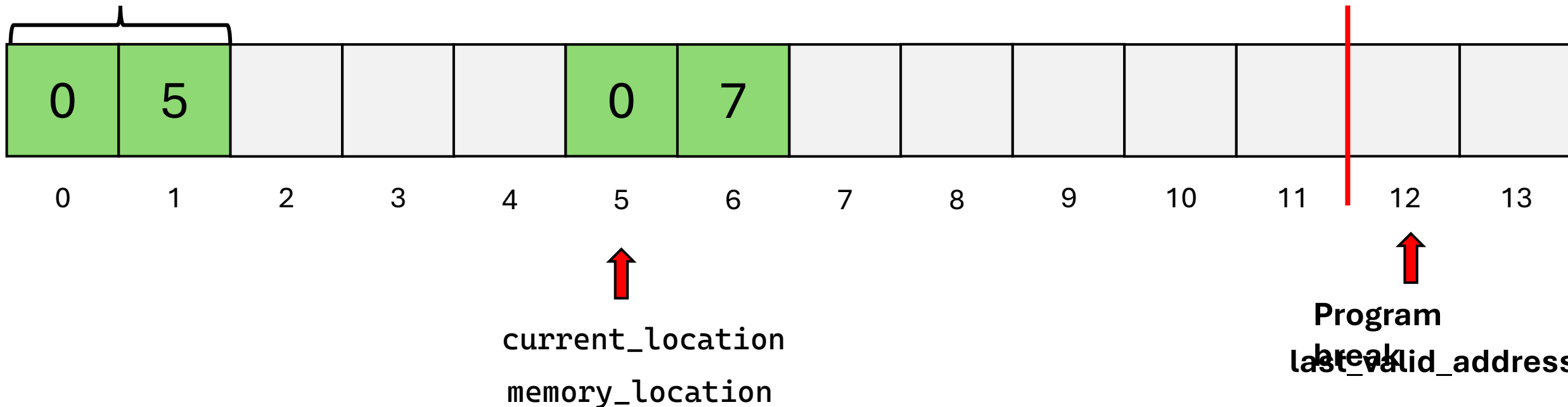
Цикл закончился -> снова необходимо просить памяти



Аллокация памяти

mem_control_block

```
int* y = (int*)malloc(5);
```



```
memory_location = memory_location + sizeof(struct mem_control_block);
```

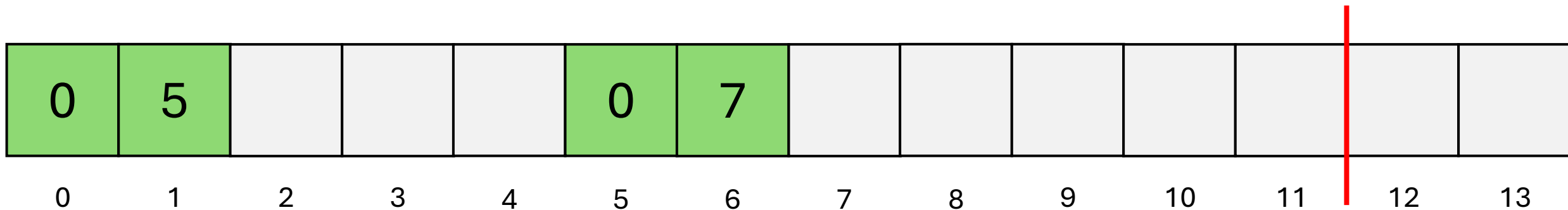
```
memory_location = 5 + 2 = 7;
```

```
return memory_location;
```



Аллокация памяти

`free(x);`



Адрес начала выделенной памяти

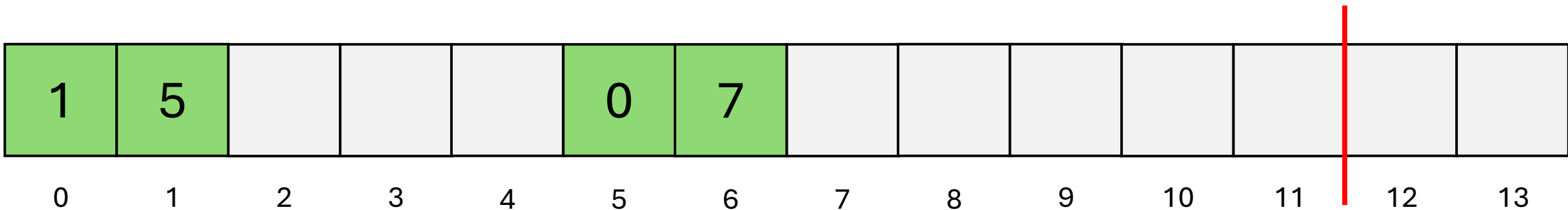
Program
break
last_valid_address

```
mcb = x - sizeof(struct mem_control_block);  
mcb->is_available = 1;
```



