

# **Операционные системы**

## **Сборник практических заданий**

**Москва, 2024**

## Оглавление

|  |    |
|--|----|
| Практическое задание 1. Загрузка операционной системы .....  | 3  |
| Практическое задание 2. Процессы в операционной системе..... | 10 |
| Практическая работа 3. Поток в операционной системе .....    | 18 |
| Практическая работа 4. Тупики и синхронизация потоков .....  | 27 |
| Практическая работа 5. Файловые системы и RAID массивы ..... | 35 |

# Практическое задание 1. Загрузка операционной системы

## Теоретическая часть

Загрузка операционной системы проходит в несколько этапов.

Первое, что запускает процессор при включении компьютера — это код BIOS (или же UEFI, но здесь я буду говорить только про BIOS), который "зашит" в памяти материнской платы (конкретно — по адресу 0xFFFFF0).

Сразу после включения BIOS запускает Power-On Self-Test (POST) — самотестирование после включения. BIOS проверяет работоспособность памяти, обнаруживает и инициализирует подключенные устройства, проверяет регистры, определяет размер памяти и так далее и так далее.

Следующий шаг — определение загрузочного диска, с которого можно загрузить ОС. Загрузочный диск — это диск (или любой другой накопитель), у которого последние 2 байта первого сектора (под первым сектором подразумевается первые 512 байт накопителя, т.к. 1 сектор = 512 байт) равны 55 и AA (в шестнадцатеричном формате). Как только загрузочный диск будет найден, BIOS загрузит первые его 512 байт в оперативную память по адресу 0x7c00 и передаст управление процессору по этому адресу.

Само собой, в эти 512 байт не выйдет уместить полноценную операционную систему. Поэтому обычно в этот сектор кладут первичный загрузчик, который загружает основной код ОС в оперативную память и передает ему управление.

С самого начала процессор работает в Real Mode (= 16-битный режим). Это означает, что он может работать лишь с 16-битными данными и использует сегментную адресацию памяти, а также может адресовать только 1 Мб памяти. Но вторым мы пользоваться здесь не будем. Картинка ниже показывает состояние оперативной памяти при передаче управления нашему коду.

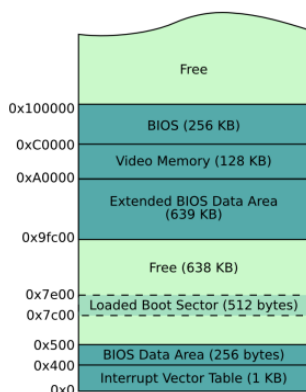


Figure 3.4: Typical lower memory layout after boot.

Последнее, о чем стоит сказать перед практической частью — прерывания. Прерывание — это особый сигнал (например, от устройства ввода, такого, как клавиатура или мышь) процессору, который говорит, что нужно немедленно прервать исполнение текущего кода и выполнить код обработчика прерывания. Все адреса обработчиков прерывания находятся в Interrupt Descriptor Table (IDT) в оперативной памяти. Каждому прерыванию соответствует свой обработчик прерывания. Например, при нажатии клавиши клавиатуры вызывается прерывание, процессор останавливается, запоминает адрес прерванной

инструкции, сохраняет все значения своих регистров (на стеке) и переходит к выполнению обработчика прерывания. Как только его выполнение заканчивается, процессор восстанавливает значения регистров и переходит обратно, к прерванной инструкции и продолжает выполнение.

Например, чтобы вывести что-то на экран в BIOS используется прерывание 0x10 (шестнадцатеричный формат), а для ожидания нажатия клавиши — прерывание 0x16. По сути, это все прерывания, что нам понадобятся здесь.

Также, у каждого прерывания есть своя подфункция, определяющая особенность его поведения. Чтобы вывести что-то на экран в текстовом формате (!), нужно в регистр AH занести значение 0x0e. Помимо этого, у прерываний есть свои параметры. 0x10 принимает значения из ah (определяет конкретную подфункцию) и al (символ, который нужно вывести). Таким образом,

```
mov ah, 0x0e
mov al, 'x'
int 0x10
```

выведет на экран символ 'x'. 0x16 принимает значение из ah (конкретная подфункция) и загружает в регистр al значение введенной клавиши. Мы будем использовать функцию 0x0.

## Основы ассемблера

Микропроцессор содержит набор регистров:

### *Пользовательские регистры*

- Регистры общего назначения
- Сегментные регистры
- Регистр флагов
- Регистр управления

Программная модель микропроцессора содержит 32 регистра, которые можно разделить на две группы:

- 16 пользовательских регистров;
- 16 системных регистров.

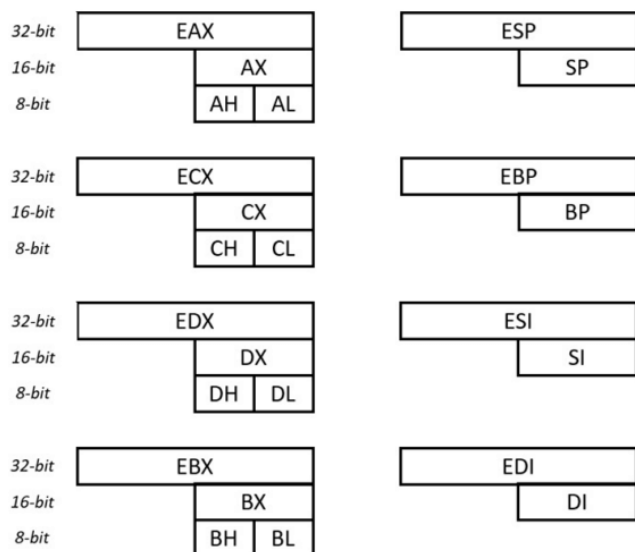
Пользовательскими регистры называются потому, что программист может их использовать при разработке программ.

К пользовательским регистрам относятся:

1. восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов (регистры общего назначения)
  - a. eax/ax/ah/al
  - b. ebx/bx/bh/bl
  - c. edx/dx/dh/dl
  - d. ecx/cx/ch/cl
  - e. esi/si
  - f. edi/di
  - g. ebp/bp
  - h. esp/sp
2. шесть регистров сегментов: cs, ds, ss, es, fs, gs

3. регистр состояния (регистр флагов) `eflags/flags`;
4. регистр управления (регистр указателя команды) `eip/ip`.

Обратим внимание, что к возможно обращаться к частям регистра.



Основные команды ассемблера:

| Команда   | Описание  |
|---|---|
| <code>MOV &lt;операнд<sub>1</sub>&gt;, &lt;операнд<sub>2</sub>&gt;</code> | По команде MOV значение второго операнда записывается в первый операнд.                                     |
| <code>ADD &lt;операнд<sub>1</sub>&gt;, &lt;операнд<sub>2</sub>&gt;</code> | Команда ADD складывает операнды и записывает их сумму на место первого операнда.                            |
| <code>SUB &lt;операнд<sub>1</sub>&gt;, &lt;операнд<sub>2</sub>&gt;</code> | Команда SUB вычитает из первого операнда второй и записывает полученную разность на место первого операнда. |
| <code>INC &lt;операнд&gt;</code>  | Увеличить операнд на 1  |
| <code>DEC &lt;операнд&gt;</code>  | Уменьшить операнд на 1  |
| <code>JMP &lt;метка&gt;</code>  | Команда безусловного перехода   |
| <code>CMP &lt;операнд<sub>1</sub>&gt;, &lt;операнд<sub>2</sub>&gt;</code> | Команда сравнения операндов   |
| <code>JE &lt;метка&gt;</code>   | Переход если равно  |
| <code>JNE &lt;метка&gt;</code>  | Переход если не равно   |

## Видеоадаптер

При выводе текста различные видеосистемы работают одинаково. Для экрана (25 строк по 80 символов) отводится 4000 байт (по 2 байта на каждый символ). Первый (четный) байт содержит код ASCII, который аппаратно преобразуется в связанный с ним символ и посылается на экран. Второй (нечетный) байт содержит атрибуты. Это информация о том, как должен быть выведен символ.

### Структура байта атрибутов:

|            |                           |
|------------|---------------------------|
| биты 0-1-2 | код цвета символа         |
| бит 3      | бит интенсивности         |
| биты 4-5-6 | код цвета фона знакоместа |
| бит 7      | - 0 (мигание - 1)         |

### Коды цветов:

|   |   |
|---|---|
| 0 | черный                                      |
| 1 | синий                                       |
| 2 | зеленый                                     |
| 3 | циан (голубой)                              |
| 4 | красный                                     |
| 5 | магента                                     |
| 6 | коричневый (с битом интенсивности - желтый) |
| 7 | белый                                       |

## Практическая часть

### 1. Изучите программный код, приведенный ниже.

```

;
; Простой загрузочный сектор, который выводит сообщение на экран с помощью процедуры BIOS.
;
mov ah, 0x0e ; При значениях ah = 0x0E, int 0x10 печать происходит на текущей активной странице.
mov al, 'H'
int 0x10
mov al, 'e'
int 0x10
mov al, 'l'
int 0x10
mov al, 'l'
int 0x10
mov al, 'o'
int 0x10
jmp $ ; Бесконечный цикл
;
; Заполнение и магический номер BIOS.
;
times 510 -( $ - $$ ) db 0 ; Заполнить загрузочный сектор нулями
dw 0xaa55 ; последние два байта - магический номер, чтобы биос определил что это загрузочный сектор

```

Скомпилируйте код с помощью *nasm* и запустите его в эмуляторе *qemu*


```
nasm -f bin hello.asm -o os.img
```

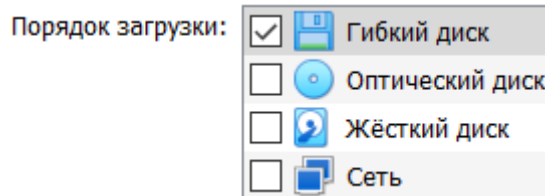
Убедитесь, что на экране появилось сообщение «Hello»

*Измените программу таким образом, чтобы на экран выводились ваши ФИО*

Запустите собранный образ диска на эмуляторе VirtualBox

Для этого выполните следующие действия:

1. Откройте программу VirtualBox
2. Создайте новую виртуальную машину (Кнопка «Создать»)
  - a. Задайте имя машины – ваши инициалы.
  - b. Тип: Other
  - c. Версия: Other/Unknown
  - d. Основная память: 4 Mb
  - e. Процессоры: 1
  - f. Виртуальный жесткий диск: не подключать
3. Зайдите в настройки созданной машины и перейдите в раздел «Носители»
  - a. Удалите все существующие носители и добавьте контроллер Floppy ()
  - b. Выберите гибкий диск – созданный образ из предыдущего пункта
4. В разделе система установите Гибкий диск первым в порядке загрузки. Остальные галочки уберите.



