Урок 2

Ядро операционной системы

Варианты архитектуры ядра. Ядро ОС MS DOS и Linux. Функции и задачи ядра, его взаимодействие с процессами.

Введение

Операционные системы использовались и используются не всегда. Например, без использования операционной системы функционирует прошивка микроконтроллера, которая представляет собой одну программу. В настоящее время есть и другие программы, которые могут выполняться без операционной системы:

- программа POST в составе ПЗУ;
- код загрузочного сектора, программы MBR и EBR в начале загрузочного диска (в том числе код, выдающий сообщение: Disk boot failure, insert system disk and press any key и его вариации, например от ndd);
- загрузчики операционной системы: ntldr, LILO, GRUB;
- программы для тестирования неисправностей (например, Memtest 86+);
- прошивка интерпретаторов BASIC на заре появления и распространения микрокомпьютеров.

При этом такие программы могут предоставлять определённые возможности операционных систем.

Потребность в операционной системе появляется, когда предполагается использование нескольких программ, а также возможность для пользователя управлять ресурсами компьютера, то есть требуется оболочка. Однако такую оболочку представляет собой прикладное программное обеспечение, а часть операционной системы, которая управляет непосредственно ресурсами, предоставляя ресурсы и интерфейсы для прикладных программ, называется ядром операционной системы. Одного ядра для работы недостаточно, но без ядра операционная система превращается в прикладную программу, исполняющуюся в монопольном режиме.

Даже такие простые по современным меркам операционные системы, как CP/M и MS DOS, обладали ядром.

Архитектура ОС на примере MS DOS

Если рассмотреть содержание дискеты с MS DOS, в простейшем варианте там будут следующие файлы:

- io.sys;
- msdos.sys;
- config.sys;
- autoexec.bat;
- command.com.

Особого загрузчика в MS DOS не было, потому файлы io.sys и msdos.sys должны были располагаться в начальной области диска. Простым копированием этих файлов сделать дискету загрузочной не получилось бы – для этого использовалась утилита sys.com

Предположим, что дискета вставлена в дисковод, компьютер включен. Система ещё не начала читать загрузочный сектор дискеты, а уже выполнена программа POST из состава BIOS (basic input output system), определена таблица векторов прерываний, уже реализованы базовые возможности управления устройствами через прерывания BIOS. Дальше можно загрузить единственную программу, которая сможет работать даже без ОС, используя прерывания BIOS или работая с оборудованием напрямую.

io.sys формально представляет собой расширение функций BIOS. Он перекрывает и дополняет функции, связанные с вводом и выводом. На самом деле это более сложный и важный компонент операционной системы. После замещения функций BIOS (по существу, io.sys содержит

низкоуровневые драйверы операционной системы ms dos) io.sys читает текстовый файл config.sys. Этот INI-подобный файл содержит некоторые настройки системы (например, разрешение или запрет прерывания программ по CTRL-C, определение количества открытых файлов) и список загружаемых драйверов (управления памятью, русификаторы и т.д.). io.sys загружает драйверы в память, после чего передает управление msdos.sys.

msdos.sys – ядро операционной системы, которое предоставляет интерфейс API для прикладных программ. msdos.sys определяет прерывания, которые обрабатываются MS DOS. В частности, это прерывания 0x20 и 0x21.

Следующие примеры демонстрируют обращения DOS-программ к находящемуся в памяти msdos.sys через API.

Пример COM программы для MS-DOS:

```
.386
model tiny                                  /Указание модели памяти
Code segment use16                 /Начало описания сегмента кода
ASSUME cs:Code, ds:Code     /Ассоциация регистров с сегментом
    org    100h                             /Генерация смещения на 256 байт
start:                                          /Метка начала программы
    push    cs                               /Запись регистра CS в стек
    pop    ds                                 /Загрузка регистра DS значением из
стека
    mov    dx, offset mess             /Помещение в DX смещения строки mess
    mov    ah, 09h                           /Запись в AH номера функции вывода
строки
    int    21h                               /Вызов сервиса MS-DOS
    int    20h                                     /Завершение COM программы в
MS-DOS
mess    db    'Hello world!','$'       /Объявление строки
Code ends                                  /Завершение описания строки
end start
```

