Лабораторная работа 1

ПРИНЦИПЫ РАЗРАБОТКИ ОПЕРАЦИОННЫХ СИСТЕМ

Цель работы — изучение основ разработки ОС, принципов низкоуровневого взаимодействия с аппаратным обеспечением, программирования системной функциональности и процесса загрузки системы.

Теоретические сведения

Принцип хранения информация на магнитных носителях

Основополагающими понятиями при адресации к данным, хранимым на магнитном диске, являются: головка, цилиндр, сектор. Если носитель состоит из нескольких дисков (например, жесткий диск), считывание и запись информации на каждый из дисков производится собственной головкой, номер которой — это номер диска носителя.

Каждый диск носителя разделен на кольцеобразные дорожки — цилиндры. Каждая дорожка в свою очередь делится на секторы, размер которых фиксирован (обычно — 512 байт). Запись/чтение диска возможно по секторам, но для этого необходимо указать номера головки, цилиндра и сектора.

Простейшим примером накопителя на магнитных дисках является гибкий магнитный диск (дискета). Для дискеты нумерация головок и цилиндров ведется с нуля, а секторов — с единицы. Первый сектор дискеты — это загрузочный сектор. Он содержит код, который считывается и выполняется при загрузке системы. В общем случае это может быть любая программа, однако, загрузочные дискеты содержат в первом секторе код загрузчика ОС. Адрес первого сектора описывается следующими числами:

```
0 (дорожка) : 0 (головка): 1 (сектор)
```

Стандартная дискета содержит 2880 секторов: 2 магнитные головки, каждый цилиндр (дорожка) содержит 18 секторов, а головка — 80 цилиндров (дорожек).

Порядок нумерации выбран таким образом, что при последовательном увеличении номера сектора вначале увеличивается номер головки, затем номер дорожки. Это сделано для сокращения перемещений блока головок при обращении к последовательным логическим номерам секторов.

Для перевода абсолютного номера сектора в "геометрический" (логический) номер можно использовать табл. 1.

Таблица 1

Соответствие абсолютных и логических номеров секторов
(для магнитных дисков объемом 1,44Мб)

Абс. номер сектора	**Смещение, байт, дес.**	**Смещение, байт, hex**	**Лог. дорожка**	**Лог. головка**	**Лог. сектор**
1	0	0x0000	0	0	1
2	512	0x0200	0	0	2
…					
18	8704	0x2200	0	0	18
19	9216	0x2400	0	1	1
20	9728	0x2600	0	1	2
…					
36	17920	0x4600	0	1	18
37	18432	0x4800	1	0	1
38	18944	0x4A00	1	0	2
…					
54	27136	0x6A00	1	0	18
55	27648	0x6C00	1	1	1
…					
72	36352	0x8E00	1	1	18
…					
2880	1474048	0x167E00	79	1	18

Процесс загрузки компьютера

После старта процессор выполняет операцию самотестирования — BIST (`built-in self test`). Затем происходит выполнение первой инструкции, находящейся по физическому адресу `0xFFFFFFF0` (в вершине адресного пространства), по которому записан код EPROM (`Erasable Programmable Read-Only Memory`), содержащий программу дальнейшей инициализации аппаратуры компьютера.

Программное обеспечение, которое получает управление сразу после загрузки компьютера, называется BIOS (`Basic Input/Output System`, Базовая Система Ввода/Вывода), задача которого — определение состава аппаратного обеспечения компьютера, проведение тестов работоспособности и запуск загрузки ОС.

Загрузчик ОС располагается в первом секторе диска. В зависимости от установок BIOS порядок устройств, на которых осуществляется поиск загрузчика, может меняться, но, в общем случае, этими устройствами могут быть: дискета, жесткий диск, привод лазерных дисков, сервер сети.

Наиболее простой способ загрузки ОС, поддерживаемый всеми версиями BIOS, — это загрузка с дискеты. Если BIOS находит дискету в дисководе, выполняется считывание ее первого сектора (512 байт) в оперативную память по физическому адресу `0x00007C00`.

Для загрузки компьютера могут быть использованы только загрузочные носители. Носитель является загрузочным, если считанный сектор содержит байты 0xAA и 0x55 по смещению 510 байт от начала

сектора (т.е. этими двумя байтами завершается сектор). Если носитель является загрузочным — BIOS передает управление коду загруженного в память сектора. Таким образом, первая исполняемая инструкция кода находится по физическому адресу загрузки сектора, т.е. `0x00007C00`.

Исполнение загрузочного кода

Архитектуру IA-32 имеют все 32-разрядные процессоры Intel. Все они являются обратно совместимыми вплоть до процессора 8086. Другое название этой архитектуры — x86. Процессоры IA-32 могут функционировать в двух основных режимах: режиме реальных адресов (или реальном режиме) и защищенном режиме.

В реальном режиме процессор функционирует практически как 8086, но может использовать 32-битные регистры. Все инструкции процессора являются 16-битными. Память адресуется с помощью пары регистров: сегментного и общего назначения, причем для вычисления физического адреса памяти сегментная часть адреса умножается на 16 и суммируется со значением регистра общего назначения. Например, при попытке считать значение по адресу 0400h:00f0h (где первая часть — хранится в сегментном регистре, вторая часть — в регистре общего назначения), процессор будет обращаться по физическому адресу 0400h * 16 + 00f0h = 40f0h. Таким способом может быть адресовано чуть более 1Мб памяти.

При включении компьютера процессор Intel находится в реальном режиме.

Можно рассмотреть пример программы, выполняемой при загрузке компьютера с дискеты и выводящей на экран приветствие. Для ее запуска необходимо создать файл *start.asm*, содержащий приведенный ниже код и скомпилировать его с помощью транслятора ассемблера.

```
use16
org 0x7C00

start:
      ; Инициализация адресов сегментов. Эти операции требуется не для любого BIOS, но их рекомендуется проводить.
      mov ax, cs ; Сохранение адреса сегмента кода в ax
      mov ds, ax ; Сохранение этого адреса как начало сегмента данных
      mov ss, ax ; И сегмента стека
      mov sp, start ; Сохранение адреса стека как адрес первой инструкции этого кода. Стек будет расти вверх и не перекроет код.
      mov ah, 0x0e ; В ah номер функции BIOS: 0x0e - вывод символа на активную видео страницу (эмуляция телетайпа)

      mov al, 'H' ; В al помещается код символа
      int 0x10    ; Вызывается прерывание. Обработчиком является код BIOS. Символ будет выведен на экран.

      mov al, 'e'
      int 0x10

      mov al, 'l'
      int 0x10
      int 0x10
```

```
      mov al, 'o'
      int 0x10

inf_loop:
      jmp inf_loop ; Бесконечный цикл

; Внимание! Сектор будет считаться загрузочным, если содержит в конце своих
512 байтов два следующих байта: 0x55 и 0xAA
      times (512 - ($ - start) - 2) db 0 ; Заполнение нулями до границы 512
- 2 текущей точки
      db 0x55, 0xAA ; 2 необходимых байта чтобы сектор считался загрузочным
```

Данный пример компилируется с помощью транслятора fasm следующей командой:

```
fasm start.asm
```

В результате будет получен бинарный файл *start.bin*. Чтобы запустить код необходимо записать полученный бинарный файл в первый сектор загрузочной дискеты (или диска), например с помощью программы *dd*, и перезагрузить компьютер. Более удобным и рекомендуемым вариантом проверки программы является запуск кода на эмуляторе QEMU командой:

```
qemu –fda start.bin
```

Данная команда запустит эмулятор QEMU виртуального IBM-совместимого компьютера, в качестве дискеты в первом дисководе гибких дисков будет размещен образ *start.bin*, полученной в результате компиляции примера.