Аллокация памяти

`free(x);`



Адрес начала выделенной памяти

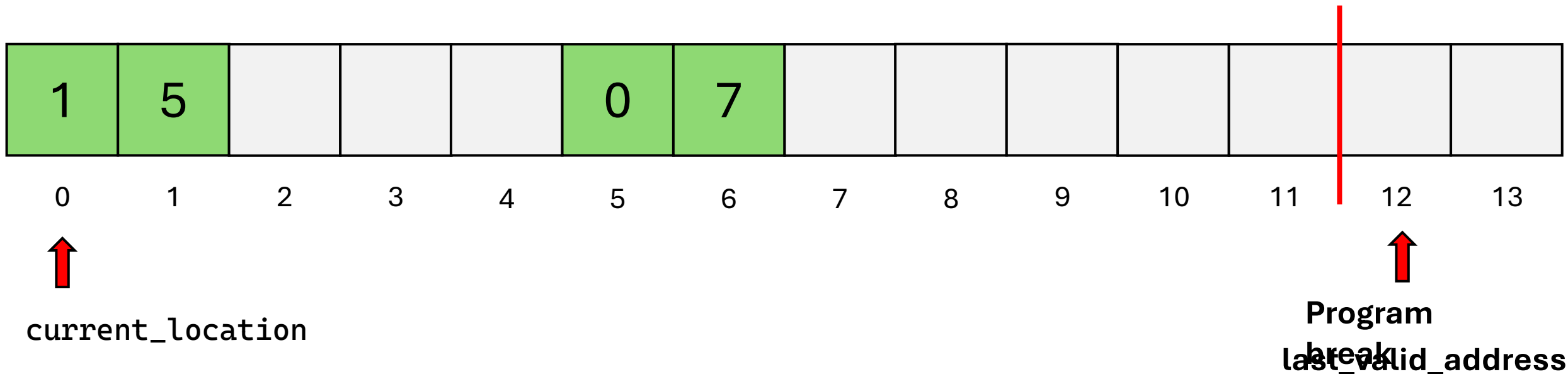
Program
break
last_valid_address

```
mcb = x - sizeof(struct mem_control_block);  
mcb->is_available = 1;
```



Аллокация памяти

```
int* y = (int*)malloc(2);
```



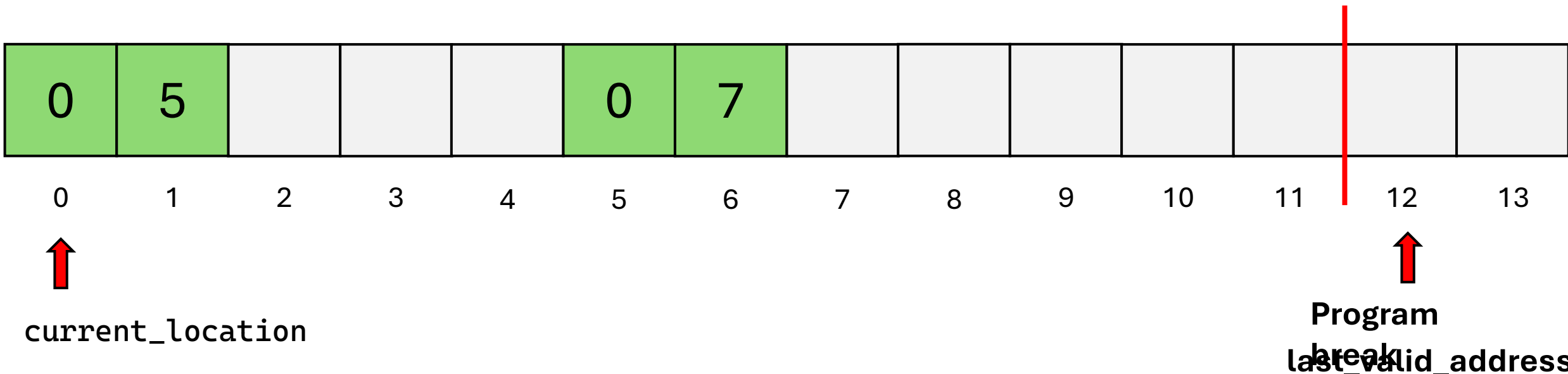
```
while (current_location != last_valid_address){  
    if (current_location_mcb->is_available) {//выделяем память}  
        current_location = current_location + current_location_mcb->size;  
    }  
}
```

