Содержание

Введение

Язык ассемблера 1

```
xchgq %rsi, %rdx
```

```
addq %RDX, %RSI
```

```
sub $32, %RSI
mov $5, %RAX
mul %RSI
mov $9, %RBX
div %RBX
mov %RAX, %RSI
```

```
pushq %RSI
movq %RDX, %RSI
popq %RDX
```

Язык ассемблера 2

```
swap:
    pushq (%RDI)
    pushq (%RSI)

    popq (%RDI)
    popq (%RSI)

    retq
```

```
min:
    movq %RDI, %RAX
    cmp %RDI, %RSI
    cmovl %RSI, %RAX
    ret
```

```
pow:
        movq $1, %rax
mul:
        sub $1, %rsi
        jl end
        mulq %rdi
        jmp mul
end:
        retq
```

RAX	Используется для возврата значения из функции.
RDI	Используется как первый аргумент функции.
RSI	Используется как второй аргумент функции.
RDX	Используется как третий аргумент функции.
RCX	Используется как четвертый аргумент функции.

Загрузка ОС

4096

Следующий шаг | Решить снова

Управление памятью

Физическая память

EBX

Логическая память

- [x] ОС может создать несколько таблиц дескрипторов.
- [x] Процесс может использовать несколько различных сегментов.
- [ ] Физический адрес при использовании сегментации всегда совпадает с логическим (эффективным адресом/смещением).
- [x] Участки физической памяти, описываемой дескрипторами сегментов, могут пересекаться.
- [ ] При использовании сегментации нельзя сделать так, чтобы два процесса имели доступ к одному участку физической памяти.

Следующий шаг | Решить снова

Страничная организация памяти

2

ВЗЯТО ИЗ КОММЕНТАРИЕВ К ЗАДАНИЮ!!! (НЕ Я ПИСАЛ)

Нам глобально чО нужна та, ***? - Написать код > взять датасет> подружить код и датасет > получить результат. Так начнем же с кода!

1. Это python3, но ты можешь его запускать как хочешь (хоть в редакторе, хоть в реакторе, в х**кторе):

```
#!/usr/bin/env python3
# coding=utf-8
```

```
import sys

def get_page(logical_addr: int):
      pml4 = (logical_addr >> 39) & 0x1ff
      dir_ptr = (logical_addr >> 30) & 0x1ff
      directory = (logical_addr >> 21) & 0x1ff
      table = (logical_addr >> 12) & 0x1ff
      offset = logical_addr & 0xfff
      return (pml4, dir_ptr, directory, table, offset)

def get_page_phy_addr(value: int):
      return (value & ((0xfffffffff) << 12))

def get_phy_addr(page: tuple, mem_struct: dict, cr3: int):
      value = cr3
      for i in range(len(page) - 1):
      index = page[i] * 8
      value = mem_struct.get(index + value, 0)
      if value & 1 == 0:
            print("fault")
            return
      value = get_page_phy_addr(value)
      print(value + page[-1])

def main():
      reader = (tuple(map(int, line.split())) for line in sys.stdin)
      mem_rows, queries, cr3 = next(reader)
      mem_struct = dict([next(reader) for _ in range(mem_rows)])
      for _ in range(queries):
      logical_addr, = next(reader)
      page = get_page(logical_addr)
      get_phy_addr(page, mem_struct, cr3)

if __name__ == "__main__":
      main()
```

====сохрани и назови его task.py====

2. Скачай датасет хоть по касательной, хоть по ***; мне все равно.

3. Врубай лентяя и делай так:

cat dataset_НОМЕР_ТВОГО_ДАТАСЕТА.txt | python3 task.py >> answer.txt

4. Ответ answer.txt отправляй на проверку.

(все файлы должны быть в одной директории\ или в одной и той же папке)

ГОТОВО!