Варианты синтаксиса языка ассемблера

Язык ассемблера — система обозначений, используемая для представления в удобочитаемой форме программ, записанных в машинном коде. Существует два основных формата записи инструкций процессора (мнемоник), называемых *синтаксис Intel* и *синтаксис AT&T* соответственно.

Особенности синтаксиса Intel:

1. В командах пересылки данных приемник данных находится слева от источника. Например, код `mov eax, ebx` пересылает (копирует) в регистр eax значение, содержащееся в регистре ebx.
2. Название регистров записывается без каких-либо префиксов. Все регистры являются зарезервированными словами (метки, функции и макросы не могут называться так же, как какой-либо регистр).
3. Константы в коде записываются без каких-либо префиксов.
4. Адрес текущего смещения в памяти — знак $ (знак доллара).
5. Доступ к памяти по адресу, расположенном в регистре, осуществляется с помощью записи [reg+N], где N — смещение в байтах в памяти относительно адреса, хранящегося в регистре reg. N можно упустить. Например, команда `mov al, [bx]` копирует один байта из памяти с адресом, хранящимся в регистре bx в регистр al.

Особенности синтаксиса AT&T:

1. Регистры записываются с префиксом % (знак процента).
2. Каждая команда записывается с суффиксом, указывающим размер операндов команды (b — операнды размером 1 байт, w — 2 байта, l — 4 байта).
3. В командах пересылки данных приемник данных находится справа от источника. Например, код `movl %ebx, %eax` пересылает (копирует) в регистр eax значение, содержащееся в регистре ebx.
4. Константы записываются с префиксом $ (знак доллара). Если знак доллара не записан — процессор будет использовать значение, хранящееся в памяти по указанному адресу. Например, `movl $0xFFFFF, %eax` записывает в регистр eax значение FFFFF, а `movl 0xFFFFF, %eax` считывает 4 байта из памяти по адресу 0xFFFFF и записывает их в регистр eax.
5. Адрес текущего смещения в памяти — знак . (точка).
6. Доступ к памяти по адресу, расположенном в регистре, осуществляется с помощью записи N(%reg), где N — смещение в байтах в памяти относительно адреса, хранящегося в регистре reg. Например, команда `movb 0(%bx), %al` копирует один байта из памяти с адресом, хранящимся в регистре bx в регистр al.

Например, следующая функция реализована с использованием синтаксиса AT&T и предназначена для посимвольного вывода на экран строки, оканчивающейся нулевым символом, с помощью функции BIOS:

```
puts:
      movb 0(%bx), %al
      test %al, %al
      jz end_puts

      movb $0x0e, %ah
      int $0x10

      addw $1, %bx
      jmp puts

      end_puts:
      ret
```

Для вызова этой функции необходимо объявить и задать строку, поместить ее адрес в bx и вызывать с помощью инструкции call:

```
      movw $loading_str, %bx
      call puts

      …; Другие инструкции

loading_str:
      .asciz "Loading..."
```

Объявление данных и код вызова функции puts в синтаксисе Intel выглядит следующим образом:

```
      mov bx, loading_str ; Для GNU assembler: mov bx, offset loading_str
      call puts

      …; Другие инструкции

loading_str:
      db "Loading...", 0
```

В современных инструментах разработки используются разные синтаксисы. Например, в исходном коде программы на языке Си, компилируемой с помощью GNU C Compiler (gcc) допускаются ассемблерные вставки только в синтаксисе AT&T. Компилятор Microsoft C Complier (cl.exe), входящий в состав Visual Studio, поддерживает ассемблерные вставки только в синтаксисе Intel.

Трансляторы языка ассемблера поддерживают либо оба синтаксиса (GNU assembler, YASM), либо только синтаксис Intel (FASM, NASM).

Трансляторы языка ассемблера

Перевод программы на языке ассемблера в исполнимый машинный код (вычисление выражений, раскрытие макрокоманд, замена мнемоник инструкций машинными кодами и символьных адресов на абсолютные или относительные адреса) производится *ассемблером* — программой-транслятором.

Директивы, команды и макроманды трансляторов незначительно отличаются. Примеры приводятся в табл. 2.

Таблица 2

Отличия трансляторов с языка ассемблера

Директива / команда	FASM	GNU assembler (Intel syntax)	GNU assembler (AT&T syntax)	YASM (Intel syntax)	YASM (AT&T syntax)
Указание адреса загрузки кода	`org 0x7C00`	(аргументы командной строки)	(аргументы командной строки)	`[ORG 0x7c00]`	`.org 0x7c00`
Переход в 16-битный режим мнемоник	`use16`	`.code16`	`.code16`	`[BITS 16]`	`.code16`
Переход в 32-битный режим мнемоник	`use32`	`.code32`	`.code32`	`[BITS 32]`	`.code32`
Указание синтаксиса команд	(не поддерживается)	`.intel_syntax noprefix`	`.att_syntax`	(аргументы командной строки)	(аргументы командной строки)
Объявление байта данных	`db 0xff`	`.byte 0xff`	`.byte 0xff`	`db 0xff`	`.byte 0xff`
Объявление 2-байтового слова данных	`dw 0xffff`	`.word 0xffff`	`.word 0xffff`	`dw 0xffff`	`.word 0xffff`
Заполнение нулями до конца сектора	`times (512 - ($ - start) - 2) db 0`	`.zero (512 - ($ - _start) - 2)`	`.zero (512 - (. - _start) - 2)`	`times (512 - ($ - start) - 2) db 0`	`.zero (512 - (. - start) - 2)`

Загрузка LDT	`lgdt [gdt_info]`	`lgdt gdt_info`	`lgdt gdt_info`	`lgdt [gdt_info]`	`lgdt gdt_info`
Комментарий	`;`	`;`	`#`	`;`	`#`
Запись адреса строки в регистр	`mov bx, loading_str`	`mov bx, offset loading str`	`movw $str_loading, %bx`	`mov bx, loading_str`	`movw $loading_str, %bx`
Объявление строки	`loading_str: db "Loading…", 0`	`loading_str: .asciz "Loading..."`	`str_loading: .asciz "Loading..."`	`loading_str: db "Loading...", 0`	`loading_str: .asciz "Loading..."`
"Дальний" переход	`jmp 0x8:protected mode`	`jmp 0x8:protected mode`	`ljmp $0x8, $protected_mode`	`jmp 0x8:protected mode`	`ljmp $0x8, $protected_mode`
Обращение по адресу, хранимому в регистре	`mov al, [bx]`	`mov al, [bx]`	`movb 0(%bx), %al`	`mov al, [bx]`	`movb 0(%bx), %al`