5. Выйдите из настроек и запустите вашу операционную систему.  
Если все сделано верно, то на экране появится сообщение.

## 2. Изучите программный код, приведенный ниже.

```
[BITS 16]
[ORG 0x7c00]
_start:
    cli
    mov ax, cs
    mov ds, ax
    mov ss, ax
    mov sp, _start

;; Загрузка регистра GDTR:
lgdt [gd_reg]

;; Включение A20:
in al, 0x92
or al, 2
out 0x92, al

;; Установка бита PE регистра CR0
mov eax, cr0
or al, 1
mov cr0, eax

;; С помощью длинного прыжка мы загружаем
;; селектор нужного сегмента в регистр CS
;; (напрямую это сделать нельзя)
;; 8 (1000b) - первый дескриптор в GDT, RPL=0
jmp 0x8: _protected

[BITS 32]
_protected:
;; Загрузим регистры DS и SS селектором
;; сегмента данных
mov ax, 0x10
```

```

mov ds, ax
mov ss, ax

mov ebx, Message
call print_string_pm
mov eax, $1
mov ebx, $1
add eax, ebx

;; Завесим процессор
hlt
jmp short $

Message db "Hello World", 0x0

%define VIDEO_MEMORY 0xb8000
%define WHITE_ON_BLACK 0x0f
; устанавливаем атрибуты символа.

; печатает строку, оканчивающуюся символом 0x0
print_string_pm :
    pusha
    mov edx, VIDEO_MEMORY ; Устанавливает в edx значение начала памяти видеобуфера

print_string_pm_loop :
    mov al, [ ebx ] ; Сохраняет значение символа из EBX в AL
    mov ah, WHITE_ON_BLACK ; Устанавливает атрибуты в AH
    cmp al, 0 ; если ( al == 0 ) , то жто конец строки ->
    je print_string_pm_done ; прыжок на окончание
    mov [ edx ], ax ; Переместите символ и атрибуты в edx
    add ebx, 1 ; Увеличте ebx, чтобы обратиться к следующему символу
    add edx, 2 ; Перейдите к следующей ячейке в видеопамати
    jmp print_string_pm_loop ; продолжаем цикл, пока не выведены все символы

print_string_pm_done :
    popa
    ret ; выход из функции

gdt:
    dw 0, 0, 0, 0 ; Нулевой дескриптор

    db 0xFF ; Сегмент кода с DPL=0
    db 0xFF ; Базой=0 и Лимитом=4 Гб
    db 0x00
    db 0x00
    db 0x00
    db 10011010b
    db 0xCF
    db 0x00

    db 0xFF ; Сегмент данных с DPL=0
    db 0xFF ; Базой=0 и Лимитом=4Гб
    db 0x00
    db 0x00
    db 0x00
    db 10010010b
    db 0xCF
    db 0x00

;; Значение, которое мы загрузим в GDTR:
gd_reg:
    dw 8192
    dd gdt

times 510-($-$$) db 0
db 0xaa, 0x55

```

Скомпилируйте и запустите программу и убедитесь, что на экране появилось сообщение.

*Измените программу таким образом, чтобы на экран выводились ваши ФИО и номер группы. Установите цвет символа, чей номер равен вашему номеру в списке mod 8*



### **Контрольные вопросы**

1. Опишите процесс загрузки ОС
2. В чем отличие защищенного режима от реального?
3. Как работает видеоадаптер?

### **Список рекомендованной литературы**

1. Карпов В.Е., Коньков К.А. Основы операционных систем. Москва: Физматкнига, 2019. 326 pp.
2. Таненбаум Э., Бос Х. Современные операционные системы. Санкт-Петербург: Питер, 2021. 1119 pp.
3. <https://natalia.appmat.ru/c&c++/assembler.html>

## Практическое задание 2. Процессы в операционной системе

### Теоретическая часть

#### Процессы в операционной системе

Одним из основных понятий в операционных системах является понятие **процесса** – *программа во время исполнения или объект, которому выделяются ресурсы вычислительной системы, такие как процессорное время, память и т.д.*

Каждому процессу выделено адресное пространство, в котором содержится код программы (*text section*), данные для ее работы (*data section*), связанные файлы, с которыми идет работа, ожидающие обработки сигналы и т.п. Помимо этого, каждый процесс имеет собственный уникальный номер – идентификатор (PID – *process identifier*). Этот номер присваивается операционной системой при старте процесса, при этом значение PID нового процесса будет больше, чем у ранее созданного процесса, до тех пор, пока не будет присвоен максимально возможный номер для типа данных PID, который хранится в файле */proc/sys/kernel/pid\_max*. После этого ОС назначает новым процессам номера от 1, если данные значения были освобождены другими процессами. Самому первому процессу, отвечающему за планирование остальных процессов, присваивается 0 и данный PID не может быть передан другому процессу.

Любой процесс порождается другим процессом, таким образом, в ОС всегда функционирует дерево процессов, каждый из которых является либо родительским процессом, т.е. процессом, порождающим другие, либо дочерним (порождаемым), либо и тем и другим. Чтобы можно было отследить преемственность процессов, помимо уникальных идентификаторов PID у каждого порожденного процесса есть данные о родителе – идентификатор родительского процесса (PPID – *parent process identifier*). В случае, если родительский процесс завершается раньше дочернего, то значение PPID у последнего приравнивается к 1 – это процесс *init*, который функционирует все время работы операционной системы.

#### Работа с процессами из консоли

Для администрирования текущих процессов операционной системы из консоли можно воспользоваться утилитой *ps*, которая позволяет просматривать такую информацию о работе запущенных в системе процессах как: PID, PPID, занимаемое процессорное время, занимаемую память и т.д. Данную информацию утилита берет из каталога */proc*, где в виде файлов хранится состояние ядра операционной системы и запущенных процессов.

Пример запуска:

```
$ ps -u
```

Запустим утилиту с наиболее интересными нам параметрами. Для этого выполним команду:

```
$ ps -ao user,pid,ppid,vsz,stat,cmd
```

Результат работы приведен на рисунке 1.

```
os@os-VirtualBox:~/Desktop$ ps -ao user,pid,ppid,rss,stat,cmd
USER      PID     PPID    RSS  STAT  CMD
os         997      992  37712  Sl+   /usr/lib/xorg/Xorg vt2 -displayfd
os        1079      992   2408  Sl+   /usr/libexec/gnome-session-binary
os        2980     1937   3308  R+    ps -ao user,pid,ppid,rss,stat,cmd
```

Рисунок 1 Результат работы ps

Поясним результат работы утилиты. В столбце USER указывается имя пользователя, запустившего данный процесс. В данном случае это пользователь os. PID – идентификатор запущенного процесса. PPID – идентификатор процесса-родителя. RSS — это размер резидентной памяти, указывающий сколько памяти выделено для этого процесса и находится в ОЗУ в момент вызова команды. Поле STAT обозначает текущее состояние процесса. Наиболее часто встречаются следующие значения:

- R - процесс выполняется (или находится в очереди исполнения)
- S - процесс ожидает завершения события
- Z - процесс «зомби»

Также к данным значениям могут добавляться специальные символы. Например, «l» обозначает, что данное приложение является многопоточным, а «+» - процесс относится к группе переднего плана. Поле COMMAND содержит название запущенной команды.

Любой процесс может быть завершен с помощью утилиты kill, если известен его идентификатор – PID и пользователь обладает на это достаточными правами.

Аналогичную информацию возможно увидеть с помощью утилиты top.

```
$ top
```

## Работа с процессами на языке программирования C

Порождение дочернего процесса родительским возможно осуществить с помощью системного вызова *fork()* и семейства функций *exec()*.

Процесс, созданный с помощью системного вызова *fork()* (от англ. «вилка») является точной копией родительского процесса, за исключением таких параметров как PID и PPID.

Чтобы при выполнении кода различать родительский и дочерний процессы, функция *fork()* возвращает разные для них значения. Так, при успешном создании нового процесса в родительский процесс возвращается PID дочернего, а в дочерний процесс возвращается – 0. За счет проверки возвращаемого значения можно по-разному организовать дальнейшую работу родственных процессов.

```

pid = fork();
if(pid == -1){
    ...
    /* ошибка */
    ...
} else if (pid == 0){
    ...
    /* дочерний процесс */
    ...
} else {
    ...
    /* родительский процесс */
    ...
}

```

После окончания работы дочерний процесс должен отправить сигнал о своем завершении родительскому процессу. Для ожидания данного сигнала существует системный вызов *wait()*.