Простой подход к аллокации памяти

```
struct head {
    head *prev;
    head *next;
    std::size_t size;
    std::size_t free; };

head *mymem;

void mysetup(void *buf, std::size_t size) {
    head *h = (head *)buf;
    mymem = h;
    h->prev = 0;
    h->next = 0;
    h->free = 1;
    h->size = size - sizeof(head); }

// Функция аллокации
void *myalloc(std::size_t size) {
    head *h = mymem;
    do {
        if (h->free && (h->size - sizeof(head) >= size)) {
            // Нашли достаточную область памяти
            if (h->size >= size + 3 * sizeof(head)) {
                // Достаточно места, чтобы разбить блок на две части
                head *n = (head *)(((void *)h) + h->size - size);
                n->next = h->next;
                n->prev = h;
                h->next = n;
                if (n->next) n->next->prev = n;
                n->size = size;
                h->size -= size + sizeof(head);
                n->free = 0;
                return (void *)(n +1);}
            else {
                // Выделяем блок полностью
                h->free = 0;
                return (void *)(h +1); } }
    } while (h = h->next);
```

```
    return NULL; }

// Функция освобождения
void myfree(void *p) {
    head *h = ((head *)p) - 1;
    h->free = 1;
    // Пробуем объединить с предыдущим блоком
    if (h->prev && h->prev->free) {
        head *n = h->prev;
        n->size += h->size + sizeof(head);
        n->next = h->next;
        if (n->next) {
            n->next->prev = n; }
        h = n; }
    if (h->next && h->next->free) {
    // Пробуем объединить со следующим блоком
        head *n = h->next;
        h->size += n->size + sizeof(head);
        h->next = n->next;
        if (n->next) {
            n->next->prev = h; } } }
```

Buddy аллокатор

- [x] Уровень парного блока может быть меньше i
- [ ] Уровень парного блока может быть больше i
- [x] Уровень парного блока может быть равен i