Сборка кода загрузочного сектора, в зависимости от используемого транслятора, осуществляется следующими командами (предполагается, что исходный код расположен в файле bootsect.asm):

Таблица 3

Команды трансляции кода загрузчика в бинарный код

Транслятор	**Синтакис**	**Команды**
FASM	Intel	`fasm bootsect.asm`
GNU assembler	Intel	`as --32 -o bootsect.o bootsect.asm` `ld -Ttext 0x7c00 --oformat binary -m elf_i386 -o bootsect.bin bootsect.o`
	AT&T	
YASM	Intel	`yasm -f bin -o bootsect.bin bootsect.asm`
	AT&T	`yasm -p gas -f bin -o bootsect.tmp bootsect.asm` `dd bs=31744 skip=1 if=bootsect.tmp of=bootsect.bin`

Примечание: dd является программой, не поставляемой вместе с транслятором YASM. В случае отсутствия этой программы ее необходимо установить в систему отдельно, либо поместить бинарный файл программы dd.exe в каталог установки yasm. Обычно эта операция требуется в Windows.

Основные функции BIOS

Механизм BIOS предоставляет довольно большое количество системных функций, доступ к которым осуществляется посредством аппарата прерываний. Для вызова прерывания в определенные регистры процессора заносят необходимые данные, после чего выполняют команду `int`, параметром которой является номер прерывания. Номер функции прерывания заносится в регистр AH перед вызовом.

Для работы загрузчика требуются следующие функции BIOS:

1. `int 0x10` — видео сервис. Посредством вызова прерывания `0x10` можно вызвать функцию `0x00` — установка текстового режима терминала или функцию `0x0E` — печать символа на экран.

2. `int 0x13` — дисковый ввод/вывод. Функция `0x02` — считывание заданного количества секторов с диска в память. Перед ее вызовом необходимо выставить следующие значения регистров:

- DL — номер диска (носителя). Например, 0 — дисковод *A:*;
- DH — номер головки;
- старшие два бита CL и CH — номер цилиндра (дорожки);
- CL — младшие шесть бит — номер сектора;
- AL — количество секторов (в сумме не более одного цилиндра/дорожки, т.е. для магнитного диска 1,44Мб — не более 18 секторов за один вызов);
- ES:BX — адрес буфера, в который считываются данные.

В случае ошибки функция устанавливает флаг CF. Поскольку функция будет использоваться для чтения с дискеты, поэтому можно считать, что номер цилиндра — CH, а номер сектора — CL.

3. `int 0x16` — работа с клавиатурой. Функция `0x00` — ожидание нажатия и считывание нажатой пользователем клавиши.

Система защиты современных процессоров

Защита процессора — это совокупность приемов, с помощью которых процессор может контролировать выполнение кода:

- запрет выполнения определенных инструкций;
- запрет доступа к определенным областям памяти;
- запрет доступа к определенным внешним устройствам.

Во всех современных процессорах используются одни и те же принципы защиты. В большинстве процессоров используется концепция двух уровней привилегий (уровень Супервизора и уровень Пользователя). Как правило, если текущий код выполняется на уровне Супервизора, то ему разрешено делать все, что угодно. Если же это пользовательский код, то вступают в силу вышеуказанные ограничения.

Фирма Intel в процессорах архитектуры IA-32 (Intel Architecture 32-bit) используется не два, а четыре уровня привилегий, которые именуются кольцами и отсчитываются от нуля (наиболее защищенный) до трех (наименее защищенный). Причем 0 — наиболее привилегированный уровень, а 3 — наименее привилегированный. В модели "Супервизор-Пользователь" уровень 0 соответствует уровню Супервизора, а 3 — уровню Пользователя. Остальные уровни привилегий обычно не используются в программах.

Классический вариант архитектуры ОС: ядро системы работает на уровне Супервизора, а пользовательские задачи на уровне Пользователя, причем, пользовательские задачи не могут вмешаться в работу друг друга или ядра, а само ядро имеет полный доступ ко всем процессам в ОС.

Для осуществления вызова пользовательскими приложениями функции ядра процессоры предоставляют механизм переключения уровня привилегий. Супервизор может стать Пользователем, выполняя любой код, Пользователь может стать Супервизором, выполняя доверительный код.

Доверительный код назван таким образом, потому что Пользователь может выполнять в режиме Супервизора только тот код, который

Супервизор разрешит ему выполнять. Внешне это выглядит, как предоставление системных функций пользовательским приложениям. Например, вызов функции открытия файла `open()` в ОС UNIX означает, что при передаче управления на точку входа функции `open()` пользовательская программа получает привилегии Супервизора (благодаря чему становится возможным доступ к диску). Этот механизм передачи управления системе называется механизмом системных вызовов.

Основы архитектуры IA-32

Реальный режим работы процессора предназначен для совместимости с предыдущими моделями процессоров и механизмы защиты в нем отсутствуют. Защищенный режим — основной режим процессора, который должны использовать все современные ОС, и который позволяет использовать все возможности процессора (32-битную адресацию памяти, все инструкции, аппаратную многозадачность, защиту).

Задача защищенного режима может выполняться в режиме виртуального 8086, который очень похож на реальный режим. Таким образом, например, осуществляется эмуляция MS-DOS в системе Windows.

При включении компьютера процессор оказывается в реальном режиме и задача любой программы-загрузчика ОС — перевести процессор в защищенный режим.

Защищенный режим IA-32

Для поддержки функциональных возможностей процессора архитектура IA-32 предоставляет набор системных регистров и различных структур данных. Основная структура данных защищенного режима — дескриптор, который имеет размер восемь байт данных и используется для контроля сегментации, аппаратной многозадачности, прерываний.

Находясь в защищенном режиме, процессор всегда использует сегментацию — все доступное адресное пространство процессора разделяется на защищенные части, которые называются сегментами и в дальнейшем каждое обращение к памяти осуществляется через один из сегментов. В дескрипторе сегмента хранится информация о типе сегмента (кода или данных), его базе (адресе начала сегмента), лимите (размере сегмента), уровне привилегий сегмента.

Применение сегментации позволяет реализовать механизм защиты на уровне сегментов. Например, ядро ОС размещают в сегменте с уровнем привилегий ноль, а пользовательскую программу — в сегменте с уровнем привилегий три. В результате пользовательская программа не сможет вмешаться в работу ядра ОС.

Глобальная таблица дескрипторов

Размер сегментных регистров в защищенном режиме — 10 байт. Из них 8 недоступны для пользователя и называются теневой частью, или

кэшем дескриптора. Доступные 2 байта сегментных регистров составляют селектор:

- биты 16-3 — номер дескриптора сегмента в таблице;
- бит 2 — используемая таблица (0 — глобальная, 1 — локальная);
- биты 1-0 — запрашиваемый уровень привилегий (RPL).