Рассмотрим следующий пример:

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main()
{
    pid_t PID=fork();
    if(PID==0)
    {
        printf("Hello ");
    }
    else
    {
        wait(0);
        printf("world!\n");
    }
    return 0;
}

```

В приведенном примере показана работа системного вызова *fork()*. После его вызова создается копия текущего процесса. Далее код начинает выполняться со следующей строки уже в **двух** процессах. В зависимости от возвращенного значения *fork*, в процессах выполнится первое или второе условие ветвления. Дочерний процесс выведет “Hello!” и затем завершится. В то же время родительский процесс остановится в системном вызове *wait(0)* и после получения кода возврата от дочернего процесса выведет строку *world* и завершится.

Приведенная в приложении анимация иллюстрирует данный пример.

В случае некорректной работы программ возможны две ситуации:

1. Дочерний процесс завершил свою работу, отправил сигнал об окончании, однако родительский процесс еще не выполнил вызов *wait()*, ожидающий данный сигнал. Таким образом, дочерний процесс уже «умер» (работа выполнена, память освобождена), но не «погребен» (информация об окончании работы не получена родителем, номер дочернего процесса все еще в списке существующих). Процесс в таком состоянии называется «зомби-процессом».
2. Родительский процесс завершил свою работу, но процесс потомок все еще выполняется. Такой процесс «без родителя» называется «процессом-сиротой». В данном случае, система назначает его родителем процесс *init*.

Вторым способом запуска процесса в Unix является вызов функции из семейства *exec*. Функция *exec()* (*execute*) загружает и запускает другую программу. Таким образом, новая программа полностью замещает текущий процесс. Новая программа начинает свое выполнение с функции *main*.

Если говорить об управлении процессами из программ, написанных на языке C, то для получения информации об идентификаторе исполняемого процесса используется системный вызов *getpid()*, а для получения значения идентификатора родительского процесса – *getppid()*.

Прототипы системных вызовов *getpid()*, *getppid()*, *fork()* и соответствующие им типы данных описаны в системных файлах *<sys/types.h>* и *<unistd.h>*.

## 2. Организация взаимодействия процессов

Все процессы так или иначе взаимодействуют между собой. Самым простым способом обмена информацией как между родственными процессами, так и в рамках одного процесса, является неименованные каналы (*pipe*, труба, конвейер). Данный механизм реализует потоковую модель передачи данных только в одну сторону и представляет из себя область памяти, которая недоступна пользовательским процессам напрямую, а доступ ко «входу» и «выходу» осуществляется через специально объявленные дескрипторы. Каналы удобны тем, что позволяют обмениваться данными программам, которые первоначально не были написаны для этого. Например, в команда в терминале:

```
ls | grep x
```

запускает две утилиты *ls* и *grep* одновременно и при этом за счет использования спецсимвола *|* соединяет вывод первой утилиты со входом второй. В приведенном примере *ls* выводит список файлов в каталоге, а *grep* ищет строки из списка, которые содержат символ *x*, таким образом операционная система создает неименованный канал, который недоступен пользователю и существует только в рамках двух запущенных утилит.

Один канал может быть использовать для **одного** направления передачи данных и только в рамках взаимодействия родственных процессов. Для организации двунаправленной связи между родственными процессами необходимо использовать два неименованных канала.

Взаимодействие между процессами, созданными в рамках одной программы можно организовать с помощью системного вызова *pipe()*. Поясним его принцип работы на примере:

```
1.  ///***/
2.  int fd[2];
3.  pipe(fd);
4.  pid = fork();
5.  if(pid == 0)
6.      {
7.          close(fd[0]); /* Child process closes up input side of pipe */
8.          write(fd[1], string, (strlen(string)+1));
9.          // working
10.     }
11.     else
12.     {
13.         close(fd[1]); /* Parent process closes up output side of pipe */
14.         nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
15.         // working
```

```
16.         }
17.     ///***/
```

Общая идея данной программы – передача сообщения от дочернего процесса к родительскому.

В строчке 2 создается файловый дескриптор, представляющий из себя массив из двух элементов. При инициализации канала (строка 3) первый элемент дескриптора открывается на чтение, а второй – на запись. После вызова `fork` (строка 4), каждый из процессов стал обладателем значения `fd`. В строках 7-9 дочерний процесс закрывает дескриптор чтения (т.к. мы будем вести запись данных) и с помощью системного вызова `write` записывает некоторую информацию. В родительском процессе (строки 13-15) наоборот, происходит закрытие дескриптора записи и вызов `read` для приема сообщения.

### Практическое задание.

Следующие задания рекомендуется выполнять последовательно. Результаты работы программ и ответы на вопросы занесите в отчет.

#### 3.1. Скомпилируйте и запустите следующий код.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
    printf("PID= %d\n", getpid());
    printf("PPID= %d\n", getppid());
    return 0;
}
```

Данная программа возвращает значение PID и PPID процесса.

*Запустите программу несколько раз. Какие значение изменились? С помощью утилиты **ps** определите родителя запущенных вами процессов.*

#### 3.2. Скомпилируйте и запустите следующую программу.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main()
{
    pid_t PID=fork();
    if(PID==0)
    {
        printf("Hello ");
        sleep(30);
    }
    else
    {
        wait(0);
        printf("world!\n");
    }
    return 0;
}
```

*Запустите в соседнем окне еще один терминал и выполните команду “**ps -ao user,pid,ppid,rss,stat,cmd**”. Сколько процессов было порождено запуском программы? Объясните значения в полях **PID** и **PPID**.*

#### 3.3. Скомпилируйте две программы – *Parent.c* и *Child.c*. Обратите внимание на то, что название программы *Child.c* должно быть *child*.

### *Child.c*

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

int main()
{
    printf("I am Child!\n");
    sleep(30);
    return 0;
}
```

### *Parent.c*

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

int main()
{
    execl("child","child", NULL);
    printf("I am Parent!\n");
    sleep(30);
    return 0;
}
```

*Запустите программу **Parent**. Обратите внимание на сообщение, выведенное на экран. Аналогично предыдущему пункту проанализируйте результат вывода **ps**.*

3.4. Скомпилируйте и запустите следующий код.

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

int main()
{
    pid_t pid = fork();
    if (pid==0)
    {
        printf("PID = %d, PPID = %d\n", getpid(), getppid());
        sleep(2);
        printf("PID = %d, PPID = %d\n", getpid(), getppid());
    }
    else
    {
        sleep(1);
        exit(0);
    }
}
```

*Проанализируйте возвращенные значения **PID** и **PPID** в первой и второй строчке. Объясните смоделированную ситуацию.*

3.5. Скомпилируйте и запустите следующий код.

```

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

int main()
{
    pid_t pid = fork();
    if (pid==0)
    {
        printf("I am alive!!\n");
    }
    else
    {
        sleep(30);
    }
    return 0;
}

```

*Аналогично проанализируйте результат вывода ps. Обратите внимание на столбцы STAT, RSS. Объясните их значения в запущенном вами процессе и порожденным им дочернем. Дождитесь завершения процесса и запустите утилиту ps еще раз. Объясните полученное значение.*

3.6. Скомпилируйте и запустите следующий код.

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

int main(void)
{
    int    fd[2];
    pid_t  pid;
    char    string[] = "Hello, Parent!\n";
    char    readbuffer[80];
    pipe(fd);
    pid = fork();
    if(pid == 0)
    {
        close(fd[0]);
        write(fd[1], string, (strlen(string)+1));
        exit(0);
    }
    else
    {
        close(fd[1]);
        read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Parent received: %s", readbuffer);
    }

    return(0);
}

```

*Проанализируйте полученный результат. Измените программу так, чтобы происходил двойной обмен сообщениями – после получения строки от дочернего процесса, он возвращал строку «Hello, Child».*

3.7. Напишите программу, порождающую три процесса, которые по кругу обмениваются сообщением приветствия.



### Контрольные вопросы

1. В чем отличие процесса от программы?
2. В чем отличие между системными вызовами `fork` и `exec`?
3. Что такое каналы в ОС? Для чего они предназначены?
4. Что будет выведено на экран в результате работы следующей программы:

```
1.  /***/  
    fork();  
    fork();  
    fork();  
    printf("Hello world!");  
    /***/
```

### Список рекомендованной литературы

1. Э. Таненбаум и Х. Бос, Современные операционные системы, Санкт-Петербург: Питер, 2021, с. 1119.
2. В. Е. Карпов и К. А. Коньков, Основы операционных систем, Москва: Физматкнига, 2019, с. 326.
3. Р. Лав, Ядро Linux. Описание процесса разработки, Москва, Санкт-Петербург, Киев: Вильямс, 2013, с. 51 - 73.
4. В. Столлингс, Операционные системы. Внутренняя структура и принципы проектирования, Москва: Диалектика, 2020, с. 1261.
5. Р. Арпачи-Дюссо и А. Арпачи-Дюссо, Операционные системы. Три простых элемента, Москва: ДМК, 2021, с. 730.

# Практическая работа 3. Потоки в операционной системе

## Теоретическая часть

### Понятие потока в ОС

В предыдущей лабораторной работе было дано определения процесса. Напомним, что процесс - программа во время исполнения или объект, которому выделяются ресурсы вычислительной системы, такие как процессорное время, память и т. д. Каждый процесс, содержит в себе набор последовательно выполняемых инструкций. Такой набор называется **поток**ом (*thread*). Не нужно путать его со стандартными потоками ввода-вывода (*streams*). Они отвечают за коммуникацию между процессами.

Каждый процесс как минимум состоит из одного потока. В настоящее время очень популярным решением является применение многопоточности в программировании – создание в рамках одного процесса нескольких потоков, работающих параллельно (квазипараллельно).

Представим следующую ситуацию. Нам необходимо написать программу, которая останавливает свою работу по нажатию кнопки. Если программа будет работать всего в один поток, то придется вместе с логикой программы производить опрос кнопки. Это не является эффективным решением, т. к. в момент нажатия кнопки, возможно выполнение кода программы, а не команды опроса. Таким образом нажатие будет пропущено. Гораздо более эффективным способом является разбиение программы на два потока – один отвечает за логику работы, а второй за работу с кнопкой.

В современном мире растет число процессоров, содержащих в себе несколько ядер. Применяя параллельное программирование, мы можем ускорить работу нашей программы в несколько раз! Однако, обратим внимание на то, что созданные потоки не обязательно будут выполняться параллельно на разных ядрах. На процессоре с одним ядром также возможна многопоточность – потоки будут исполняться квазипараллельно. Другими словами, часть процессорного времени будет выполняться один поток, часть – другой. Эти части настолько малы, что для пользователей создается иллюзия параллельности. Существует высказывание – «*Concurrency is not parallelism*». Если перевод слова *parallelism* не требует пояснений, то *Concurrency* обычно переводят как многопоточность. Т.е. «многопоточность  $\neq$  параллельность».

Все потоки создаются в рамках одного процесса. Таким образом, потоки делят одно адресное пространство. Если внутри процесса будет создана глобальная переменная, то она будет видна как из одного потока, так и из другого. Попытка одновременного изменения этой переменной несколькими потоками приведет к неопределенному состоянию. Неизвестно что произошло ранее – запись первым потоком или запись вторым. (Рисунок 1)

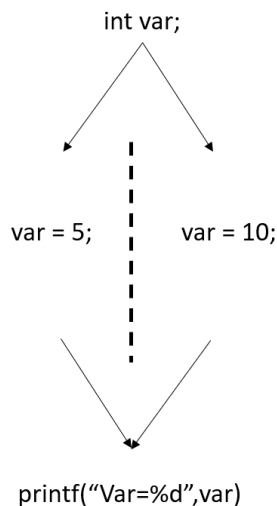


Рисунок 2 Неопределенное состояние

Ситуация, когда две инструкции из разных потоков, одна из которых – инструкция записи, пытаются получить доступ к одной ячейки памяти называется *гонкой данных* или *data race*. Пример, приведенный выше – пример гонки данных. Для безопасной работы с потоками существуют специальные механизмы синхронизации, речь о которых пойдет в следующей лабораторной работе.

## Организация потоков в ОС Linux

Для создания переносимых многопоточных программ был разработан специальный стандарт *POSIX.1c*. Пакет, предназначенный для работы с потоками, называется *pthread*. Таким образом, потоки, созданные с помощью данного пакета, часто называют POSIX потоками. Он поддерживается большинством UNIX систем, однако по умолчанию не поддерживается Windows.

Для создания и управления потоками в C используются следующие основные функции из заголовочного файла «*pthread.h*»:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg) -  
функция создания потока;  
void pthread_exit(void *retval) - функция для завершения потока;  
int pthread_join (pthread_t THREAD_ID, void ** DATA) - функция для ожидания потока;
```

## Механизмы синхронизации потоков

Для корректной работы многопоточной программы необходимо синхронизировать работу потоков. Для этого существуют несколько стандартных механизмов. Наиболее распространенным и популярным является мьютекс (*mutex*).