4

SLAB аллокатор

```
#include <inttypes.h>
#include <stdlib.h>
/**
 * Эти две функции вы должны использовать для аллокации
 * и освобождения памяти в этом задании. Считайте, что
 * внутри они используют buddy аллокатор с размером
 * страницы равным 4096 байтам.
 **/
```

```
/**
 * Аллоцирует участок размером 4096 * 2^order байт,
 * выровненный на границу 4096 * 2^order байт. order
 * должен быть в интервале [0; 10] (обе границы
 * включительно), т. е. вы не можете аллоцировать больше
 * 4Mb за раз.
 **/
void *alloc_slab(int order);
/**
 * Освобождает участок ранее аллоцированный с помощью
 * функции alloc_slab.
 **/
void free_slab(void *slab);

typedef struct memory_block
{
    struct memory_block *next;
} block_mem_t;

typedef struct slab_header
{
    block_mem_t *blocks;
    size_t free_num;
    struct slab_header *next;
    struct slab_header *prev;
} slab_head_t;

#define SLAB_OBJECTS_MIN_NUM 100

/**
 * Эта структура представляет аллокатор, вы можете менять
 * ее как вам удобно. Приведенные в ней поля и комментарии
 * просто дают общую идею того, что вам может понадобится
 * сохранить в этой структуре.
 **/
struct cache {
    slab_head_t *full;
    slab_head_t *partly_full;
    slab_head_t *free;

    size_t object_size; /* размер аллоцируемого объекта */
    int slab_order; /* используемый размер SLAB-а */
    size_t slab_objects; /* количество объектов в одном SLAB-е */
};

static void chache_init_slab(slab_head_t *slab, size_t object_size, size_t
objects_num)
{
    void *mem = (uint8_t*)slab + sizeof(slab_head_t);
    size_t offset = sizeof(block_mem_t) + object_size;
```

```
    for (int i = 0; i < objects_num; ++i)
    {
        ((block_mem_t*)mem)->next = (i + 1) == objects_num ? NULL :
(block_mem_t*)((uint8_t*)mem + offset);
        mem = (uint8_t*)mem + offset;
    }

    slab->blocks = (block_mem_t*)((uint8_t*)slab + sizeof(slab_head_t));
    slab->free_num = objects_num;
}

static slab_head_t *cache_create_slab(int order)
{
    slab_head_t *slab;

    slab = (slab_head_t*)alloc_slab(order);

    slab->blocks = NULL;
    slab->free_num = 0;
    slab->next = NULL;

    return slab;
}

/**
 * Функция инициализации будет вызвана перед тем, как
 * использовать это кеширующий аллокатор для аллокации.
 * Параметры:
 *  - cache - структура, которую вы должны инициализировать
 *  - object_size - размер объектов, которые должен
 *    аллоцировать этот кеширующий аллокатор
 **/
void cache_setup(struct cache *cache, size_t object_size)
{
    int order = 0;
    size_t meta_size = sizeof(slab_head_t);

    cache->full = NULL;
    cache->partly_full = NULL;
    cache->free = NULL;
    cache->object_size = object_size;

    size_t min_mem_required = sizeof(slab_head_t) + (sizeof(block_mem_t) +
object_size) * SLAB_OBJECTS_MIN_NUM;
    while (((1UL << order) * 4096) < min_mem_required)
    {
        ++order;
    }

    cache->slab_order = order;
```

```
    size_t addition_mem = ((1UL << order) * 4096 / min_mem_required - 1) *
min_mem_required +
                          ((1UL << order) * 4096 % min_mem_required);

    cache->slab_objects = SLAB_OBJECTS_MIN_NUM + addition_mem /
(sizeof(block_mem_t) + object_size);
}

/**
 * Функция освобождения будет вызвана когда работа с
 * аллокатором будет закончена. Она должна освободить
 * всю память занятую аллокатором. Проверяющая система
 * будет считать ошибкой, если не вся память будет
 * освбождена.
 **/
void cache_release(struct cache *cache)
{
    slab_head_t *tmp;

    tmp = cache->full;
    while (tmp)
    {
        cache->full = tmp->next;
        free_slab(tmp);
        tmp = cache->full;
    }

    tmp = cache->partly_full;
    while (tmp)
    {
        cache->partly_full = tmp->next;
        free_slab(tmp);
        tmp = cache->partly_full;
    }

    tmp = cache->free;
    while (tmp)
    {
        cache->free = tmp->next;
        free_slab(tmp);
        tmp = cache->free;
    }
}

static void cache_move_slab(struct cache *cache, slab_head_t **dest,
slab_head_t *src)
{
    if (cache->full == src)
        cache->full = src->next;
    else if (cache->partly_full == src)
```

```
        cache->partly_full = src->next;
    else if (cache->free == src)
        cache->free = src->next;

    if (src->prev)
    {
        src->prev->next = src->next;
    }
    if (src->next)
    {
        src->next->prev = src->prev;
    }

    src->prev = NULL;
    src->next = *dest;
    if (*dest)
        (*dest)->prev = src;
    *dest = src;
}

static void cache_free_block(slab_head_t *slab, block_mem_t *block)
{
    block->next = slab->blocks;
    slab->blocks = block;

    ++slab->free_num;
}

static block_mem_t *cache_alloc_block(slab_head_t *slab)
{
    block_mem_t *block;
    static size_t i = 0;

    block = slab->blocks;
    slab->blocks = block->next;
    --slab->free_num;

    block->next = NULL; // do we need it actually?

    return block;
}

/**
 * Функция аллокации памяти из кеширующего аллокатора.
 * Должна возвращать указатель на участок памяти размера
 * как минимум object_size байт (см cache_setup).
 * Гарантируется, что cache указывает на корректный
 * инициализированный аллокатор.
 **/
void *cache_alloc(struct cache *cache)
{
```

```
    block_mem_t *block;

    if (cache->partly_full)
    {
        block = cache_alloc_block(cache->partly_full);

        if (cache->partly_full->free_num == 0)
        {
            cache_move_slab(cache, &cache->full, cache->partly_full);
        }
    }
    else if (cache->free)
    {
        block = cache_alloc_block(cache->free);
        cache_move_slab(cache, &cache->partly_full, cache->free);
    }
    else
    {
        slab_head_t *slab = cache_create_slab(cache->slab_order);

        chache_init_slab(slab, cache->object_size, cache->slab_objects);

        block = cache_alloc_block(slab);
        cache->partly_full = slab;
    }

    return (uint8_t*)block + sizeof(block_mem_t);
}


/**
 * Функция освобождения памяти назад в кеширующий аллокатор.
 * Гарантируется, что ptr - указатель ранее возвращенный из
 * cache_alloc.
 **/
void cache_free(struct cache *cache, void *ptr)
{
    slab_head_t *slab = (slab_head_t*)((uint64_t)ptr & ~((1UL <<
cache->slab_order) * 4096 - 1));
    block_mem_t *block = (block_mem_t*)((uint8_t*)ptr - sizeof(block_mem_t));

    cache_free_block(slab, block);

    if (slab->free_num == 1)
    {
        cache_move_slab(cache, &cache->partly_full, slab);
    }
    else if (slab->free_num == cache->slab_objects)
    {
        cache_move_slab(cache, &cache->free, slab);
    }
```

```
}


/**
 * Функция должна освободить все SLAB, которые не содержат
 * занятых объектов. Если SLAB не использовался для аллокации
 * объектов (например, если вы выделяли с помощью alloc_slab
 * память для внутренних нужд вашего алгоритма), то освбождать
 * его не обязательно.
 **/
void cache_shrink(struct cache *cache)
{
    slab_head_t *tmp;

    tmp = cache->free;
    while (tmp)
    {
        cache->free = tmp->next;
        free_slab(tmp);
        tmp = cache->free;
    }
}
```