При загрузке селектора в сегментный регистр в его теневую часть загружается дескриптор, на который указывает селектор, благодаря этому процессору не требуется запрашивать таблицу дескрипторов.

Для того чтобы перевести процессор в защищенный режим, используется *глобальная таблица дескрипторов*. Она содержит по одному восьмибайтному дескриптору для каждого сегмента. Дескриптор содержит 32-разрядный адрес начала сегмента (База), 20-битный размер сегмента (Лимит) и 12 бит описывающих тип сегмента. Полный формат дескриптора представлен на рис. 1.

<table>
<tr><td>Байт 0</td><td>Бит 7</td><td>Бит 6</td><td>Бит 5</td><td>Бит 4</td><td>Бит 3</td><td>Бит 2</td><td>Бит 1</td><td>Бит 0</td></tr>
<tr><td>Байт 1</td><td colspan="8">Биты 7-0 лимита сегмента</td></tr>
<tr><td>Байт 2</td><td colspan="8">Биты 15-8 лимита сегмента</td></tr>
<tr><td>Байт 3</td><td colspan="8">Биты 7-0 базы сегмента</td></tr>
<tr><td>Байт 4</td><td colspan="8">Биты 15-8 базы сегмента</td></tr>
<tr><td>Байт 5</td><td colspan="8">Биты 23- 16 базы сегмента</td></tr>
<tr><td>Байт 6</td><td>P - бит присутствия сегмента</td><td colspan="2">DPL - уровень привилегий дескриптора</td><td>S - системный 0 / обычный 1</td><td colspan="4">Тип сегмента</td></tr>
<tr><td>Байт 7</td><td>G – лимит в байтах/ 4</td><td>B – разрядность 16/32</td><td>0</td><td>AVL</td><td colspan="4">Биты 19-16 лимита сегмента</td></tr>
<tr><td>Байт 8</td><td colspan="8">Биты 31-24 базы сегмента</td></tr>
</table>

Рис. 1. Формат дескриптора

Минимальная таблица дескрипторов должна состоять из трех записей:

- нулевой дескриптор;
- дескриптор, описывающий область памяти, в которой размещен программный код;
- дескриптор, описывающий область памяти, в которой размещены данные.

Следующий пример демонстрирует описание синтаксисом Intel минимальной таблицы дескрипторов, необходимой для переключения процессора в защищенный режим. Каждый сегмент имеет базу 0 и лимит 4 Гб, что покрывает всю адресуемую процессором память.

```
gdt:
      ; Нулевой дескриптор
      db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

      ; Сегмент кода: base=0, size=4Gb, P=1, DPL=0, S=1(user),
      ; Type=1(code), Access=00A, G=1, B=32bit
      db 0xff, 0xff, 0x00, 0x00, 0x00, 0x9A, 0xCF, 0x00

      ; Сегмент данных: base=0, size=4Gb, P=1, DPL=0, S=1(user),
      ; Type=0(data), Access=0W0, G=1, B=32bit
      db 0xff, 0xff, 0x00, 0x00, 0x00, 0x92, 0xCF, 0x00

gdt_info: ; Данные о таблице GDT (размер, положение в памяти)
      dw gdt_info - gdt   ; Размер таблицы (2 байта)
      dw gdt, 0           ; 32-битный физический адрес таблицы.
```

Описание той же таблицы в синтаксисе AT&T, подходящем также и для транслятора GNU assembler в синтаксисе Intel:

```
gdt:
      .byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
      .byte 0xff, 0xff, 0x00, 0x00, 0x00, 0x9A, 0xCF, 0x00
      .byte 0xff, 0xff, 0x00, 0x00, 0x00, 0x92, 0xCF, 0x00

gdt_info:
      .word gdt_info - gdt
      .word gdt, 0
```

Переключение процессора в защищенный режим

Фактически, за вход в защищенный режим отвечает нулевой бит регистра `CR0`, который также называется битом PE (Protection Enable). Но перед включением защищенного режима шестибайтная структура, состоящая из 32-разрядного линейного адреса таблицы дескрипторов (GDT — Global Descriptor Table) и 16-битного лимита таблицы, должна быть загружена в регистр `GDTR`. Во время этого процесса прерывания должны быть отключены. Также необходимо включить адресную линию `A20`.

Следующая программа реализует последовательность действий:

- отключение прерываний;
- загрузка адреса и размера таблицы GDT;
- включение адресной линии `A20`;
- установка бита `PE` регистра `CR0` в 1;
- длинный переход ("прыжок") в 32-битный сегмент (на метку с адресом `protected_mode`).

```
      ; Отключение прерываний
      cli

      ; Загрузка размера и адреса таблицы дескрипторов
      lgdt [gdt_info] ; Для GNU assembler должно быть "lgdt gdt_info"

    ; Включение адресной линии A20
      in al, 0x92
      or al, 2
      out 0x92, al
```

```
      ; Установка бита PE регистра CR0 - процессор перейдет в защищенный
режим
      mov eax, cr0
      or al, 1
      mov cr0, eax

      jmp 0x8:protected_mode ; "Дальний" переход для загрузки корректной
информации в cs (архитектурные особенности не позволяют этого сделать
напрямую).

use32
protected_mode:
      ; Здесь идут первые инструкции в защищенном режиме
```

Тот же пример в синтаксисе AT&T:

```
      # Отключение прерываний
      cli

      # Загрузка размера и адреса таблицы дескрипторов
      lgdt gdt_info

    # Включение адресной линии A20
      inb $0x92, %al
      orb $2, %al
      outb %al, $0x92

      # Установка бита PE регистра CR0 - процессор перейдет в защищенный
режим
      movl %cr0, %eax
      orb $1, %al
      movl %eax, %cr0

      ljmp $0x8, $protected_mode # "Дальний" переход для загрузки
корректной информации в cs, архитектурные особенности не позволяют этого
сделать напрямую).

.code32
protected_mode:
      # Здесь идут первые инструкции в защищенном режиме
```

Передача управления от загрузчика ядру

Загрузчик, переведя процессор в защищенный режим, может на этом завершить свою работу и передать управление ядру. Ядро предварительно должно быть загружено в память по некоторому определенному загрузчиком адресу. Следующий пример содержит код, передающий управление ядру, загруженному по физическому адресу `0x10000`: (`0x1000:0x0000` при сегментной адресации реального режима):

```
use32
protected_mode:
      ; Загрузка селекторов сегментов для стека и данных в регистры
      mov ax, 0x10 ; Используется дескриптор с номером 2 в GDT
      mov es, ax
      mov ds, ax
      mov ss, ax

      ; Передача управления загруженному ядру
      call 0x10000      ; Адрес равен адресу загрузки в случае если ядро
скомпилировано в "плоский" код
```

Тот же код в синтаксисе AT&T:

```
.code32
protected_mode:
      movw $0x10, %ax
      movw %ax, %es
      movw %ax, %ds
      movw %ax, %ss
      call 0x10000
```

Работа с дисплеем

Все, что изображено на мониторе — и графика, и текст, одновременно присутствует в памяти, встроенной в видеоадаптер. Для того чтобы изображение появилось на мониторе, оно должно быть записано в память видеоадаптера. Для этого отводится специальная область памяти, начинающаяся с абсолютного адреса 0x000B8000 (для текстовых режимов) и заканчивающаяся на 0x000C7FFF. Все, что программы записывают в эту область памяти, немедленно пересылается в память видеоадаптера. В текстовых режимах для хранения каждого изображенного символа используются два байта: байт с ASCII-кодом символа и байт с его атрибутом, так что по адресу 0x000B8000 находится байт с кодом символа, находящимся в верхнем левом углу экрана; по адресу 0x000B8001 находится атрибут этого символа; по адресу 0x000B8002 находится код второго символа в верхней строке экрана и т.д. Таким образом, любой код, выполняющийся в защищенном режиме, может вывести текст на экран простой командой пересылки данных, не прибегая ни к каким специальным функциям.