Ячейка свободна, поэтому используем её



Аллокация памяти

```
int* y = (int*)malloc(2);
```

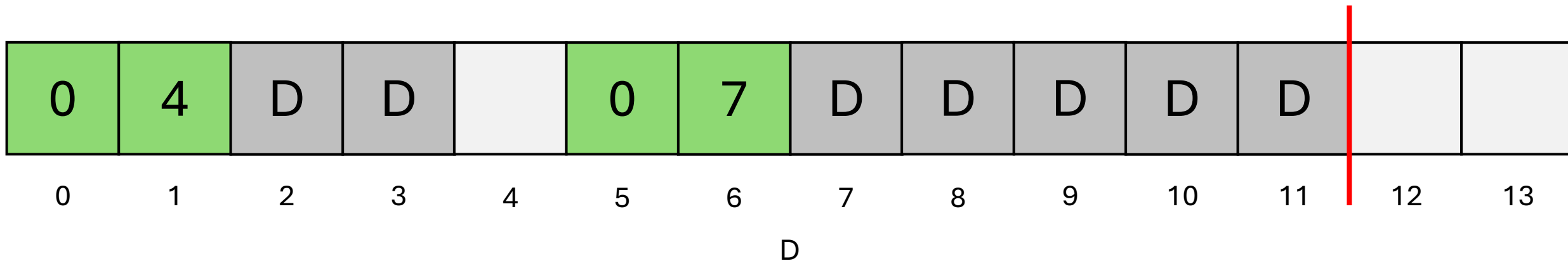


```
while (current_location != last_valid_address){  
    if (current_location_mcb->is_available) {//выделяем память}  
        current_location = current_location + current_location_mcb->size;  
    }  
}
```

Ячейка свободна, поэтому используем её



Недостатки реализации



1. Неэффективное использование памяти:
 - 1.1. Пустые ячейки (например, 4)
 - 1.2. Положение структур невозможно изменить
2. Медленная работа



Перечисления

```
enum Numbers  
{  
    Zero,  
    One,  
    Two,  
    Three,  
    Four  
};
```

Перечисление — это пользовательский тип, состоящий из набора целочисленных констант, называемых перечислителями.



Перечисления

```
enum Numbers  
{  
    Zero,  
    One,  
    Two,  
    Three,  
    Four  
};
```

Каждый параметр в *enumerator-list* присваивает имя значению набора перечисления. По умолчанию первый параметр *enumeration-constant* связан со значением 0. Следующий параметр *enumeration-constant* в списке связывается со значением (*constant-expression* + 1), если явно не указано другое значение. Имя параметра *enumeration-constant* эквивалентно его значению.



Переменное число аргументов (C++)

```
#include <iostream>
using namespace std;

template < typename T >
void summarray(T arg[], int length) {
    T S = 0;
    for (int n = 0; n < length; n++) {
        S+= arg[n];
    }
    cout << "S=" << S << endl;
}

template<class...A>
void func(A...args) {
    const int size = sizeof...(args);
    double res[sizeof...(args) + 1] = { args... };
    summarray(res, size);
}

int main(void)
{
    func();
    func(1);
    func(1, 2);
    func(1, 2, 3);
    func(1, 2, 3, 4, 5, 6);
    func(1.0, 0.2, 0.03, 0.004, 0.0005);
    return 0;
}
```



Переменное число аргументов (C)

```
#include <stdio.h>
#include <stdarg.h>
double sum_all(int num, ...)
{
    double sum = 0.0, s;
    va_list argptr;
    va_start(argptr, num);
    for (int i = 0; i < num; i++) {
        s = va_arg(argptr, double);
        sum += s;
    }
    va_end(argptr);
    return sum;
}

int main(void)
{
    double S;
    S = sum_all(5, 1.0, 0.2, 0.03, 0.004, 0.0005);
    printf("Sum = %f\n", S);
    return 0;
}
```

Sum = 1.234500



Основные определения из курса ОС

Процесс — программа во время исполнения и все её элементы: адресное пространство, глобальные переменные, регистры, стек, счетчик команд, состояние, открытые файлы, дочерние процессы и т. д

Поток - самостоятельная цепочка последовательно выполняемых операторов программы, соответствующих некоторой подзадаче