Планирование и многозадачность

Потоки исполнения и многопоточность

```
10
```

Переключение потоков

```
5
```

Планирование и критерии планирования

```
input()
print(' '.join(map(lambda t: str(t[0]), sorted(enumerate(map(int,
input().split())), key=lambda t: t[1]))))
```

Реалистичное планирование

```
#include <stdlib.h>

struct potock {
    int id;
    int time;
    struct potock *next;
};

static struct sched_rr {
    int slice; /* длительность кванта времени в тиках */
    struct potock *head; /* голова очереди на выполнение */
    struct potock *tail; /* хвост очереди на выполнение */
    struct potock *blkd; /* односвязный список заблокированных потоков */
} sched_rr;

/* переменная нужна лишь чтобы писать привычные стрелочки, а не точки */
static struct sched_rr *sched;

/* добавляет поток в хвост очереди, которая может быть пустой */
#define to_tail(p) do { \
    if(!sched->head) sched->head = p; \
    else sched->tail->next = p; \
    sched->tail = p; \
    p->time = sched->slice; \
    p->next = NULL; \
} while(0)

/* добавляет поток в голову списка заблокированных */
#define to_blkd(p) do { \
    p->next = sched->blkd; \
    sched->blkd = p; \
} while(0)

/* удаляет первый поток из очереди, предполагается, что очередь не пуста */
#define skip_head() do { sched->head = sched->head->next; } while(0)

void scheduler_setup(int timeslice)
{

    sched = &sched_rr;
    sched->head = sched->blkd = NULL;
    sched->slice = timeslice;
}

void new_thread(int thread_id)
{
    struct potock *ptr;
```

```
    ptr = (struct potock *) malloc(sizeof(struct potock));
    ptr->id = thread_id;
    to_tail(ptr);
}

void exit_thread()
{
    struct potock *ptr;

    if(ptr = sched->head) { /* возможно, if не нужен */
        skip_head();
        free(ptr);
    }
}

void block_thread()
{
    struct potock *ptr;

    if(ptr = sched->head) { /* возможно, if не нужен */
        skip_head();
        to_blkd(ptr);
    }
}

void wake_thread(int thread_id)
{
    struct potock *cur;

#if 0 /* два варианта поиска по id в списке */
      /* оба не нравятся, но хватит думать - прыгать надо */

    struct potock *ptr = sched->blkd;

    if(ptr->id == thread_id) {
        cur = ptr;
        sched->blkd = ptr->next;
    } else {
        while(ptr->next->id != thread_id) ptr = ptr->next;
        cur = ptr->next;
        ptr->next = ptr->next->next;
    }
#else
    struct potock **ptr = &sched->blkd;

    cur = sched->blkd;
    while(cur->id != thread_id) {
        ptr = &cur->next;
        cur = cur->next;
    }
    *ptr = cur->next;
```

```
#endif
    to_tail(cur);
}

void timer_tick()
{
    struct potock *ptr;

    if((ptr = sched->head) && !--ptr->time) {
        skip_head();
        to_tail(ptr);
    }
}

int current_thread()
{
    if(sched->head) return sched->head->id;
    else return -1;
}
```