*Mutex* – (от англ. *mutual exclusion* — «взаимное исключение») - примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода. Мьютекс можно представить в виде переменной, которую можно перевести в одно из двух состояний – заблокировано и разблокировано. При входе в критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние.

Для работы с мьютексами используются следующие функции:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) - инициализация мьютекса
int pthread_mutex_lock(pthread_mutex_t *mutex) - захват мьютекса
int pthread_mutex_unlock(pthread_mutex_t *mutex) - освобождение мьютекса
```

Помимо обычных мьютексов используются рекурсивные мьютексы. Данный тип отличается от обычных мьютексов тем, что его возможно захватить несколько раз одним и тем же потоком. Инициализация в POSIX происходит следующим образом:

```
pthread_mutex_t Mutex;
pthread_mutexattr_t Attr;

pthread_mutexattr_init(&Attr);
pthread_mutexattr_settype(&Attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&Mutex, &Attr);
```

## Интерфейс OpenMP

OpenMP — это библиотека для параллельного программирования вычислительных систем с общей памятью. С ее помощью возможно без особых затрат распараллелить код программы на несколько потоков.

Для использования библиотеки OpenMP вам необходимо подключить заголовочный файл "omp.h", а также добавить опцию сборки -fopenmp (для компилятора gcc).

После запуска программы создается единственный процесс с единственным потоком, который начинается выполняться, как и обычная последовательная программа. Встретив параллельную область (задаваемую директивой #pragma omp parallel) процесс порождает ряд потоков (их число можно задать явно, однако по умолчанию будет создано столько потоков, сколько в вашей системе вычислительных ядер). Границы параллельной области выделяются фигурными скобками, в конце области потоки уничтожаются. В следующем примере программа выведет на экран сообщение внутри параллельной области несколько раз.

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

int main()
{
    printf("Hello!\n");
    #pragma omp parallel
    {
        printf("I'm in the parallel section!\n");
    }
    printf("Goodbye!\n");
    return 0;
}
```

Результат работы программы:

```
os@os-VirtualBox:~/Desktop/lab3$ ./omp
Hello!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
Goodbye!
```

Запуск данной программы производился на 6 ядерном процессоре. Таким образом, мы видим 6 записей в параллельной секции.

## Практическая часть

В ходе выполнения практического задания вам необходимо провести эксперименты по расчету численного значения определенного интеграла при различных условиях.

В качестве интеграла предлагается взять следующий:  $I = \int_0^1 \frac{4}{1+x^2} dx$

За число разбиений интеграла для вычисления примите значения  $10^3$ ,  $10^6$ ,  $10^9$ . Результаты экспериментов необходимо заносить в отчет. Полученное время выполнения программ рекомендуется свести в таблицу 1, приведенную ниже.

В отчет необходимо добавить графики из задания 3.

Также необходимо указать характеристики устройств, на которых производятся вычисления – название процессора, число ядер, тактовые частоты. Для этого возможно воспользоваться программой CPU-Z.

Для корректной работы на виртуальной машине необходимо включить возможность доступа к нескольким ядрам процессора. Для этого в настройках виртуальной машины (Настройки → Система → Процессор) выберите несколько ядер для работы.

### 4.1. Непосредственное вычисление интеграла.

Изучите приведенный ниже код:

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

//Input arguments:
//1: Начало интервала
//2: Конец интервала
//3: Число разбиений
//
//example:
// ./integral 0 10 1000000

double integrate(double a, double b, int n) {

    //тут будет ваш код

}

int main(int argc, char* argv[])
{
    double I;
    if (argc != 4) {
        fprintf(stderr, "Not enough arguments\n");
        exit(1);
    }
    long int N = atoi(argv[3]);
    I = integrate(strtod(argv[1], 0), strtod(argv[2], 0), N);
    printf("I = %f\n", I);

    return 0;
}
```

*Создайте файл `integral.c` и скопируйте в него данную программу. Дополните функцию `rectangle_integral`, чтобы она возвращала значение посчитанного интеграла. Скомпилируйте проект и запустите его на виртуальной машине.*

*Для компиляции воспользуйтесь следующим ключом:*

**gcc integral.c -o integral -lm**

*Сверьте решение, полученное вашей программой с точным решением. (Например, полученным с помощью онлайн калькулятора)*

*С помощью утилиты `time` замерьте время выполнения вашей программы*

**`time ./integral 0 10 1000000`**

*Требуемое значение будет выведено в строке `real`.*

#### 4.2. Разбиение вычисления выполнения интеграла на несколько процессов

Изучите приведенный ниже код:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <math.h>

double integrate(double a, double b, double n) {

    //тут будет ваш код

}

int main(int argc, char* argv[])
{
    double I;
    if (argc != 4) {
        fprintf(stderr, "Not enough arguments\n");
        exit(1);
    }
    long int N = atoi(argv[3]);

    int n;
    int fds[2];
    pid_t pid;
    pipe(fds);
    pid = fork();
    if (pid != (pid_t)0)
    {
        double r1 = 0, r2 = 0;
        int s;
        close(fds[1]);
        r1 = integrate(strtod(argv[1], 0), strtod(argv[2], 0) / 2, N / 2);
        read(fds[0], &r2, sizeof(double));
        close(fds[0]);
        printf("I = %g\n", r1 + r2);
        wait(&s);
        return 0;
    }
    else
    {
        double s;
        close(fds[0]);
        s = integrate(strtod(argv[2], 0) / 2, strtod(argv[2], 0), N / 2);
        write(fds[1], &s, sizeof(double));
        close(fds[1]);
        return 0;
    }
}
```

*Аналогично заданию 1 дополните код программы. Запустите ее и убедитесь, что значение рассчитано верно.*

*Замерьте время выполнения программы. Сравните с п.1 Сделайте выводы.*

#### 4.3. Вычисление интеграла с помощью POSIX threads

Изучите приведенный ниже код:

```

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>

double integrate(double a, double b, double n) {

    //тут будет ваш код

}

struct IntegrateTask { // Шаблон для структуры "Задача потока"
    double from, to, step, res; // интегрировать "от" (from), "до" (to), с "шагом" (step),
результат сохранить в res
};

void* integrateThread(void* data) { // ф-я приведения типов задания и т.д.
    struct IntegrateTask* task = (struct IntegrateTask*)data; // объявления структуры task и
присвоения аргументов
    task->res = integrate(task->from, task->to, task->step); // вызов ф-и интегрирования с
передачей параметров (задача)

    pthread_exit(NULL); // завершение потока
}

int main(int argc, char* argv[]) {

    double I;
    if (argc != 5) {
        fprintf(stderr, "Not enough arguments\n");
        exit(1);
    }
    long int N = atoi(argv[3]);
    long int NUM_THREADS = atoi(argv[4]);
    pthread_t threads[NUM_THREADS]; // Объявляем массив структур потоков (системные)
    struct IntegrateTask tasks[NUM_THREADS]; // Объявляем массив структур заданий потокам

    struct IntegrateTask mainTask = { strtod(argv[1],0),strtod(argv[2],0),N / NUM_THREADS }; //
Общее задание, интегрировать от 0 до 10 с шагом 0.0000001
    double distance = (mainTask.to - mainTask.from) / NUM_THREADS; // Делим общее задание на
части

    int i;

    for (i = 0; i < NUM_THREADS; ++i) // создаем задания и потоки
    {
        tasks[i].from = mainTask.from + i * distance; // задаем "от"
        tasks[i].to = mainTask.from + (i + 1) * distance; // задаем "до"
        tasks[i].step = mainTask.step; // задаем "шаг"

        pthread_create(&threads[i], NULL, integrateThread, (void*)& tasks[i]); // создание
потоков и передача параметров (задания)
    }

    double res = 0;
    for (i = 0; i < NUM_THREADS; ++i)
    { // Барьер
        pthread_join(threads[i], NULL); // ждем завершения потока
        res += tasks[i].res; // суммируем результаты
    }
    printf("I = %lf \n", res);

    return 0;
}

```

*Аналогично заданию 1 дополните код программы.*

*Скомпилируйте ее с помощью следующей команды:*

**gcc integral.c -o integral -lm -lpthread**

*Запустите ее и убедитесь, что значение рассчитано верно.*



Рассчитайте время выполнения программы при различном числе потоков – от 1 до 8. Постройте **график** зависимости время выполнения от числа потоков.

Сравните время выполнения программы, выполняемой двумя потоками с программой из задания 2. Объясните полученный результат.

#### 4.4. Использование директив OpenMP

Исправьте код программы из задания 1, добавив в функцию `integrate` перед циклом `for` следующую строчку:

```
#pragma omp parallel for reduction(+:sum)
```

Где `sum` – накапливаемая сумма при расчете интеграла.

Скомпилируйте программу с помощью следующей команды:

**gcc integral.c -o integral -lm -fopenmp**

Запустите программу и вычислите время выполнения. Сравните его со временем выполнения программы предыдущих заданий.