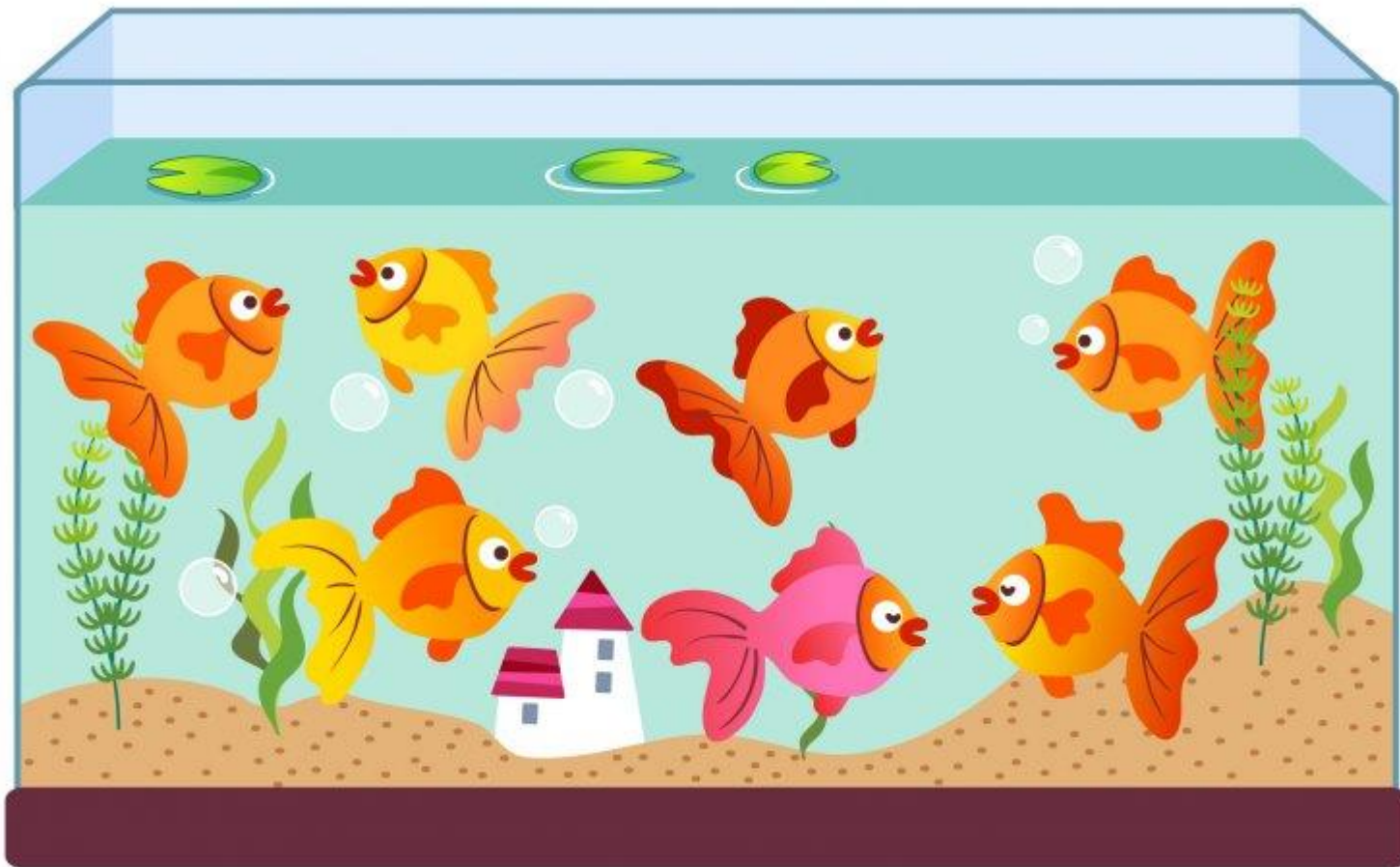
Прерывание — событие, при котором меняется последовательность команд, выполняемых процессором

Системный вызов — это интерфейс для получения услуг операционной системы

Файл — поименованная совокупность данных



Основные определения из курса ОС





Основные определения из курса ОС



Аквариум - процесс

Рыбки - потоки

Корм - ресурсы



```
#include <stdio.h>
```

```
int main()  
{  
    printf("Hello world!");  
    return 0;  
}
```



Стандартный ввод и вывод

[arch/x86/boot/printf.c](#)

```
int printf(const char* fmt, ...)
{
    char printf_buf[1024];
    va_list args;
    int printed;
    va_start(args, fmt);
    printed = vsprintf(printf_buf, fmt, args);
    va_end(args);
    puts(printf_buf);
    return printed;
}
```




Стандартный ввод и вывод

[arch/x86/boot/printf.c](#)

```
int vsprintf(char* buf, const char* fmt, va_list args)
{
    /////
    case '%':
        *str++ = '%';
        continue;
    /* integer number formats - set up the flags and "break" */
    case 'o':
    case 'x':
    case 'X':
    case 'd':
    case 'i':
    case 'u':
        break;
    /////
    return str - buf;
}
```



Стандартный ввод и вывод

[/arch/nios2/boot/compressed/console.c](#)

```
static int puts(const char* s)
{
    while (*s)
        putchar(*s++);
    return 0;
}
```