- ○ Алгоритм будет таким же честным, как и оригинальный Round Robin.
- ◉ Алгоритм потеряет свойство честности, которое было у Round Robin.

Следующий шаг | Решить снова

Средства синхронизации потоков

Состояние гонки по данным и взаимное исключение

- ◉ Да
- ○ Нет

Взаимное исключение с использованием RW регистров, часть 1

- [x] 1, 4, 2, 3, 5, 6
- [x] 4, 1, 2, 3, 5, 6
- [ ] 1, 4, 2, 5, 3, 6
- [x] 4, 1, 5, 6, 2, 3
- [ ] 1, 2, 4, 3, 5, 6
- [ ] 1, 2, 3, 4, 5, 6

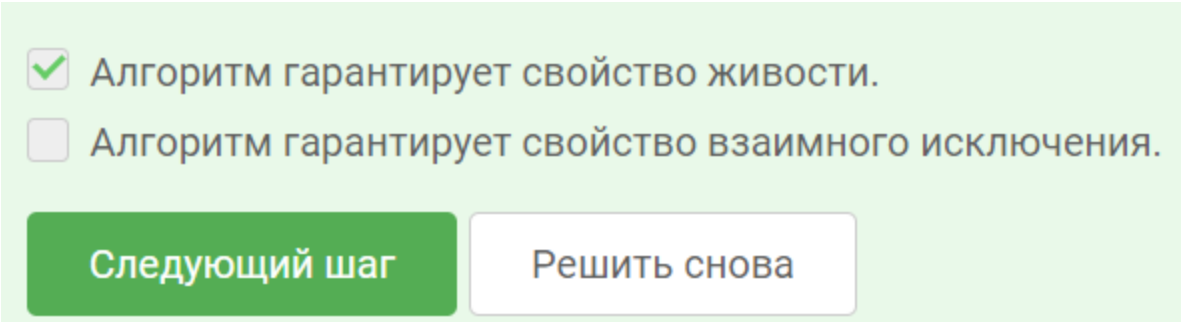
Следующий шаг | Решить снова

- [x] a == 1; b == 0; c == 1; d == 1;
- [ ] a == 0; b == 0; c == 1; d == 1;
- [x] a == 1; b == 1; c == 1; d == 1;
- [x] a == 0; b == 1; c == 1; d == 1;
- [ ] a == 0; b == 1; c == 1; d == 0;
- [ ] a == 1; b == 0; c == 0; d == 1;

Следующий шаг | Решить снова

- [x] Алгоритм гарантирует свойство взаимного исключения.
- [ ] Алгоритм гарантирует свойство живости.