Таблица 1. Сравнение времени выполнения программ

| Задание  | Время выполнения (сек) |                   |                   |
|--|------------------------|-------------------|-------------------|
|  | n=10 <sup>3</sup>      | n=10 <sup>6</sup> | n=10 <sup>9</sup> |
| Последовательное выполнение (задания 1)        |                        |                   |                   |
| Выполнение двумя процессами (задание 2)        |                        |                   |                   |
| Выполнение с помощью POSIX threads (задание 3) |                        |                   |                   |
| Число потоков                                  |                        |                   |                   |
| 1  |                        |                   |                   |
| 2  |                        |                   |                   |
| 3  |                        |                   |                   |
| 4  |                        |                   |                   |
| 5  |                        |                   |                   |
| 6  |                        |                   |                   |
| 7  |                        |                   |                   |
| 8  |                        |                   |                   |
| Выполнение с помощью OMP (задание 4)           |                        |                   |                   |

#### Контрольные вопросы

1. Какие основные отличия между процессами и потоками?
2. В чем преимущества использования потоков?
3. Для чего используется интерфейс OpenMP?

### **Список рекомендованной литературы**

1. Э. Таненбаум и Х. Бос, Современные операционные системы, Санкт-Петербург: Питер, 2021, с. 1119.
2. В. Е. Карпов и К. А. Коньков, Основы операционных систем, Москва: Физматкнига, 2019, с. 326.
3. Учебник по OpenMP [Электронный ресурс] // Блог программиста: [сайт]. [2018]. URL: <https://pro-prof.com/archives/4335> (дата обращения: 12.09.2022).

## Практическая работа 4. Тупики и синхронизация потоков

### Теоретическая часть

Многопоточность при разработке приложений является очень мощным инструментом для повышения эффективности программ. Однако, применение данного подхода влечет за собой множество «подводных камней», которые могут повлиять на работоспособность всей системы.

Первым подобным «подводным камнем» является проблема тупиков (deadlock). Представим ситуацию, когда несколько потоков конкурируют за один и тот же ресурс. Если данный ресурс недоступен, то потоки переводятся в режим ожидания. В случае, если ресурс не будет освобожден, ожидание становится бесконечным и программа «зависает». Подобная ситуация называется тупиком. Поток находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. [1] [2]

Примером тупика служит транспортная пробка, представленная на Рисунок 1.

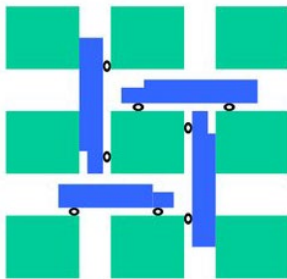


Рисунок 3 Транспортная пробка

В данной ситуации машины могут двигаться только вперед и разъехаться будет невозможно.

Вторым подводным камнем является проблема синхронизации потоков. В предыдущей лабораторной работе мы убедились, что без средств синхронизации невозможна корректная работа многопоточной программы, использующей глобальные переменные. Наиболее популярным средством синхронизации являются мьютексы. В ОС Linux в своей основе мьютексы содержат аппаратную поддержку, представленную в виде *spinlock* [1], основанного на алгоритме MCS [1]. В данном случае используется специальная ассемблерная инструкция, позволяющая производить атомарную операцию. В старых версиях Linux подобной инструкцией была *tsl* – test-and-set instruction. [3]

Однако существуют и программные алгоритмы синхронизации потоков. Рассмотрим наиболее популярные из них.

#### 1. Алгоритм Деккера

|        |   |   |
|--------|---|---|
| lock   | <pre> graph TD     Start([lock<br/>(Integer i)]) --&gt; J["j = 1-i"]     J --&gt; FlagJ{"flag[j] = 1"}     FlagJ -- True --&gt; TurnJ1{"turn = j"}     TurnJ1 -- True --&gt; FlagI0["flag[i] = 0"]     FlagI0 --&gt; TurnJ2{"turn = j"}     TurnJ2 -- True --&gt; FlagI0     TurnJ2 -- False --&gt; FlagI1["flag[i] = 1"]     FlagJ -- False --&gt; End([End])     FlagI1 --&gt; End </pre> | <p>Алгоритм реализуется при помощи двух флагов и переменной <i>turn</i>.</p> <p>При захвате мьютекса потоком, переменной <i>j</i> присваивается значение, равное 1 – номер текущего потока, переданного в качестве параметра – номера конкурирующего потока. Далее до тех пор, пока значение флага <b>конкурирующего</b> потока равен 1 мы проводим проверку условия очереди. Если значение очереди соответствует номеру <b>конкурирующего</b> потока, то мы выставляем флаг <b>текущего</b> потока в 0 и заходим в бесконечный цикл, до тех пор, пока значение очереди не изменится. Далее значение флага <b>текущего</b> потока возвращается в 1.</p> |
| unlock | <pre> graph TD     Start([unlock<br/>(Integer i)]) --&gt; Turn["turn = 1-i"]     Turn --&gt; FlagI["flag[i] = 0"]     FlagI --&gt; End([End]) </pre>  | <p>Значению переменной <i>turn</i> присваивается номер <b>конкурирующего</b> потока. Значение флага готовности опускается в 0.</p>  |

## 2. Алгоритм Петерсона

|      |  |   |
|------|--|---|
| lock | <pre> graph TD     Start([lock<br/>(Integer i)]) --&gt; J["j = 1-i"]     J --&gt; FlagI["flag[i] = 1"]     FlagI --&gt; Turn["turn = j"]     Turn --&gt; Decision{"flag[j] == 1 &amp;&amp; turn == j"}     Decision -- True --&gt; Decision     Decision -- False --&gt; End([End]) </pre> | <p>Алгоритм реализуется при помощи двух флагов и переменной <i>turn</i>.</p> <p>При захвате мьютекса потоком, переменной <i>j</i> присваивается значение, равное 1 – номер текущего потока, переданного в качестве параметра – номера конкурирующего потока. Значение флага <b>текущего</b> потока выставляется в 1, а значению очереди присваивается номер <b>конкурирующего</b> потока.</p> <p>Далее до тех пор, пока значение флага <b>конкурирующего</b> потока равно 1 и значение очереди равно номеру <b>конкурирующего</b> потока мы находимся в ожидании в бесконечном цикле.</p> |
|------|--|---|

|        |   |   |
|--------|---|---|
| unlock | <pre> graph TD     A([unlock<br/>(Integer i)]) --&gt; B[flag[i] = 0]     B --&gt; C([End]) </pre> | Значению переменной флага готовности <b>текущего</b> потока опускается в 0. |
|--------|---|---|

### 3. Алгоритм пекаря (булочной)

|        |   |  |
|--------|---|--|
| lock   | <pre> graph TD     A([lock<br/>(Integer i)]) --&gt; B[choosing[i] = 1]     B --&gt; C[ticket[i] = max(ticket[0], ticket[1]) + 1]     C --&gt; D[choosing[i] = 0]     D --&gt; E[j = 0 to Nthreads]     E -- Next --&gt; F{choosing[i] = 1}     F -- True --&gt; F     F -- False --&gt; G{ticket[i] &amp;&amp; ((ticket[i] &lt; ticket[j])    ((ticket[i] = ticket[j]) &amp;&amp; j &lt; i))}     G -- True --&gt; G     G -- False --&gt; H([End])     E -- Done --&gt; H </pre> | <p>При входе в критическую секцию каждому потоку назначаются номера. Номер пришедшего потока будет на 1 больше, чем номер предыдущего.</p> <p>Общий счётчик показывает номер исполняемого в данный момент потока. Все остальные потоки ждут, пока не закончится обслуживание текущего клиента и счетчик покажет следующий номер.</p> <p>Первым шагом флаг <i>choosing</i> текущего потока выставляется в 1. После происходит получение билета – на один больше уже получивших и опускание флага <i>choosing</i>. Далее в цикле для всех потоков проходит проверка того, что ни один поток не находится в состоянии получения билета – значение <i>choosing</i> у всех потоков равно 0. Следующим шагом <b>текущий</b> поток ожидает, пока <b>конкурирующие</b> потоки с меньшим номером или с таким же номером, но с более высоким приоритетом (номером потока), закончат свою работу.</p> |
| unlock | <pre> graph TD     A([unlock<br/>(Integer i)]) --&gt; B[ticket[i] = 0]     B --&gt; C([End]) </pre>   | Происходит возврат билета – значение номера <b>текущего</b> потока приравнивается 0  |

## Практическая часть

### 1. Анализ многопоточных программ.

1.1. Создайте файл thread.c и скопируйте в него данную программу. Для компиляции воспользуйтесь следующим ключом:

```
gcc thread.c -o thread -lpthread
```

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

void* func(void *args) {
    printf("Hello from thread!\n");
}

int main() {
    pthread_t thread;
    printf("Hello from main!\n");
    pthread_create(&thread, NULL, func, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

*Запустите полученную программу? Какой результат был получен?  
Модернизируйте программу так, чтобы было создано 10 потоков с функцией func.*

1.2. Создайте файл data\_race.c и скопируйте в него данную программу. Для компиляции воспользуйтесь следующим ключом:

```
gcc data_race.c -o data_race -lpthread
```

```
#include <pthread.h>
#include <stdio.h>
int Global = 0;
void *Thread1() {
    Global++;
}
void *Thread2() {
    Global--;
}
int main() {
    pthread_t t1,t2;
    pthread_create(&t1, NULL, Thread1, NULL);
    pthread_create(&t2, NULL, Thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n",Global);
    return Global;
}
```

*Запустите полученную программу. Какой результат был получен?*

1.3. Проверим программу на наличие гонок данных. Для этого скомпилируем её с ключом -fsanitize=thread.

```
gcc -fsanitize=thread data_race.c -o data_race -lpthread
```

*Запустите программу. Поясните полученный результат.*

- 1.4. С помощью приведенного ниже скрипта происходит запуск программы до тех пор, пока она не выведет значение, отличное от 1 или число итераций не превысит 1000. Для этого создайте файл script.sh, сделайте его исполняемым и запустите его, в качестве параметра указав название исполняемого файла:

```
touch script.sh
nano script.sh
#скопируйте текст скрипта, сохраните его
chmod +x script.sh
./ script.sh data_race
```

Листинг скрипта:

```
#!/bin/bash
var=2
N=0
while [[ $var -eq 2 ]] && [[ $N -ne 2000 ]]
do
var=$(./$1)
N=$((N+1))
done
echo $N
```