Минимальная реализация ядра ОС на языке ассемблера

Следующий пример демонстрирует минимальную реализацию ядра операционной системы на языке ассемблера в синтаксисе Intel:

```
use32 ; 32-битный код
org 0x10000 ; Адрес, по которому будет загружен этот код загрузчиком ядра

      mov edi, 0xb8000    ; В регистр edi помещается адрес начала буфера
видеопамяти.
      mov esi, str_hello  ; В регистр esi помещается адрес начала строки
      call video_puts     ; Вызывается функция для видеовывода в буфер
памяти видеокарты

infinite_loop:
      ; Перевод процессора в бесконечный цикл
      hlt
      jmp infinite_loop

video_puts:
      ; Функция выводит в буфер видеопамяти (передается в edi) строку,
оканчивающуюся 0 (передается в esi)
      ; После завершения edi содержит адрес по которому можно продолжать
вывод следующих строк
      mov al, [esi]
      test al, al
```

```
	jz video_puts_end

	mov ah, 0x07 ; Цвет символа и фона. Возможные варианты: 0x00 is
black-on-black, 0x07 is lightgrey-on-black, 0x1F is white-on-blue
	mov [edi], al
	mov [edi+1], ah

	add edi, 2
	add esi, 1
	jmp video_puts

video_puts_end:
	ret

str_hello:
	db "Welcome to HelloWorldOS (asm edition)!", 0
```

Ядро данной ОС выводит на экран строку "Welcome to HelloWorldOS (asm edition)" и переходит в бесконечный цикл вызовов инструкции hlt процессора. Данный пример может быть скомпилирован в бинарный образ с помощью команды (предполагается, что код записан в файле kernel.asm):

```
fasm kernel.asm
```

Полученный образ kernel.bin можно разместить на втором гибком диске. На первом гибком диске можно разместить загрузчик, который:

1. Загружает ядро со второго гибкого диска по адресу 0x1000:0x0000. Загрузка производится прямым чтением секторов с помощью функции BIOS.
2. Переводит процессор в защищенный режим.
3. Передает управление загруженному ядру.

В этом случае проверить работоспособность загрузчика и ядра ОС можно на эмуляторе QEMU следующей командой:

```
qemu –fda bootsect.bin –fdb kernel.bin
```

Минимальная реализация ядра ОС на языке Си

Следующий пример демонстрирует реализацию минимального ядра ОС на языке Си:

```
// Эта инструкция обязательно должна быть первой, т.к. этот код
компилируется в бинарный,
// и загрузчик передает управление по адресу первой инструкции бинарного
образа ядра ОС.
__asm("jmp kmain");

#define VIDEO_BUF_PTR		(0xb8000)

void out_str(int color, const char* ptr, unsigned int strnum)
{
	unsigned char* video_buf = (unsigned char*) VIDEO_BUF_PTR;
	video_buf += 80*2 * strnum;

	while (*ptr)
	{
		video_buf[0] = (unsigned char) *ptr; // Символ (код)
```

```
            video_buf[1] = color; // Цвет символа и фона

            video_buf += 2;
            ptr++;
      }
}

const char* g_test = "This is test string.";

extern "C" int kmain()
{
      const char* hello = "Welcome to HelloWorldOS (gcc edition)!";

      // Вывод строки
      out_str(0x07, hello, 0);
      out_str(0x07, g_test, 1);

      // Бесконечный цикл
      while(1)
      {
            asm("hlt");
      }

      return 0;
}
```

Данный пример может быть скомпилирован с помощью компилятора gcc следующими командами (предполагается, что код записан в файле kernel.cpp):

```
g++ -ffreestanding -m32 -o kernel.o -c kernel.cpp
ld --oformat binary -Ttext 0x10000 -o kernel.bin --entry=kmain -m elf_i386 kernel.o
```

Примечание: в случае, если компиляция выполняется в 64-разрядной среде Linux, в системе должны быть установлены пакеты gcc-multilib, binutils, g++,g++-multilib. В Linux Debian и подобных дистрибутивах пакеты можно установить с помощью apt-get (запускается через sudo).

Проверить работоспособность ядра можно с помощью загрузчика, рассмотренного в предыдущем разделе. Следует обратить внимание, что размер ядра, получаемого в результате компиляции, может быть больше, чем 512 байт (один сектор). Для корректной работы ядра загрузчик должен загрузить все секторы, занимаемые ядром, с гибкого диска. Чтобы узнать количество необходимых для чтения секторов необходимо взять размер файла kernel.bin и разделить его на размер сектора (512 байт), с округлением в большую строну.

Проверка на эмуляторе QEMU выполняется той же командой:

```
qemu –fda bootsect.bin –fdb kernel.bin
```

Код на языке Си может быть собран с помощью компилятора Microsoft C Compiler следующими инструкциями (если не указаны переменные окружения, то сборку необходимо производить из Visual Studio Command Prompt):

```
cl.exe /GS- /c kernel.cpp
link.exe /OUT:kernel.bin /BASE:0x10000 /FIXED
/FILEALIGN:512 /MERGE:.rdata=.data /IGNORE:4254
/NODEFAULTLIB /ENTRY:kmain /SUBSYSTEM:NATIVE
kernel.obj
```

Для успешной компиляции ассемблерные вставки в коде программы на Си должны быть написаны с использованием синтаксиса Intel:

```
// Начало файла kernel.cpp
extern "C" int kmain();
__declspec(naked) void startup()
{
      __asm {
            call kmain;
      }
}

      // Бесконечный цикл работы ядра
      while(1)
      {
            __asm hlt;
      }
```

Выходным файлом, получаемым в результате компиляции с помощью Microsoft C Compiler, является PE-файл. Чтобы успешно загрузить такое ядро в память и передать управление на первую инструкцию кода (stub в начале файла, отсылающий к функции kmain), загрузчик должен обладать некоторой информацией о размещении бинарного кода и данных в этом файле.

Загрузка PE-ядра

PE-файл kernel.bin, получаемый при компиляции кода ядра с помощью компилятора Microsoft, содержит несколько служебных заголовков. Заголовки содержат следующие данные про каждую из секций:

- адрес, по которому должна быть загружена секция;
- размер секции;
- расположение секции в файле.