Следующий шаг | Решить снова



Взаимное исключение с использованием RMW регистров

```
/* load_linked читает значение из ячейки памяти, на которую
   указывает x, и возвращает прочитанное значение. */
int load_linked(atomic_int *x);

/* store_conditional сохраняет значение new_value в ячейку
   памяти, на которую указывает x, но только при выполнении
   двух условий:
     - ячейка памяти ранее была прочитана с помощью load_linked
     - между последним load_linked для этой ячейки памяти и
       вызовом store_conditional никто не пытался записать
       значение в ячейку памяти
   Функция возвращает true, если значение было успешно записано
   и false в противном случае. */
bool store_conditional(atomic_int *x, int new_value);


/* Следующие две функции - ваше задание. Эти функции нужно
   реализовать используя load_linked и store_conditional для
   обращений к atomic_int */

/* atomic_fetch_add добавляет arg к значению записанному
   на которое указывает x. И возвращает предыдущее значение,
   на которое указывал x, как результат.

   Т. е. если x указывает на значение 3138 и вы вызываете
   atomic_fecth_add с arg == 10, то функция должна изменить
   значение, на которое указывает x на 3148 и вернуть 3138
   в качестве результата. */
int atomic_fetch_add(atomic_int *x, int arg)
{
    int oldval;
     do
     {
          oldval = load_linked(x);
     } while (!store_conditional(x, oldval + arg));

     return oldval;
```

```
}


/* atomic_compare_exchange сравнивает значение, на которое
   указывает x, со значением, на которое указывает expected_value,
   если они равны, то функция должна записать new_value в ячейку,
   на которую указывает x и возвращает true.
   В противном случае, функция должна записать значение, на
   которое указывает x, в ячейку, на которую указывает
   expected_value и вернуть false.

   Т. е. если x указывает на значение 42, expected_value тоже
   указывает на 42, и функция вызывается с new_value == 3148,
   то функция должна записать 3148 в ячейку, на которую указывает
   x, и вернуть true.

   А если x указывает на значение 3148, expectet_value указывает
   на значение 42, и new_value == 0, то функция должна записать
   в *expected_value значение 3148 и вернуть false. */
bool atomic_compare_exchange(atomic_int *x, int *expected_value,
                             int new_value)
{
    do
      {
            int oldval = load_linked(x);
            if (oldval != *expected_value)
            {
                  *expected_value = oldval;
                  return false;
            }
      } while (!store_conditional(x, new_value));

      return true;
}
```

Deadlock-и и средства борьбы с ними

- ○ Да
- ◉ Нет

- ◉ Да
- ○ Нет

```
/* Напоминание, как выглядят интерфейсы блокировки и переменной
   состояния.

   ВАЖНО: обратите внимание на функции lock_init и condition_init,
          я не уделял этому внимание в видео, но блокировки и
          переменные состояния нужно инциализировать.
*/

struct lock;
void lock_init(struct lock *lock);
void lock(struct lock *lock);
void unlock(struct lock *lock);

struct condition;
void condition_init(struct condition *cv);
void wait(struct condition *cv, struct lock *lock);
void notify_one(struct condition *cv);
void notify_all(struct condition *cv);



/* Далее следует интерфейс, который вам нужно реализовать.

   ВАЖНО: в шаблоне кода стукрутуры содержат поля, некоторые
          функции уже реализованы и присутсвуют комментарии
          - это не более чем подсказка. Вы можете игнорировать
          комментарии, изменять поля структур и реализации
          функций на ваше усмотрение, при условии, что вы
          сохранили интерфейс.

          Вам нельзя изменять имена структур (wdlock_ctx и
          wdlock), а также имена функций (wdlock_ctx_init,
          wdlock_init, wdlock_lock, wdlock_unlock).
*/

struct wdlock_ctx;

struct wdlock {
    /* wdlock_ctx должен хранить информацию обо всех
       захваченных wdlock-ах, а это поле позволит связать
       wdlock-и в список. */
     struct wdlock *next;

    /* Текущий владелец блокировки - из него мы извлекаем
       timestamp связанный с блокировкой, если блокировка
       свободна, то хранит NULL. */
     const struct wdlock_ctx *owner;

    /* lock и cv могут быть использованы чтобы дождаться
       пока блокировка не освободится либо у нее не сменится
       владелец. */
```

```
      struct lock lock;
      struct condition cv;
};


/* Каждый контекст имеет свой уникальный timestamp и хранит
   список захваченных блокировок. */
struct wdlock_ctx {
      unsigned long long timestamp;
      struct wdlock *locks;
};


/* Всегда вызывается перед тем, как использовать контекст.

   ВАЖНО: функция является частью интерфейса - не меняйте
          ее имя и аргументы.
*/
void wdlock_ctx_init(struct wdlock_ctx *ctx)
{
      static atomic_ullong next;

      ctx->timestamp = atomic_fetch_add(&next, 1) + 1;
      ctx->locks = NULL;
}

/* Всегда вызывается перед тем, как использовать блокировку.

   ВАЖНО: функция является частью интерфейса - не меняйте
          ее имя и аргументы.
*/
void wdlock_init(struct wdlock *lock)
{
      lock_init(&lock->lock);
      condition_init(&lock->cv);
      lock->owner = NULL;
}

/* Функция для захвата блокировки l контекстом ctx. Если
   захват блокировки прошел успешно функция должна вернуть
   ненулевое значение. Если же захват блокировки провалился
   из-за проверки timestamp-а, то функция должна вернуть 0.

   Помните, что контекст должен хранить информацию о
   захваченных блокировках, чтобы иметь возможность освободить
   их в функции wdlock_unlock.

   ВАЖНО: функция является частью интерфейса - не меняйте
          ее имя и аргументы.
*/
int wdlock_lock(struct wdlock *l, struct wdlock_ctx *ctx)
```

```
{
    return 1;
}

/* Функция для освбождения всех блокировок, захваченных
   контекстом ctx. При этом может случиться так, что этот
   контекст не владеет ни одной блокировкой, в этом случае
   ничего делать не нужно.

   ВАЖНО: функция является частью интерфейса - не меняйте
          ее имя и аргументы.
*/

void wdlock_unlock(struct wdlock_ctx *ctx)
{

}
```