*Какой результат вернул скрипт? Запустите его несколько раз. Прокомментируйте результат работы. Проверьте программу на гонки данных с помощью ThreadSanitizer. Модернизируйте программу таким образом, чтобы создавалось 100 потоков Thread1 и 100 потоков Thread2. Запустите программу и проанализируйте результат.*

*Добавьте в программу мьютексы. Проверьте программу на гонки данных с помощью ThreadSanitizer.*

## 2. Тупики и методы их решения

- 2.1. Скомпилируйте и запустите следующий код.

```
gcc deadlock.c -o deadlock -lpthread
```

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *function1();
void *function2();
pthread_mutex_t first_mutex; // mutex lock
pthread_mutex_t second_mutex;

int main()
{
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);
    pthread_t one, two;
    pthread_create(&one, NULL, function1, NULL);
    //sleep(1);
    pthread_create(&two, NULL, function2, NULL);
    pthread_join(one, NULL);
    pthread_join(two, NULL);
    printf("Thread joined\n");
}

void *function1()
{
    pthread_mutex_lock(&first_mutex);
    printf("Thread ONE acquired first_mutex\n");
    //sleep(3);
    pthread_mutex_lock(&second_mutex);
    printf("Thread ONE acquired second_mutex\n");
    pthread_mutex_unlock(&second_mutex);
    printf("Thread ONE released second_mutex\n");
    pthread_mutex_unlock(&first_mutex);
    printf("Thread ONE released first_mutex\n");
}

void *function2()
{
    pthread_mutex_lock(&second_mutex);
    printf("Thread TWO acquired second_mutex\n");
    pthread_mutex_lock(&first_mutex);
    printf("Thread TWO acquired first_mutex\n");
    pthread_mutex_unlock(&first_mutex);
    printf("Thread TWO released first_mutex\n");
    pthread_mutex_unlock(&second_mutex);
    printf("Thread TWO released second_mutex\n");
}

```

*Какой результат выполнения кода?*

Скомпилируйте программу с ключом *Thread Sanitizer*. Запустите программу, используя анализатор поиска тупиков.

```

gcc deadlock.c -o deadlock -lpthread -fsanitize=thread
TSAN_OPTIONS=detect_deadlocks=1:second_deadlock_stack=1 ./deadlock

```

*Какой результат выполнения кода?*

Раскомментируйте строки в коде, содержащие вызов `sleep` и заново скомпилируйте и запустите программу.

*Объясните полученный результат. Измените программу таким образом, чтобы избежать тупика. (Thread Sanitizer не должен обнаруживать ошибки)*

2.2. Скомпилируйте и запустите следующий код.



```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
static int global_var = 0;
void *function1();
pthread_mutex_t mutex;

int main()
{
    pthread_mutex_init(&mutex, NULL); // initialize the lock
    pthread_t one, two;
    pthread_create(&one, NULL, function1, NULL); // create thread
    pthread_join(one, NULL);
    printf("Global var = %d\n", global_var);
    return 0;
}

void summ_f()
{
    pthread_mutex_lock(&mutex);
    if(global_var!=5)
    {
        global_var++;
        summ_f();
    }
    pthread_mutex_unlock(&mutex);
}

void *function1()
{
    summ_f();
}

```

*Объясните полученный результат. Измените программу таким образом, чтобы избежать тупика.*

### 3. Алгоритмы синхронизации потоков.

В приложении к лабораторной работе содержатся листинги программ *mutex.c*, *Deccer.c*, *Bakery.c*, *Peterson.c*, *Deccer.h*, *Bakery.h*, *Peterson.h*.

В программе происходит вызов функции *Thread* из двух потоков. В случае запуска программы без механизмов синхронизации произойдет гонка данных и результат будет непредсказуем. Ваша задача – реализовать методы *lock* и *unlock* самостоятельно на основе одного из приведенных алгоритмов – Петерсона, Деккера и булочника.

Выбор варианта: (Номер в списке % 3) + 1.

| Вариант | Алгоритм  |
|---------|-----------|
| 1       | Петерсона |
| 2       | Деккера   |
| 3       | Булочника |

Для проверки результата запустите программу через скрипт, приведенный выше.

*Запустите программу с мьютексом и без него. Объясните полученный результат. Скомпилируйте и запустите программу с утилитой Thread Sanitizer. Какой результат был получен? Объясните почему.*

### Контрольные вопросы

1. Что такое тупики в ОС?

2. Приведите примеры тупиков?
3. Чем аппаратные алгоритмы синхронизации отличаются от программных?
4. Опишите принцип работы алгоритма Петерсона
5. Опишите принцип работы алгоритма Деккера
6. Опишите принцип работы алгоритма булочника

### **Список рекомендованной литературы**

1. Карпов В.Е., Коньков К.А. Основы операционных систем. Москва: Физматкнига, 2019. 326 pp.
2. Таненбаум Э., Бос Х. Современные операционные системы. Санкт-Петербург: Питер, 2021. 1119 pp.
3. mutex.c [Электронный ресурс] // Linux kernel code: [сайт]. URL: <https://elixir.bootlin.com/linux/latest/source/kernel/locking/mutex.c> (дата обращения: 11.10.2022).
4. MCS locks and qspinlocks [Электронный ресурс] // LWN: [сайт]. URL: <https://lwn.net/Articles/590243/> (дата обращения: 11.10.2022).
5. Test and set instruction [Электронный ресурс] // ibm: [сайт]. URL: <https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=configuration-test-set-instruction> (дата обращения: 11.10.2022).

## Практическая работа 5. Файловые системы и RAID массивы

### Введение

Все данные на любом запоминающем устройстве хранятся в виде последовательности байт. Для непосредственной работы с ними необходимо обращаться к регистрам устройства, искать адреса, по которым лежат данные и производить с ними взаимодействие. Делать все это вручную возможно, но очень сложно и неудобно. Поэтому была разработана абстракция в виде файловой системы, которая абстрагирует пользователя от устройства и предоставляет ему возможность работать с данными на более простом уровне.

### Теоретическая часть

#### Файловые системы

Дадим основные определения:

**Файл** – поименованная совокупность данных

**Файловая система** – часть ОС, отвечающая за работу с файлами.

Можно выделить основные функции файловой системы:

1. Создание, удаление, модификация файлов
2. Разделение файлов друг от друга, поддержание целостности
3. Совместная работа нескольких процессов с файлами
4. Изменение структуры файла
5. Восстановление после стирания
6. Обеспечение разных методов доступа и режима секретности
7. Обращение к файлу по символическому имени
8. Дружественный интерфейс [1]

Файловая система позволяет при помощи системы справочников (каталогов, директорий) связать уникальное имя файла с блоками памяти, содержащими данные файла. Помимо собственно файлов и структур данных, используемых для управления файлами (каталоги, дескрипторы файлов, различные таблицы распределения внешней памяти), понятие "файловая система" включает программные средства, реализующие различные операции над файлами. [2]

Над файлами возможно производить стандартные операции – открытие, закрытие, чтение, запись, удаление и др.

#### Структура файловой системы

Файловые системы хранятся на дисках. Если посмотреть на схематичное изображение диска (Рисунок 1), то можно увидеть, что его возможно разбить на набор **секторов**, участки которого будут последовательно считываться. Несколько последовательных секторов возможно объединить в один **кластер**.

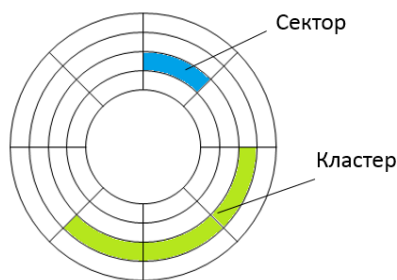


Рисунок 4 Структура диска

Большинство дисков может быть разбито на один или несколько разделов, на каждом из которых будет своя файловая система. Каждая файловая система будет содержать в себе участок, отвечающий за её загрузку и организацию содержимого в ней. Остальные участки памяти будут занимать непосредственно файлы. [3]

## Типы файловых систем

Существует множество различных типов файловых систем. У каждой из них свое устройство и предназначение. Рассмотрим наиболее популярные из них.

### FAT

**FAT** (*File Allocation Table* «таблица размещения файлов») — классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей. Используется в дискетах, картах памяти и некоторых других носителях информации. Ранее находила применение и на жёстких дисках.

В файловой системе FAT смежные секторы диска объединяются в единицы, называемые кластерами. Количество секторов в кластере равно степени двойки (см. далее). Для хранения данных файла отводится целое число кластеров (минимум один). Размер такого кластера стандартизирован и равен числу секторов в нем умноженному на число байт в таком секторе. Обычно, размер сектора равен 512 байтам. Пусть в одном кластере содержится 4 сектора, тогда его объем будет 2 Кб. Если мы захотим сохранить файл, объемом 1 байт, то произойдет выделение целого кластера размером 2 Кб, который будет практически пуст.

Для определения в каких кластерах содержатся данные файловая система содержит специальную таблицу размещения файлов – FAT. В ней каждому файлу предоставляется цепочка из номеров кластеров, в котором он содержится.

Существует четыре версии FAT — FAT12, FAT16, FAT32 и exFAT (FAT64). Они отличаются разрядностью записей в таблице размещения файлов, то есть количеством бит, отведённых для хранения номера кластера. Например, у FAT16 один кластер будет обозначаться 2 байтами.

### NTFS

**NTFS** (от англ. *new technology file system* — «файловая система новой технологии») — стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft. Как и любая другая система, NTFS делит все полезное место на кластеры — блоки данных, используемые одновременно. NTFS поддерживает почти любые размеры кластеров — от 512 байт до 64 Кбайт, неким стандартом же считается кластер размером 4 Кбайт.

Основная концепция системы состоит в том, что все данные в ней являются файлами, в том числе и служебные. Они могут располагаться в любом месте тома, вместе

с обычными файлами. Ядром NTFS является таблица MFT, содержащая информацию обо всех каталогах и файлах. Каждый из них представлен как минимум одной записью в этой таблице, где указаны все их параметры и атрибуты. Дисковое пространство, как и в FAT выделяется кластерами.

## ExtX

Во большинстве дистрибутивов Linux используется файловая система ExtX, где X – её номер. Сейчас наиболее популярной является система Ext4. Файловая система начинается с зарезервированной области, далее разбивается на несколько блоков. Блоки в свою очередь возможно объединить в секции. Базовая информация о строении файловой системы хранится в суперблоке, находящимся в самом её начале. Более подробно возможно изучить в [4]

## RAID массивы

RAID расшифровывается как Redundant Array of Inexpensive Disks или избыточный массив недорогих дисков. RAID позволяет превратить несколько физических жестких дисков в один логический жесткий диск. Существует множество уровней RAID, таких как RAID 0, RAID 1, RAID 5, RAID 10 и т. д. Рассмотрим RAID 0. Основная идея заключается в разбиении всех блоков на два диска. Таким образом, в случае повреждения первого диска, второй останется целым и половина информации сохранится (Рисунок 2).

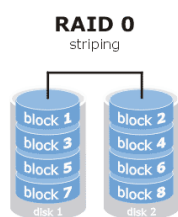


Рисунок 5 RAID 0

RAID 1, который также известен как зеркалирование дисков. RAID 1 создает идентичные копии данных. Если у вас два жестких диска объединенные в RAID 1, данные будут записаны на оба диска, т.е. на обоих дисках будут храниться одинаковые данные. Преимущество RAID 1 заключается в том, что если один из них выйдет из строя, то ваш компьютер или сервер все равно будет работать, потому что у вас есть полная неповрежденная копия данных на другом жестком диске. Вы можете вытащить неисправный жесткий диск во время работы компьютера, вставить новый жесткий диск, и он автоматически восстановит зеркало. Обратной стороной RAID 1 является отсутствие дополнительного дискового пространства. Если ваши два жестких диска имеют размер 1 ТБ, то общий полезный объем составляет 1 ТБ вместо 2 ТБ. (Рисунок 3)

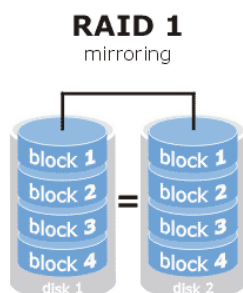


Рисунок 6 RAID 1

RAID 10 (RAID 1+0) — зеркалированный массив, данные в котором записываются последовательно на несколько дисков, как в RAID 0. Эта архитектура представляет собой массив типа RAID 0, сегментами которого вместо отдельных дисков являются массивы RAID 1. Соответственно, массив этого уровня должен содержать как минимум 4 диска (и всегда чётное количество). RAID 10 объединяет в себе высокую отказоустойчивость и производительность.

Существует два типа реализации RAID — **аппаратная** и **программная**. В случае аппаратной реализации используется контроллер жесткого диска для его создания. Контроллер жесткого диска — это карта PCIe, которую вы вставляете в компьютер. Затем вы подключаете свои жесткие диски к этой карте. При загрузке компьютера вы увидите параметр, позволяющий настроить RAID. Вы можете установить операционную систему поверх аппаратного RAID, что может увеличить время безотказной работы. Программный RAID требует, чтобы у вас уже была установлена операционная система. С одной стороны, он ничего не стоит (в отличие от аппаратных RAID-контроллеров). С другой стороны, программный RAID использует некоторое количество ресурсов центрального процессора.

### Практическая часть

В данной лабораторной работе вам предстоит научиться создавать программный RAID массив и исследовать файловую систему FAT16. Все задания выполняются последовательно, а самостоятельные упражнения выделены **зеленым цветом**. Все результаты работы необходимо заносить в отчет.

Перед тем, как приступить к практическим заданиям, необходимо установить следующие программные компоненты