Стандартный ввод и вывод

[/arch/nios2/boot/compressed/console.c](#)

```
static int putchar(int ch)
{
    uart_putc(ch);
    if (ch == '\n')
        uart_putc('\r');
    return ch;
}
```



Стандартный ввод и вывод

[/arch/nios2/boot/compressed/console.c](#)

```
static void uart_putc(int ch)
{
    int i;

    for (i = 0; (i < 0x10000); i++) {
        if (readw(uartbase + ALTERA_UART_STATUS_REG) &
            ALTERA_UART_STATUS_TRDY_MSK)
            break;
    }
    writeb(ch, uartbase + ALTERA_UART_TXDATA_REG);
}
```



Стандартный ввод и вывод

[/arch/x86/boot/tty.c](#)

```
void __section(".inittext") puts(const char* str)
{
    while (*str)
        putchar(*str++);
}
```



Стандартный ввод и вывод

/arch/x86/boot/tty.c

```
void __section(".inittext") putchar(int ch)
{
    if (ch == '\n')
        putchar('\r'); /* \n -> \r\n */

    bios_putchar(ch);

    if (early_serial_base != 0)
        serial_putchar(ch);
}
```



Стандартный ввод и вывод

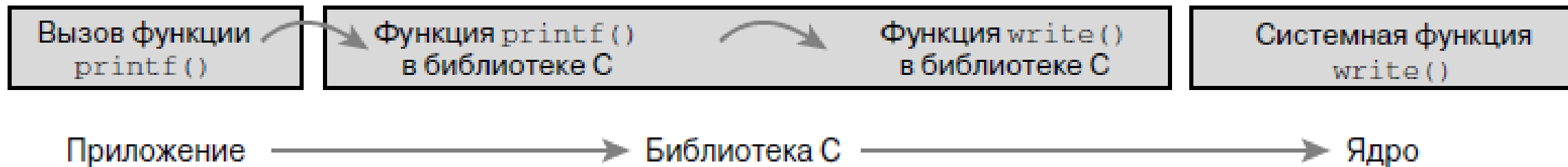
[/arch/x86/boot/tty.c](#)

```
static void __section(".inittext") bios_putchar(int ch)
{
    struct biosregs ireg;

    initregs(&ireg);
    ireg.bx = 0x0007;
    ireg.cx = 0x0001;
    ireg.ah = 0x0e;
    ireg.al = ch;
    intcall(0x10, &ireg, NULL);
}
```



Стандартный ввод и вывод



```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

```
write(fd1, buf, strlen(buf));
```




Hello world

```
#include <stdio.h>
```

```
int main()  
{  
    printf("Hello world!");  
    return 0;  
}
```

```
#include <unistd.h>
```

```
int main()  
{  
    char str[] = "Hello, world!\n";  
    write(1, str, sizeof(str) - 1);  
    _exit(0);  
}
```



Стандартный ввод и вывод

```
extern FILE *stdin; /* Standard input stream. */  
extern FILE *stdout; /* Standard output stream. */  
extern FILE *stderr; /* Standard error output stream. */
```



Основные функции работы с файлами

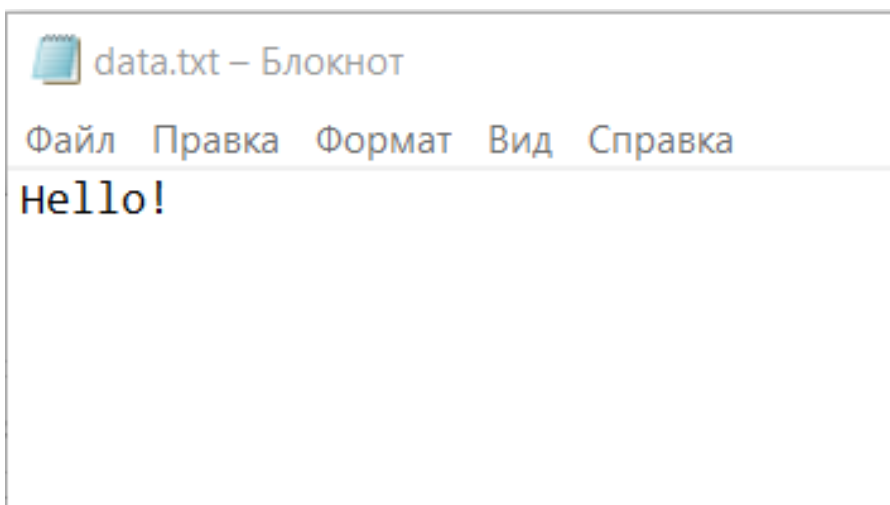
Описание	Синтаксис	Комментарий
Открытие файла	<code>errno_t fopen_s(FILE** pFile, const char* filename, const char* mode);</code>	Модификаторы: "r" Открывает для чтения "w" Открывает для перезаписи "a" Открывает для записи в конец файла
Запись в файл	<code>int fprintf(FILE *stream, const char *format [, argument]...);</code>	Если stream= stdout , то вывод будет производиться на экран
Чтение из файла	<code>char *fgets(char *str, int numChars, FILE *stream);</code>	Считывание возможно проводить в цикле до момента, когда функция вернет NULL <i>while ((fgets(c, 3, fp)) != NULL)</i> Считывает numChars – 1 символ
Закрытие файла	<code>int fclose(FILE *stream);</code>	



Работа с файлами