Для корректной работы ядра необходимо корректно загрузить в память из файла две секции: .text (секция код) и .data (секция данных).

Данные о секциях могут быть получены с помощью утилиты dumpbin:

```
dumpbin /headers kernel.bin
```

Пример полученного утилитой вывода:

```
Dump of file kernel.bin

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
```

```
             14C machine (x86)
               3 number of sections
...
SECTION HEADER #1
   .text name
     6F1 virtual size
    1000 virtual address (00011000 to 000116F0)
     800 size of raw data
     400 file pointer to raw data (00000400 to 00000BFF)
...

SECTION HEADER #2
   .data name
     8C0 virtual size
    2000 virtual address (00012000 to 000128BF)
     200 size of raw data
     C00 file pointer to raw data (00000C00 to 00000DFF)
```

Из данного вывода следует следующий алгоритм работы загрузчика:

1. Загрузить из файла (по смещению 0x400 от начала файла) 0x800 байт по адресу 0x00011000. Сектор cl=3, считать секторов al=4.

2. Загрузить из файла (по смещению 0xC00 от начала файла) 0x200 байт по адресу 0x00012000. Сектор cl=7, считать секторов al=1.

3. Передать управление по адресу 0x00011000.

В случае, если эмулятор запускается с помощью следующей командной строки:

```
qemu –fda bootsect.bin –fdb kernel.bin
```

файл kernel.bin будет являться образом гибкого диска. Тогда секция кода .text начинается с сектора 3 (т.к. размер одного сектора 0x200 или 512 байт) и занимает 4 сектора. Секция данных .data начинается с сектора 7 и занимает 1 сектор. Для корректной загрузки ядра эти константы следует заложить в загрузчик.

Следует отметить также, что при выполнении работы код ядра на Си будет увеличиваться, поэтому размер секций кода и данных также будет увеличиваться. Необходимо периодически сверять значения, записанные в загрузчике с реальными значениями kernel.bin, необходимыми для его корректной загрузки.

Прерывания и исключения процессора

Прерывания и исключения — это события, при которых процессор прекращает выполнение текущей программы и выполняет специальную процедуру — обработчик прерывания. При этом состояние процессора, как правило, сохраняется, что дает возможность продолжить выполнение прерванной программы. Исключения порождаются процессором, когда при выполнении кода происходит ошибка (например, деление на ноль).

Прерывания могут вызываться как внешними устройствами (аппаратные прерывания), так и самой программой (программные прерывания). При этом программные прерывания схожи с вызовом

процедуры. Аппаратные прерывания — способ, с помощью которых устройства могут повлиять на работу процессора, например, сигнализировать ему об окончании выполнения какого-либо действия. Прерывания подразделяются на маскируемые (которые процессор получает на вход INTR) и одно немаскируемое прерывание (NMI - NonMaskable Interrupt) (получаемое на вход NMI). Маскируемые прерывания могут быть отключены снятием флага IF регистра EFLAGS (для этого предназначена инструкция CLI). Для отключения немаскируемого прерывания необходимо перепрограммировать контроллер прерываний.

Процессору при возникновении прерывания известен только номер прерывания (вектор прерывания). Поскольку номер не содержит информации, где находится процедура-обработчик, эту информацию процессор должен получить из специальной таблицы прерываний.

В режиме реальных адресов таблица прерываний находится по абсолютному адресу 0:0x0…0:0x400 и представляет из себя 256 абсолютных (сегмент:смещение) четырехбайтных адресов процедур-обработчиков. Процессор получает из этой таблицы адрес обработчика, соответствующего номеру прерывания, и выполняет переход на функцию-обработчик.

В защищенном режиме таблица прерываний называется таблицей дескрипторов прерываний IDT (Interrupt Descriptors Table) и на ее местонахождение указывает регистр IDTR (interrupt descriptors table register). В таблице находятся не просто адреса обработчиков, а дескрипторы обработчиков. В качестве таких дескрипторов могут выступать дескрипторы:

- дескриптор шлюза прерывания;
- дескриптор шлюза ловушки.

При вызове обработчика прерывания через шлюз прерывания, дальнейшая их обработка запрещается до завершения обработчика. При вызове через шлюз ловушки этого не происходит. Соответственно, первый метод наиболее подходит для аппаратных прерываний, т.к. обработчик должен быть уверен, что он монопольно контролирует устройство во избежание аппаратных сбоев, а второй метод — для программных.

Формат дескриптора шлюза прерывания или ловушки представлены на рис. 2.

Байт 0	Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
Байт 1	Биты 7-0 смещения процедуры-обработчика							
Байт 2	Биты 15-8 смещения процедуры-обработчика							
Байт 3	Биты 7-0 селектора сегмента процедуры-обработчика							
Байт 4	Биты 15-8 селектора сегмента процедуры-обработчика							
Байт 5	0	0	0	0	0	0	0	0
Байт 6	P - бит присутствия сегмента	DPL - уровень привилегий дескриптора		0	Тип шлюза			
Байт 7	Биты 23-16 смещения процедуры-обработчика							
Байт 8	Биты 31-24 смещения процедуры-обработчика							

Рис. 2 Формат дескриптора шлюза прерывания (шлюза ловушки)

Поле "тип шлюза" принимает одно из следующих значений:

2. 0110 - 16-битный шлюз прерывания;
3. 0111 - 16-битный шлюз ловушки;
4. 1110 - 32-битный шлюз прерывания;
5. 1111 - 32-битный шлюз ловушки.

При вызове обработчика прерывания процессор сохраняет в стеке значения регистров EFLAGS, CS и EIP. Возврат из обработчика осуществляется командой IRETD, которая восстанавливает значения, сохраненные в стеке.

Исключения в процессорах IA-32 бывают трех видов:

- fault (сбой) — CS:EIP указывает на следующую после "пойманной" инструкцию. После завершения обработчика, программа может продолжать выполняться (как и при программных прерываниях);
- trap (ловушка) — CS:EIP указывает на ту инструкцию, из-за которой произошло исключение. После исправления ситуации возможно дальнейшее выполнение программы;
- abort (останов) — надежное восстановление выполнения невозможно, CS:EIP или EFLAGS могут иметь ошибочные значения.

Тип исключения зависит от того, когда процессор заметил ошибку, приведшую к исключению, и, следовательно, — куда указывают сохраненные в стеке CS:EIP, что в свою очередь определяет, возможно ли продолжение программы после завершения прерывания.

Для процессора Intel исключения перечислены в табл. 4.