```
sudo apt-get install mdadm -y
sudo apt-get install hexedit -y
```

Создание программного RAID10 массива.

Для создания RAID10 массива создадим bash скрипт с кодом, приведенным ниже.

```
#!/bin/bash

if [ $EUID -ne 0 ]
then
    echo -e "\033[31m Please, use sudo"
    exit
fi
declare -a loopname
for i in {0..3}
do
    loopname[i]=$ (losetup -f)
    dd if=/dev/zero of=test.bin.$i bs=1M count=10
    losetup ${loopname[i]} test.bin.$i
    sfdisk --force ${loopname[i]} < parts.txt
    partprobe ${loopname[i]}
done
mdadm --create /dev/md10 --level 10 --raid-devices=4 ${loopname[@]}p1
mdadm -D /dev/md10
mkfs.ext4 /dev/md10
mkdir mnt
mount /dev/md10 mnt
echo "Hello MIET" > 1.txt
cp 1.txt mnt
cat mnt/1.txt
cat /proc/mdstat
echo -e "\033[32m Press Enter to continue"
tput sgr0
```

```

read
mdadm /dev/md10 -f ${loopname[3]}p1
cat /proc/mdstat
mdadm /dev/md10 -r ${loopname[3]}p1
cat /proc/mdstat
cat mnt/1.txt
echo -e "\033[32m Press Enter to continue"
tput sgr0
read
dd if=/dev/zero of=mnt/test bs=1M count=10 ; sync
mdadm /dev/md10 -a ${loopname[3]}p1
echo -e "\033[32m Press Enter to continue"
tput sgr0
read
cat /proc/mdstat
echo -e "\033[32m Press Enter to finish"
tput sgr0
read
umount mnt
rm -rf mnt
mdadm --stop /dev/md10
losetup -d ${loopname[@]}
rm -f test.bin.{0..3}

```

Расположим файл *parts.txt*, в той же директории, где расположен скрипт.

Содержимое файла:

```

unit: sectors
/dev/loop0p1 : start= 2048, size= 1046528, Id=f0
/dev/loop0p2 : start= 0, size= 0, Id= 0
/dev/loop0p3 : start= 0, size= 0, Id= 0
/dev/loop0p4 : start= 0, size= 0, Id= 0
EOF

```

Запустим данный скрипт с помощью следующей команды

```
sudo ./script
```

В данном скрипте происходит создание образа файловой системы, состоящей из 4 виртуальных жестких дисков. Далее образ монтируется в папку *mnt*. После этого создается файл *1.txt* и копируется на смонтированный диск. Результат должен появиться на экране. В случае появления сообщения «*Continue creating array?*» нажмите «Y»

С помощью команды *cat /proc/mdstat* возможно посмотреть состояние RAID массива на текущий момент.

Результат ее работы будет приблизительно следующим:

```

Personalities : [raid10]
md10 : active raid10 loop0p1[4] loop3p1[3] loop2p1[2] loop1p1[1]
      14336 blocks super 1.2 512K chunks 2 near-copies [4/4] [UUUU]

unused devices: <none>

```

В данном случае видно, что был создан массив RAID10 и все 4 диска находятся в хорошем состоянии. За это отвечают символы [UUUU]

Нажмем на клавишу Enter и продолжим выполнение скрипта. С помощью следующих команд один из дисков был отмечен, как испорченный, а после удален.

```
mdadm: set /dev/loop0p1 faulty in /dev/md10
Personalities : [raid10]
md10 : active raid10 loop0p1[4](F) loop3p1[3] loop2p1[2] loop1p1[1]
      14336 blocks super 1.2 512K chunks 2 near-copies [4/3] [_UUU]

unused devices: <none>
mdadm: hot removed /dev/loop0p1 from /dev/md10
Personalities : [raid10]
md10 : active raid10 loop3p1[3] loop2p1[2] loop1p1[1]
      14336 blocks super 1.2 512K chunks 2 near-copies [4/3] [_UUU]
```

Как видно из вывода на экран, один из дисков исчез [\_UUU]. Однако, сообщение текстовое сообщение все равно возможно прочитать, т.е. система продолжает работать.

Далее нажмем Enter и восстановим наш массив. Еще раз нажав клавишу, убедимся в том, что все диски появились обратно.

*Изменив скрипт, создайте аналогичным образом RAID5. Заметили ли вы разницу при выполнении скриптов? Если да, то какую?*

### Работа с файловой системой FAT16

Создадим файл img.fat, который далее разметим файловой системой

```
dd if=/dev/zero of=img.fat bs=1024 count=10000
```

Разметим файл с помощью команды mkfs

```
mkfs -t vfat ./img.fat
```

Создадим папку, к которой будем монтировать созданную файловую систему.

```
mkdir 1
```

Монтируем файловую систему

```
sudo mount img.fat 1
```

С помощью следующего скрипта попытаемся скопировать в созданную директорию 600 файлов с текстом HelloWorld.

```
for i in {1..600}; do echo -n HelloWorld >> $i.text; sudo cp $i.text 1; rm $i.text; done
```

*Проанализируйте результат выполнения команды. Сколько файлов удалось скопировать? Почему?*

Удалите все созданные файлы и создайте директорию dir\_1. Смонтируйте в нее файловую систему img.fat (см пункты 2.2.3-2.2.4)

Запустите следующий скрипт, создающий пустые файлы и копирующий их на диск.

```
for i in {1..600}; do touch $i; sudo cp $i 1; rm $i; done
```

*Сколько файлов удалось скопировать? Попробуйте объяснить почему. Если не получится – продолжите выполнять задания дальше. Подсказка будет в одном из следующих пунктов.*



Удалим созданный файл `img.fat` и директории. Далее создадим из заново, повторив пункты 2.2.1, 2.2.2, 2.2.3, 2.2.4.

Создадим внутри созданной директории несколько файлов различных размеров: 100 байт, 1000 байт, 10000 байт. Для этого возможно приведенным ниже скриптом, где  $N$  – число создаваемых байт/10.

```
for i in {1..N}; do echo -n 0123456789 >> 1.txt; done; sudo cp 1.txt 1
```

Перейдем в созданную директорию. Убедимся, что файлы были созданы. Сохраним результат вывода следующей команды в отчет.

```
ls -l --full-time
```

Демонтируем файловую систему

```
sudo umount 1
```

Перейдем в директорию `1` и убедимся, что созданные файлы исчезли.

Откроем файл `img.fat` с помощью 16ого редактора.