Пространство пользователя

Исполняемые файлы и процессы

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

#define ELF_NIDENT   16

// Эта структура описывает формат заголовока ELF файла
struct elf_hdr {
     std::uint8_t e_ident[ELF_NIDENT];
     std::uint16_t e_type;
     std::uint16_t e_machine;
     std::uint32_t e_version;
     std::uint64_t e_entry;
     std::uint64_t e_phoff;
     std::uint64_t e_shoff;
     std::uint32_t e_flags;
     std::uint16_t e_ehsize;
     std::uint16_t e_phentsize;
     std::uint16_t e_phnum;
     std::uint16_t e_shentsize;
     std::uint16_t e_shnum;
     std::uint16_t e_shstrndx;
} __attribute__((packed));
```

```
std::uintptr_t entry_point(const char *name)
{
    FILE *elf_file;
    elf_file = fopen (name,"rb");

    struct elf_hdr elf_header;
    fread(&elf_header, 1, sizeof(elf_hdr), elf_file);

    fclose(elf_file);

    return elf_header.e_entry;
}
```

```
#define ELF_NIDENT   16
#define PT_LOAD      1

struct elf_hdr {
     std::uint8_t e_ident[ELF_NIDENT];
     std::uint16_t e_type;
     std::uint16_t e_machine;
     std::uint32_t e_version;
     std::uint64_t e_entry;
     std::uint64_t e_phoff; // program header offset
     std::uint64_t e_shoff;
     std::uint32_t e_flags;
     std::uint16_t e_ehsize;
     std::uint16_t e_phentsize;
     std::uint16_t e_phnum; // program header number
     std::uint16_t e_shentsize;
     std::uint16_t e_shnum;
     std::uint16_t e_shstrndx;
} __attribute__((packed));

struct elf_phdr {
     std::uint32_t p_type; // type
     std::uint32_t p_flags;
     std::uint64_t p_offset;
     std::uint64_t p_vaddr;
     std::uint64_t p_paddr;
     std::uint64_t p_filesz;
     std::uint64_t p_memsz; // memory size
     std::uint64_t p_align;
} __attribute__((packed));



std::size_t space(const char *name) {
```

```
    size_t result = 0;

    FILE* f= fopen(name, "rb");
    if (!f) {
        perror("unable to open");
    }

    elf_hdr elf;
    fread(&elf, sizeof(elf_hdr), 1, f);
    for (int i=0; i< elf.e_phnum; i++) {
        if (fseek (f, elf.e_phoff + i * sizeof(elf_phdr), SEEK_SET)) {
            perror("unable to seek");
        }

        elf_phdr h;
        fread(&h, sizeof(elf_phdr), 1,f);
        if (h.p_type == PT_LOAD) {
            result += h.p_memsz;
        }
    }

    fclose(f);
    return result;
}
```