Таблица 4

Исключения процессора Intel Pentium 4

Номер	Описание	Тип
0	Деление на нуль (Divide Error Exception или #DE).	fault
1	Отладочное прерывание (Debug Exception или #DB).	trap fault
2	NMI - немаскируемое прерывание.	
3	Точка останова (Breakpoint Exception или #BP).	trap
4	Переполнение (Overflow Exception или #OF). Примечание: вызывается если при выполнении инструкции INTO (Interrupt on overflow) флаг переполнения OF установлен.	trap
5	Выход за допустимые границы при BOUND (Bound Range Exceeded Exception или #BR). Примечание: вызывается если операнд инструкции BOUND выходит за границы массива.	fault
6	Неправильная инструкция (Invalid Opcode Exception или #UD). Примечание: вызывается при попытке выполнить несуществующую инструкцию или инструкцию с недопустимыми операндами.	fault
7	Математический сопроцессор не доступен (No Math или #NM). Примечание: вызывается при попытке выполнить инструкцию FPU, если его использование запрещено (проверяются несколько флагов CR0).	fault
8	Двойная ошибка (Double Fault Exception или #DF). Примечание: вызывается, если при вызове обработчика для исключения случилось еще одно исключение.	abort
9	Зарезервировано. Примечание: на 386 это было Coprocessor Segment Overrun.	
10(0xA)	Ошибочный TSS (Invalid TSS или #TS).	
11(0xB)	Несуществующий сегмент (Segment Not Present или #NP). Примечание: вызывается при обращении к сегменту (или дескриптору какого-либо шлюза), бит P которого установлен в 0.	fault
12(0xC)	Ошибка стека (Stack Fault Exception или #SS). Примечание: вызывается при превышении лимита сегмента стека или	fault

	загрузке несуществующего (P=0) дескриптора в SS.	
13(0xD)	Общее исключение защиты (General Protection Exception или #GP). Примечание: основное исключение защищенного режима.	fault
14(0xE)	Ошибка страничной адресации (Page Fault Exception или #PF). Примечание: в регистре CR2 находится адрес, обращение к которому вызвало ошибку .	fault
15(0xF)	Зарезервировано.	
16(0x10)	Ошибка сопроцессора (FPU Error или #MF).	fault
17(0x11)	Ошибка выравнивания (Alignment Check Exception или #AC). Примечание: вызывается при невыровненном обращении к памяти непривилегированным (CPL=3) кодом, если установлены флаги AC в EFLAGS и AM в CR0	fault
18(0x12)	Машинно-зависимая ошибка (Machine Check Exception или #MC).	abort
19(0x13)	Ошибка SSE/SSE2 (SIMD Floating Point Exception или #XF).	fault
20 - 31 (0x14 - 0x1F)	Зарезервированы.	

Процедура-обработчик прерываний

Следующий фрагмент программы (компилятор gcc), устанавливает пустые обработчики прерываний и позволяет запрещать и разрешать обработку прерываний:

```
#define IDT_TYPE_INTR   (0x0E)
#define IDT_TYPE_TRAP   (0x0F)

// Селектор секции кода, установленный загрузчиком ОС
#define GDT_CS          (0x8)

// Структура описывает данные об обработчике прерывания
struct idt_entry
{
    unsigned short base_lo;     // Младшие биты адреса обработчика
    unsigned short segm_sel;    // Селектор сегмента кода
    unsigned char always0;      // Этот байт всегда 0
    unsigned char flags;        // Флаги тип. Флаги: P, DPL, Типы - это
константы - IDT_TYPE...
    unsigned short base_hi;     // Старшие биты адреса обработчика
} __attribute__((packed)); // Выравнивание запрещено

// Структура, адрес которой передается как аргумент команды lidt
struct idt_ptr
{
```

```
    unsigned short limit;
    unsigned int base;
} __attribute__((packed)); // Выравнивание запрещено


struct idt_entry g_idt[256]; // Реальная таблица IDT
struct idt_ptr g_idtp;       // Описатель таблицы для команды lidt
// Пустой обработчик прерываний. Другие обработчики могут быть реализованы
по этому шаблону
void default_intr_handler()
{
      asm("pusha");
      // ... (реализация обработки)
      asm("popa; leave; iret");
}

typedef void (*intr_handler)();
void intr_reg_handler(int num, unsigned short segm_sel, unsigned short
flags, intr_handler hndlr)
{
      unsigned int hndlr_addr = (unsigned int) hndlr;

      g_idt[num].base_lo = (unsigned short) (hndlr_addr & 0xFFFF);
      g_idt[num].segm_sel = segm_sel;
      g_idt[num].always0 = 0;
      g_idt[num].flags = flags;
      g_idt[num].base_hi = (unsigned short) (hndlr_addr >> 16);
}

// Функция инициализации системы прерываний: заполнение массива с адресами
обработчиков
void intr_init()
{
    int i;
    int idt_count = sizeof(g_idt) / sizeof(g_idt[0]);

    for(i = 0; i < idt_count; i++)
        intr_reg_handler(i, GDT_CS, 0x80 | IDT_TYPE_INTR,
            default_intr_handler); // segm_sel=0x8, P=1, DPL=0, Type=Intr
}

void intr_start()
{
      int idt_count = sizeof(g_idt) / sizeof(g_idt[0]);

      g_idtp.base = (unsigned int) (&g_idt[0]);
      g_idtp.limit = (sizeof (struct idt_entry) * idt_count) - 1;

      asm("lidt %0" : : "m" (g_idtp) );
}

void intr_enable()
{
      asm("sti");
}

void intr_disable()
{
      asm("cli");
}
```

Для компилятора Microsoft C Compiler описание структур и реализация функций должны быть описаны иначе:

```
// Структура описывает данные об обработчике прерывания
#pragma pack(push, 1) // Выравнивание членов структуры запрещено
struct idt_entry
{
    unsigned short base_lo;    // Младшие биты адреса обработчика
    unsigned short segm_sel;   // Селектор сегмента кода
    unsigned char always0;     // Этот байт всегда 0
    unsigned char flags;       // Флаги тип. Флаги: P, DPL, Типы - это
константы - IDT_TYPE...
    unsigned short base_hi;    // Старшие биты адреса обработчика
};

// Структура, адрес которой передается как аргумент команды lidt
struct idt_ptr
{
    unsigned short limit;
    unsigned int base;
};
#pragma pack(pop)

__declspec(naked) void default_intr_handler()
{
      __asm {
            pusha
      }

      // ... (реализация обработки)

      __asm {
            popa
            iretd
      }
}


void intr_start()
{
      int idt_count = sizeof(g_idt) / sizeof(g_idt[0]);

      g_idtp.base = (unsigned int) (&g_idt[0]);
      g_idtp.limit = (sizeof (struct idt_entry) * idt_count) - 1;

      __asm {
            lidt g_idtp
      }

      //__lidt(&g_idtp);
}

void intr_enable()
{
      __asm sti;
}

void intr_disable()
{
      __asm cli;
}
```

Для инициализации подсистемы прерываний необходимо вызывать функцию `intr_init`. После этого необходимые обработчики (таймер, клавиатура, диск и т.д.) могут быть зарегистрированы с помощью функции `intr_reg_handler`. Регистрация таблицы дескрипторов прерываний осуществляется с помощью функции `intr_start`. Включение прерываний осуществляется функцией `intr_enable`. Примером пустого обработчика прерываний является функция `default_intr_handler`.

Перед выполнением задачи обработчик прерывания должен сохранить регистры, а по завершении обработки — послать байт 0x20 в порт 0x20 (сигнал "конец прерывания" контроллеру прерываний), если это прерывание было вызвано контроллером прерываний, после чего восстановить регистры (см. примеры в след. пункте).