```
#include <stdio.h>

int main()
{
    FILE* fp;
    fp = fopen("data.txt", "w");
    fprintf(fp, "%s", "Hello!");
    fclose(fp);
    return 0;
}
```






Работа с файлами

```
#include <stdio.h>

int main()
{
    FILE* fp;
    char sym[10];
    fopen_s(&fp, "data.txt", "w");
    fprintf(fp, "%s", "Hello world!");
    fclose(fp);

    fopen_s(&fp, "data.txt", "r");
    while ((fgets(sym, 10, fp)) != NULL)
    {
        printf("%s\n", sym);
    }
    fclose(fp);

    return 0;
}
```

 data.txt – Блокнот

Файл Правка Формат Вид Справка

Hello world!

Hello wor
ld!



Работа с файлами

```
if ((fopen_s(&fp, filename, "w")) != NULL)
{
    perror("Error");
    return 1;
}
```

При работе с файлами
рекомендуется проводить
проверку на корректное
открытие

Полный код см. в заметках к слайду



Буфер stdout

```
#include <stdio.h>

int main()
{
    fprintf(stdout, "Hello 1 ");
    fprintf(stderr, "This is error. ");
    fprintf(stdout, " Hello 2 \n");
    return 0;
}
```

Что будет выведено в результате работы программы?



Буфер stdout

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    fprintf(stdout, "Hello 1 ");
```

```
    fprintf(stderr, "This is error. ");
```

```
    fprintf(stdout, " Hello 2 \n");
```

```
    return 0;
```

```
}
```

Что будет выведено в результате работы программы?

```
sab@LAPTOP-B03PIUAN:.../Lesson4$ ./ffl
```

```
This is error. Hello 1 Hello 2
```




Перенаправление потоков ввода/вывода

```
#include <stdio.h>
int main(){
    printf("world!");
    return 0;
}
```

write.c

```
#include <stdio.h>
int main(){
    char address[100];
    printf("Hello ");
    scanf("%s", address);
    printf("%s\n", address);
    return 0;
}
```

read.
c

```
./read > log.txt
./write > text.txt
./read < text.txt
./read > log.txt < text.txt
./write | ./read
```



Просто интересный пример

```
int x = 0;
int y = 0;

int r1 = 0;
int r2 = 0;

std::thread t1([&]() {
    x = 1;
    r1 = y;
});

std::thread t2([&]() {
    y = 1;
    r2 = x;
});
```

Чему могут быть равны r1 и r2 в результате работы программы?