Пример EXE программы для MS-DOS:

```
.386
model small                                  /Указание модели памяти
Stack SEGMENT STACK use16  /Объявление сегмента стека
    ASSUME ss:Stack                         /Ассоциация регистра SS с сегментом
стека
    DB    100h dup(?)                      /Резервирование 256 байт под стек
Stack ENDS                                  /Завершение описания сегмента стека
Data SEGMENT use16                 /Объявление сегмента данных
    ASSUME ds:Data                       /Ассоциирование регистра DS с сегментом
данных
mess    db    'Hello world!','$'          /Объявление строки
Data ENDS                                        /Завершение описания сегмента
данных
Code SEGMENT use16                 /Объявление сегмента кода
    ASSUME cs:Code                       /Ассоциирование регистра CS с сегментом
кода
start:                                           /Метка начала программы
    mov    ax, seg mess                 /Загрузка в AX адреса сегмента строки
mess
    mov    ds, ax                            /Запись в DS значения AX
    mov    dx, offset mess                /Запись в DX смещения строки mess
```

```
     mov     ah, 09h                                    /Запись в AH номера функции
вывода строки
    int    21h                                        /Вызов сервиса MS-DOS
     mov     ax, 4c00h                                /Запись в AX функции завершения
программы
     int     21h                                       /Завершение EXE программы в
MS-DOS
Code ENDS                                            /Завершение описания сегмента
данных
end start
```

После того, как msdos.sys загрузился и переопределил необходимые прерывания в таблице прерываний, он остаётся в памяти, но управление вновь возвращается в io.sys

Далее io.sys загружает и передаёт управление файлу command.com (или иной оболочке, если она указана в параметре shell файла config.sys).

При таком случае запуска (а не запуска копии или запуска в режиме «Выход в DOS», доступного во многих сложных программах под DOS) command.com выполняет файл autoexec.bat, а затем переходит в режим командной строки. Система загружена.

Остальные части операционной системы – это драйверы и утилиты.

Драйверы – это программы sys и exe, загружаемые, как правило (но не обязательно), из config.sys, которые оставались в памяти постоянно (резидентно) и предоставляли интерфейс (изменённый интерфейс) для доступа к оборудованию или иные программные удобства. Как правило, драйверы перехватывают одно из прерываний для предоставления нужных функций.

Утилиты – программы, обычно вызываемые как команды из командной строки. Они также могут обладать интерфейсом.

Также в MS DOS присутствовали дополнительные программы. Вместе с первыми версиями MS DOS поставлялись версии basica или gwbasic – популярный интерпретатор языка Бейсик.

Разумеется, для полноценной работы необходимы драйверы, утилиты и прикладные программы.

Драйверы MS DOS

Драйверы в MS DOS загружались в формате sys или exe, как правило, на основе его упоминания в файле config.sys. Формат sys похож на com, но содержит дополнительную информацию о таблице размещения драйверов. Более того, драйвер мог быть загружен в autoexec.bat или вообще из командной строки (перехватив прерывание и при завершении работы, сообщив DOS, что не следует выгружать программу из памяти, то есть как резидентная программа). Никаких механизмов защиты не предполагалось – по существу, операционная система и прикладные программы работали в одном пространстве, поэтому ошибка в программе приводила к зависанию всей системы. DOS – довольно простая система, поэтому программы часто использовали непосредственно работу с оборудованием или применением специальных программ-расширителей, которые сами можно воспринимать как отдельные операционные системы. Кроме того, хотя Windows 3.11 часто обозначается как оболочка, она может загружаться и обратно возвращать управление в DOS – фактически это отдельная операционная система. Драйверы в формате sys могут быть, как мы потом увидим также и в linux, символьными и блочными.

В MS DOS не различалось пространство ядра и пространство пользователя. Любая пользовательская программа могла оставаться в памяти как драйвер (чем пользовались компьютерные вирусы) или использовать собственные драйверы.

Часто используемые в MS DOS драйверы (названия могут меняться):