```
hexedit img.fat
```

Схематично, файловую систему возможно представить следующим образом (Рисунок 4):



Рисунок 7 Файловая система FAT16

По мере выполнения заданий, заполните пропущенные значения в Таблица 1:

Таблица 1 Адреса файловой системы

| Файловая система   | Диапазон адресов:       |
|--------------------|-------------------------|
| Загрузочный сектор | 0x00000000 – 0x???????? |
| FAT Копия 1        | 0x???????? – 0x???????? |
| FAT Копия 1        | 0x???????? – 0x???????? |
| Корневой каталог   | 0x???????? – 0x???????? |
| Область данных     | 0x???????? – 0x???????? |

Размер сектора:  
 Размер кластера:  
 Общее число секторов:  
 Адрес первого кластера, содержащего файл 1.txt:  
 Адрес первого кластера, содержащего файл 2.txt:  
 Адрес первого кластера, содержащего файл 3.txt:

Рассмотрим загрузочный сектор. Его содержимое будет приблизительно таким:

```

00000000 EB 3C 90 6D 6B 66 73 2E 66 61 74 00 02 04 04 00    .<.mkfs.fat.....
00000010 02 00 02 20 4E F8 14 00 20 00 40 00 00 00 00 00    ... N... .@.....
00000020 00 00 00 00 80 00 29 B9 D0 EC B7 4E 4F 20 4E 41    .....).NO NA
00000030 4D 45 20 20 20 20 46 41 54 31 36 20 20 20 0E 1F    ME FAT16 ..
00000040 BE 5B 7C AC 22 C0 74 0B 56 B4 0E BB 07 00 CD 10    .[|."t.V.....
00000050 5E EB F0 32 E4 CD 16 CD 19 EB FE 54 68 69 73 20    ^..2.....This
00000060 69 73 20 6E 6F 74 20 61 20 62 6F 6F 74 61 62 6C    is not a bootabl
00000070 65 20 64 69 73 6B 2E 20 20 50 6C 65 61 73 65 20    e disk. Please
00000080 69 6E 73 65 72 74 20 61 20 62 6F 6F 74 61 62 6C    insert a bootabl
00000090 65 20 66 6C 6F 70 70 79 20 61 6E 64 0D 0A 70 72    e floppy and..pr
000000A0 65 73 73 20 61 6E 79 20 6B 65 79 20 74 6F 20 74    ess any key to t
000000B0 72 79 20 61 67 61 69 6E 20 2E 2E 2E 20 0D 0A 00    ry again ... ...
...
  
```

Наиболее интересными для нас будут следующие поля (Рисунок 5):

|      | 0                                    | 1  | 2 | 3                        | 4 | 5            | 6                                  | 7 | 8 | 9 | A | B                         | C | D                              | E   | F |
|------|--------------------------------------|--|---|--------------------------|---|--------------|------------------------------------|---|---|---|---|---------------------------|---|--------------------------------|---|---|
| 0x0  | Команда перехода к загрузочному коду |  |   | Имя OEM                  |   |              |                                    |   |   |   |   | Количество байт в секторе |   | Количество секторов в кластере | Размер зарезервированной области в секторах |   |
| 0x10 | Количество копий FAT                 | Максимальное количество файлов в корневом каталоге |   | Количество секторов в ФС |   | Тип носителя | Размер каждой копии FAT в секторах |   | - | - | - | -                         | - | -                              | -   | - |

Рисунок 8 Поля загрузочного сектора

Обратим внимание, что некоторые поля записаны в обратном порядке. Например, в данном случае, количество байт в секторе (12 и 13 байты) будет равно 0x00 0x02 -> поменяем местами -> 0x02 0x00 ->  $200_{16}=512_{10}$ . Т.е. 512 байт. Загрузочный сектор оканчивается последовательностью символов 0x55AA. FAT таблица располагается сразу после зарезервированной области.

Определите адрес, по которому она расположена. Внесите ответ в таблицу выше. **Решение** поместите в отчет.

Далее после FAT таблицы и ее копии располагается корневой каталог.

Вычислите его адрес. Внесите ответ в таблицу выше. **Решение** поместите в отчет.

Перейдите к корневому каталогу.

Одна запись в нем будет выглядеть приблизительно следующим образом:

|          |   |                  |
|----------|---|------------------|
| 00005800 | 41 31 00 2E 00 74 00 78 00 74 00 0F 00 89 00 00 | A1...t.x.t.....  |
| 00005810 | FF FF FF FF FF FF FF FF FF FF 00 00 FF FF FF FF | .....            |
| 00005820 | 31 20 20 20 20 20 20 20 54 58 54 20 00 9C 17 9A | 1 TXT ....       |
| 00005830 | 75 53 75 53 00 00 17 9A 75 53 03 00 36 01 00 00 | uSuS....uS..6... |

На первых двух строчках располагается обозначение имени файла. Следующие две занимает описание его характеристик и параметров.

Запись в нем устроена следующим образом (Рисунок 6):

|    | 0                      | 1 | 2  | 3 | 4                  | 5 | 6                                | 7 | 8                               | 9 | A                         | B              | C            | D                             | E                          | F |
|----|------------------------|---|--|---|--------------------|---|----------------------------------|---|---------------------------------|---|---------------------------|----------------|--------------|-------------------------------|----------------------------|---|
| 0  | Имя файла              |   |  |   |                    |   |                                  |   | Тип файла                       |   |                           | Атрибу-<br>ты* | Резерв       | 10ms<br>создан<br>ия<br>файла | Время<br>создания<br>файла |   |
| 10 | Дата создания<br>файла |   | Дата<br>последнего<br>обращения к<br>файлу |   | Не<br>используется |   | Время<br>последнего<br>изменения |   | Дата<br>последнего<br>изменения |   | Номер первого<br>кластера |                | Размер файла |                               |                            |   |

Атрибуты

| Бит      | 7 | 6 | 5     | 6              | 3             | 2                  | 1               | 0                |
|----------|---|---|-------|----------------|---------------|--------------------|-----------------|------------------|
| Значение | 0 | 0 | Архив | Директор<br>ия | Метка<br>тома | Системн<br>ый файл | Скрытый<br>файл | Только<br>чтение |

Рисунок 9 Поля корневого каталога

*Измените значения атрибутов записи таким образом, чтобы файл был обозначен, как директория.*

Время в дата шифруются следующим образом. Значение из поля разбивается на блоки по несколько бит (0-4 – часы, 5-10 – минуты, 11-15 – секунды), каждый из которых обозначает свой параметр.

Например, время создания данного файла можно вычислить так:

0x179A -> 0x9A17 -> 1001101000010111<sub>2</sub> -> 10011 010000 10111 -> 19 16 23 -> 19:16:46 (секунды считаются парами, поэтому умножаем значение на 2).

Аналогично расшифровывается дата. Значение из поля разбивается следующим образом: 0-6 – год, 7-10 – месяц, 11-15 – день.

Для данного примера:

0x7553 -> 0x5375 -> 101001101110101<sub>2</sub> -> 101001 1011 10101 -> 41 11 21 -> 21-11-2021 (к значению 41 прибавляется значение 1980)

*Измените у файла 2.txt дату создания на ваш день рождения и время создания на 12:34:56*

Изменим значение первого байта названия (В данном случае 0x31) файла 3.txt на 0xE5.

Сохраним измененные значения. Далее выйдем из редактора и смонтируем измененную файловую систему в каталог 1. Перейдем в него.

*Сравните содержимое каталога со значениями, полученными в пункте 2.2.10. Сделайте выводы.*

Демонтируйте файловую систему и снова войдите в Нех редактор.

По вычисленному ранее адресу перейдите к FAT таблице. Она будет представлять из себя примерно следующее:

|          |   |       |
|----------|---|-------|
| 00000800 | F8 FF FF FF 00 00 FF FF FF FF 06 00 07 00 08 00 | ..... |
| 00000810 | 09 00 FF FF 00 00 00 00 00 00 00 00 00 00 00    | ..... |

Первый байт – 0xF8 – является дескриптором, определяющим FAT таблицу. Далее следуют 3 байта со значениями FF.

Один элемент таблицы FAT равен числу, стоящую у ее названия. Т.к. мы работаем с FAT16, то одна запись занимает 16 бит – 2 байта. Нумерация начинается с 0.

В данном примере нулевая запись равна 0xF8FF, первая - 0xFFFF, вторая - 0x0000, третья – 0xFFFF и т.д. Нулевые значения обозначают, что кластер с заданным номером пуст. Числа большие нуля обозначают номер следующего кластера, в котором лежит часть нашего файла. 0xFFFF означает, что данный кластер заполнен данными и является последним в цепочке.

Вернемся к корневому каталогу и посмотрим значение номера первого кластера для файла 1.txt. Оно равно 3. Отсчитаем от начала 3 запись. Она равна 0xFFFF. Значит данный файл располагается только в одном кластере с номером 3. Аналогично файл 2.txt располагается в 4 кластере.

У файла 3.txt значение первого кластера равно 5. В данной паре ячеек лежит число 6 – номер следующего кластера. Далее лежат 7, 8, и 9. В девятой ячейке лежит значение 0xFFFF – конец файла. Таким образом, данный файл занимает 5 ячеек, соответственно – 5 кластеров.

*Вычислите адрес, по которому располагаются данные файла 1.txt. Для этого необходимо рассчитать размер корневого каталога и прибавить его к адресу начала каталога. Полученный адрес будет являться адресом **второго** кластера. Относительно него возможно получить адрес следующих кластеров, прибавляя его размер. Полученное значение будет адресом начала файла 1.txt. Внесите ответ в таблицу выше. **Решение** поместите в отчет.*

Аналогичным образом, возможно рассчитать, где располагаются остальные файлы. В случае расположения файлов в нескольких кластерах, данные объединяются в один файл.

*Аналогично получите адреса остальных файлов. Внесите ответ в таблицу выше. **Решение** поместите в отчет.*

Криминалистический анализ файловой системы.

*Вам был передан образ, снятый с диска, оставленного на месте преступления. Файл образа прилагается к лабораторной работе. Известно, что на нем содержался секретный файл, однако диск был испорчен и данные были повреждены. Ваша задача вручную восстановить секретный файл, не прибегая к специализированным программам. В отчете укажите последовательность действий и что было изменено.*

## Контрольные вопросы

1. Что такое файловая система?

2. Какие существуют типы файловых систем?
3. Чем FAT отличается от NTFS?
4. Для чего предназначаются RAID массивы?
5. Как организован RAID 1?

#### Список литературы

- [1] С. А. Балабаев, *Лекция 12 по курсу Операционные системы*, Зеленоград, 2024.
- [2] В. Е. Карпов и К. А. Коньков, *Основы операционных систем*, Интуит, 2005.
- [3] Э. Таненбаум и Х. Бос, *Современные операционные системы*, Санкт-Петербург: Питер, 2021.
- [4] Б. Кэрриэ, *Криминалистический анализ файловых систем*, Санкт-Петербург: Питер, 2007.