Обработка прерываний клавиатуры

При нажатии пользователем клавиши клавиатуры контроллер прерываний вызывает прерывание 0x09 процессора. Номер прерывания может быть перепрограммирован путем отправки команд контроллеру прерываний. Перечень прерываний, которые ОС обрабатывает, сообщается контроллеру в виде битовой маски. Взаимодействие с оборудованием, в том числе с контроллером прерываний, производится с помощью чтения и записи в порты ввода-вывода. Для этого предназначены следующие служебные функции:

```
static inline unsigned char inb (unsigned short port) // Чтение из порта
{
  unsigned char data;
  asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
  return data;
}

static inline void outb (unsigned short port, unsigned char data) // Запись
{
  asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
}
```

Реализация в синтаксисе Intel для компилятора Microsoft:

```
__inline unsigned char inb (unsigned short port)
{
  unsigned char data;
  __asm {
      push dx
      mov dx, port
      in al, dx
      mov data, al
      pop dx
  }
  return data;
}
```

```
__inline void outb (unsigned short port, unsigned char data)
{
  __asm {
      push dx
      mov dx, port
      mov al, data
    out dx, al
      pop dx
  }
}
```

Следующий фрагмент программы регистрирует обработчик прерывания клавиатуры и разрешает контроллеру прерываний его вызывать в случае нажатия пользователем клавиши клавиатуры:

```
#define PIC1_PORT (0x20)
void keyb_init()
{
// Регистрация обработчика прерывания
      intr_reg_handler(0x09, GDT_CS, 0x80 | IDT_TYPE_INTR, keyb_handler);
// segm_sel=0x8, P=1, DPL=0, Type=Intr

      // Разрешение только прерываний клавиатуры от контроллера 8259
      outb(PIC1_PORT + 1, 0xFF ^ 0x02); // 0xFF - все прерывания, 0x02 -
бит IRQ1 (клавиатура).
// Разрешены будут только прерывания, чьи биты установлены в 0
}
```

Обработчик прерываний может быть реализован следующим образом:

```
void keyb_handler()
{
      asm("pusha");

      // Обработка поступивших данных
      keyb_process_keys();

      // Отправка контроллеру 8259 нотификации о том, что прерывание
обработано
      outb(PIC1_PORT, 0x20);
      asm("popa; leave; iret");
}
```

Для компилятора Microsoft:

```
__declspec(naked) void keyb_handler()
{
      __asm pusha;

      // Обработка поступивших данных
      keyb_process_keys();

      // Отправка контроллеру 8259 нотификации о том, что прерывание
обработано. Если не отправлять нотификацию, то контроллер не будет посылать
новых сигналов о прерываниях до тех пор, пока ему не сообщать что
прерывание обработано.
      outb(PIC1_PORT, 0x20);

      __asm {
            popa
            iretd
      }
}
```

Функция keyb_process_keys считывает поступивший от пользователя символ. Следующий пример демонстрирует считывание скан-кода клавиши для PS/2 клавиатуры:

```
void keyb_process_keys()
{
	// Проверка что буфер PS/2 клавиатуры не пуст (младший бит
присутствует)
	if (inb(0x64) & 0x01)
	{
		unsigned char scan_code;
		unsigned char state;

		scan_code = inb(0x60); // Считывание символа с PS/2 клавиатуры

	if (scan_code < 128)  // Скан-коды выше 128 - это отпускание клавиши
		on_key(scan_code);
	}
}
```

Скан-код клавиши может быть переведен в символ путем применения таблиц, подходящих для используемого оборудования. Например, для многих клавиатур отрицательные скан-коды соответствуют отпусканию клавиши, а положительные — нажатию. Скан-код 2 обычно соответствует кнопке с цифрой '1', скан-код 14 — backspace, 15 — tab, 28 — enter. Таблицы трансляции скан-кодов клавиш в коды символов приводятся и публикуются производителями оборудования.

Управление курсором ввода

BIOS поддерживает взаимодействие с запущенной операционной системой через порты ввода-вывода. В данной лабораторной работе требуется реализовать операционную систему с командный интерфейсом. Для пользователя, взаимодействующего с ОС через командный интерфейс, ожидаемым поведением на нажатие клавиши является печать на экран символа и перемещение курсора на следующую позицию. Для печати символа можно применять прямой вывод в память видеоадаптера. Для изменения позиции курсора можно использовать взаимодействие с BIOS, например, с помощью следующей функции:

```
// Базовый порт управления курсором текстового экрана. Подходит для
большинства, но может отличаться в других BIOS и в общем случае адрес
должен быть прочитан из BIOS data area.
#define CURSOR_PORT     (0x3D4)
#define VIDEO_WIDTH      (80) // Ширина текстового экрана

// Функция переводит курсор на строку strnum (0 – самая верхняя) в позицию
pos на этой строке (0 – самое левое положение).
void cursor_moveto(unsigned int strnum, unsigned int pos)
{
    unsigned short new_pos = (strnum * VIDEO_WIDTH) + pos;
    outb(CURSOR_PORT, 0x0F);
    outb(CURSOR_PORT + 1, (unsigned char)(new_pos & 0xFF));
```

```
    outb(CURSOR_PORT, 0x0E);
    outb(CURSOR_PORT + 1, (unsigned char)( (new_pos >> 8) & 0xFF));
}
```

Порядок выполнения работы

1. Получить у преподавателя номер задания (*Приложение*).

2. Скомпилировать и запустить на эмуляторе пример загрузочного сектора start.asm, описанного в Теоретических сведениях, указанным в вашем варианте транслятором.

3. Пользуясь примерами из Теоретических сведений, разработать простой загрузчик для загрузки минимального ядра. Скомпилировать загрузчик и минимальное ядро указанными в вашем варианте задания транслятором и компилятором языка Си. Проверить работоспособность загрузчика и ядра на эмуляторе.

4. Разработать указанные в вашем варианте задания функции загрузчика.

5. Расширить реализацию минимального ядра, добавив в него функции, перечисленные в задании.

6. Предложить не менее 15 тестов для проверки работоспособности ОС (команд или иных входных данных, поступающих от пользователя). Протестировать вашу реализацию на предложенных тестах.

7. Ответить на контрольные вопросы.

Содержание отчета

1. Цели работы.

2. Задачи работы.

3. Описание алгоритма работы загрузчика ОС, процесса загрузки ОС. Описание метода решения задания к загрузчику. Блок-схема алгоритма загрузчика.

4. Описание метода решения задания к ядру ОС. Описание реализации: таблица, содержащая имена реализованных вами функций, имена и типы данных их аргументов, описание их назначения. В таблице приведите описание только тех функций, которые связаны с реализацией вашей задачи, указанной в варианте задания.

5. Тексты программ ядра и загрузчика из исходных текстов. Текст скрипта, вызывающего транслятор ассемблера, компилятор и линковщик, для сборки проекта. Тексты должны быть снабжены комментариями.

6. Входные данные и результаты работы. Должно быть проведено не менее 15 различных тестов для основных команд ОС.

7. Выводы по работе.