Store buffering problem

```
int x = 0;
int y = 0;

int r1 = 0;
int r2 = 0;

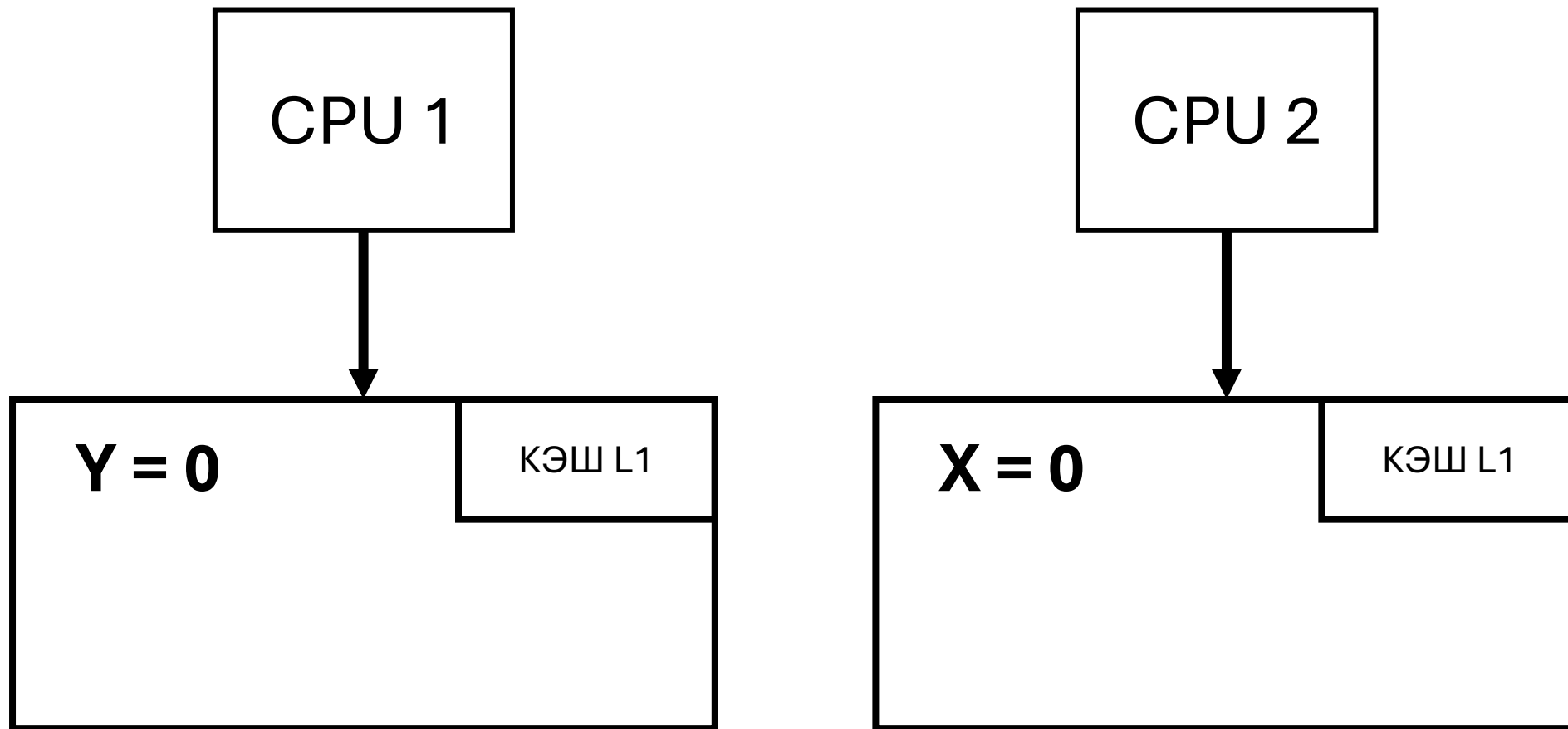
std::thread t1([&]() {
    x = 1;
    r1 = y;
});

std::thread t2([&]() {
    y = 1;
    r2 = x;
});
```

Чему могут быть равны r1 и r2 в результате работы программы?



Store buffering problem



Программа выполняется на двух ядрах. Может получиться такая ситуация, что в первом кэше есть значение **y**, не **x** нет. На втором ядре наоборот.



Store buffering problem

x = 1;
r1 = y;

CPU 1

x = 1

Y = 0

КЭШ L1

CPU 2

y = 1;
r2 = x;

y = 1

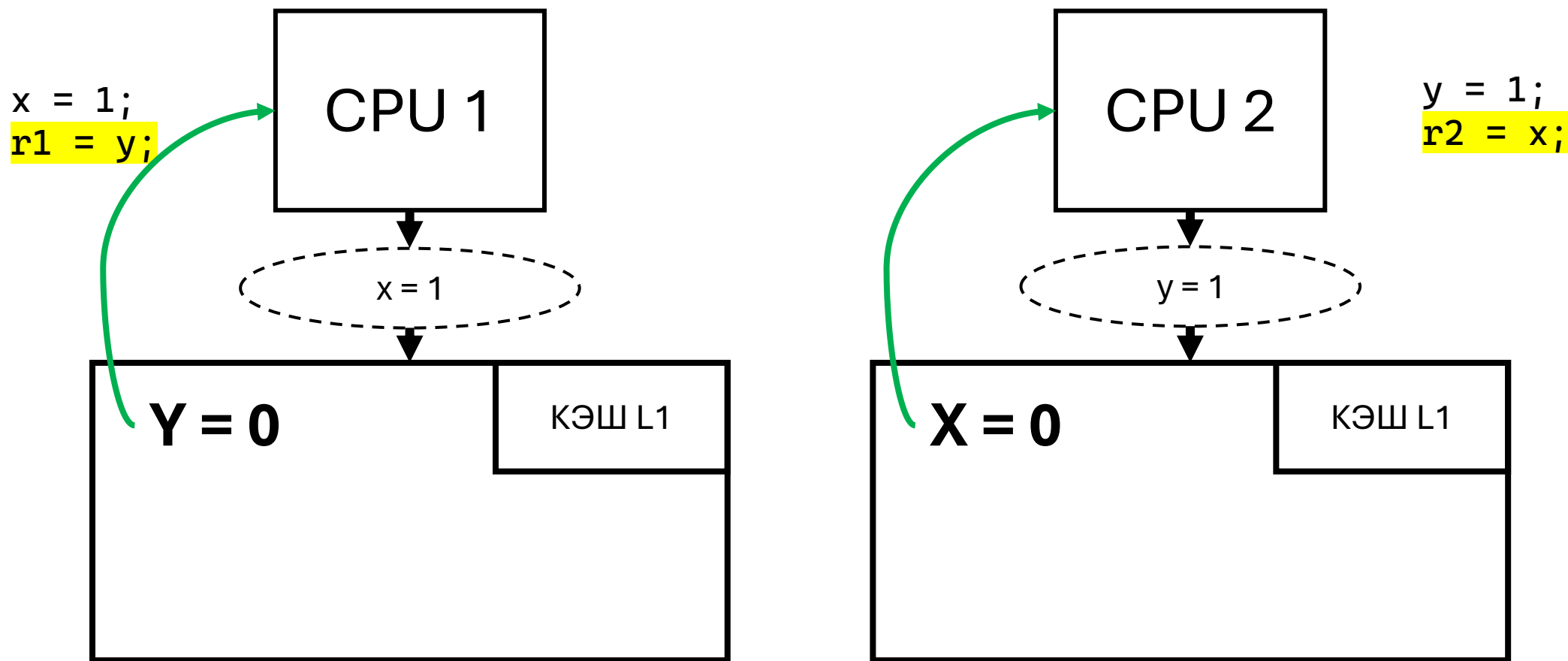
X = 0

КЭШ L1

Т.к. значения X на ядре нет, то оно не сразу пишется в кэш, т.к. это затормозит всю систему (придется менять флаги во всех ядрах), а положится в промежуточный буфер. Далее программа продолжит последовательно выполняться



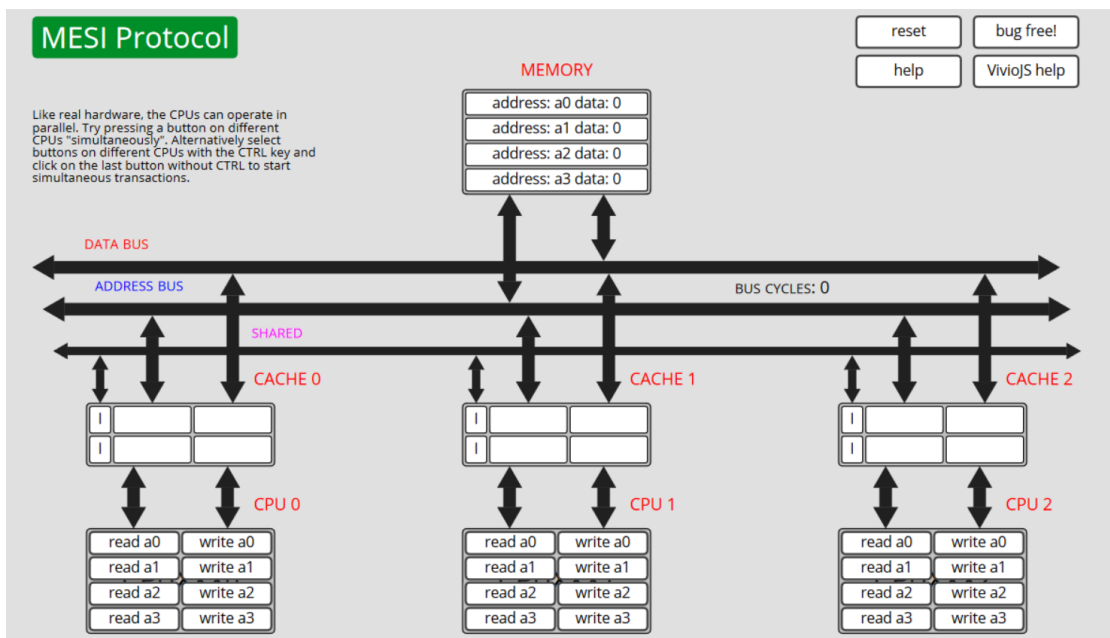
Store buffering problem



Программа возьмет значения из кэша, которые равны 0. Архитектура допускает такой вариант, т.к. это следствие (пусть и необычное) гонки данных, которая в программе не допускается



Интересные ссылки про кэш



<https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESI.htm>

https://www.youtube.com/watch?v=_p__rtZ-cjI

A Primer on Memory Consistency and Cache Coherence, Daniel J. Sorin, Mark D. Hill, David A. Wood