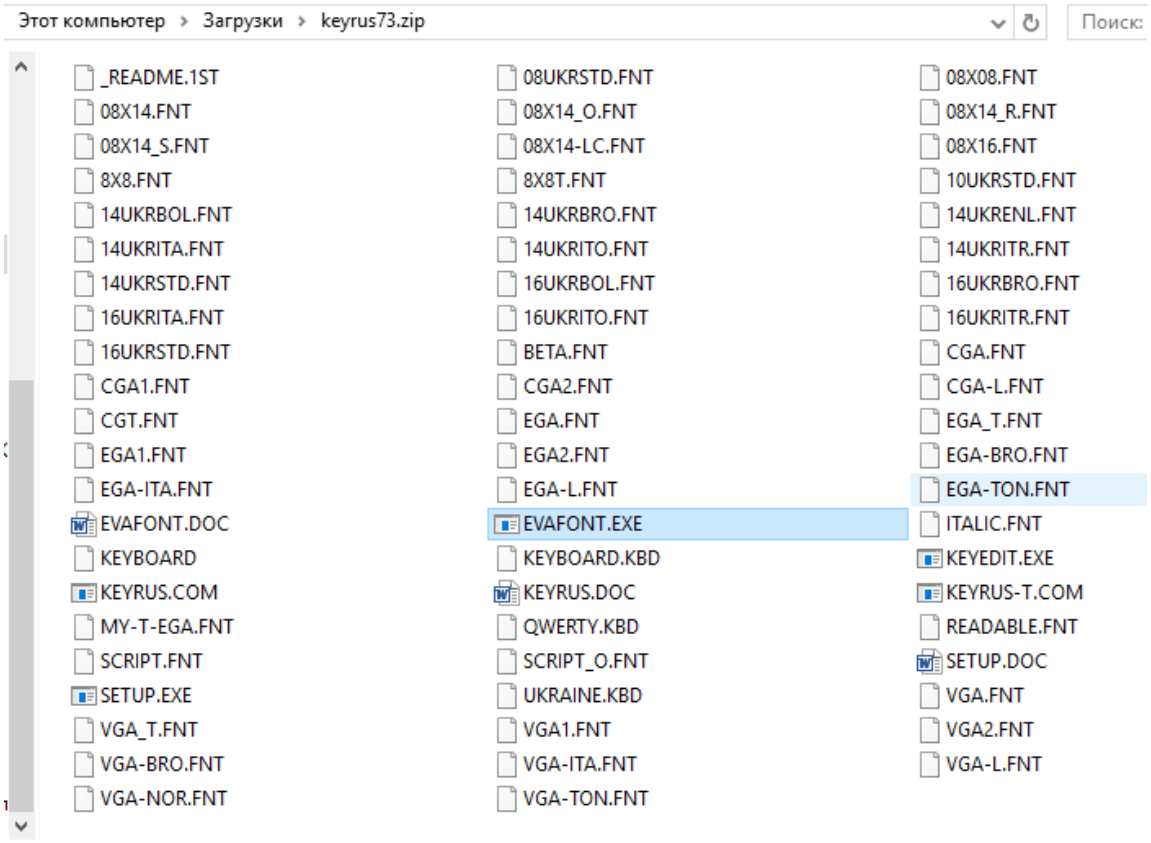
- mouse.sys/mouse.com – драйвер мыши. Содержал обработчик прерывания от мыши, отрисовывал курсор (в текстовом виде инверсией цветов позиции, в графическом виде в виде стрелки), отвечал за функции включения/выключения курсора, получения его координат и факта срабатывания клика мыши. По умолчанию поддержки мыши в MS DOS не было. Для работы с мышью требовалось, чтобы прикладные программы могли работать с драйвером мыши (включать курсор, реагировать на клик мыши);
- cdrom.sys/cdromdrv.sys/cddriver.sys/mscdex.exe – драйвер CD ROM. Делал доступным привод CD ROM, который получал свободную букву устройства (например, E:, сам диапазон доступных букв мог быть изменён в config.sys);
- ramdrive.sys/ramdrive.exe/ramdisk.exe – организовал использование части оперативной памяти как жёсткого диска. При выключении системы все данные, «записанные» на такой диск, разумеется, пропадали. Драйвер применялся для возможности запуска ОС с дискет, когда сжатые на дискете файлы распаковывались в RAM-диск и запускались оттуда;
- KeyRus.exe/kb.com – драйвер-русификатор KeyRus, написанный студентом Дмитрием Гуртяком в 1989 году. Популярная на постсоветском пространстве программа заменяла точечные шрифты на русифицированные, а также перехватывала прерывание от клавиатуры для отслеживания переключения раскладки (обычно правый Ctrl переключал РУС/ЛАТ, а правый Alt позволял набирать псевдографику, что могло быть полезно для ASCII-артов).

Шрифты в MS DOS

Рассмотрим подробнее момент, связанный с русификацией. Как написано выше, русификатор выполнял две задачи: загружал шрифт и обрабатывал прерывания от клавиатуры. Обработка прерываний от внешнего устройства – обычная ситуация для драйверов устройств.

Каким образом хранились шрифты в MS DOS? Изначально в микрокомпьютерах применялись (и сейчас применяются для определённых задач) точечные шрифты. Таблицы точечных шрифтов, хранимые в памяти, выглядят одинаково, механизм формирования нового символа для шрифта одинаков и для ZX-Spectrum (матрица 8 на 8), и для IBM PC (как правило, матрица 8 на 14, но могут быть и другие размеры).

Скачав на мемориальной странице Дмитрия Гуртяка архив с keyrus, мы увидим большое число файлов FNT.



Это связано с тем, что раскладки могут отличаться как кодировкой и поддерживаемым языком, так и разрешением экрана.

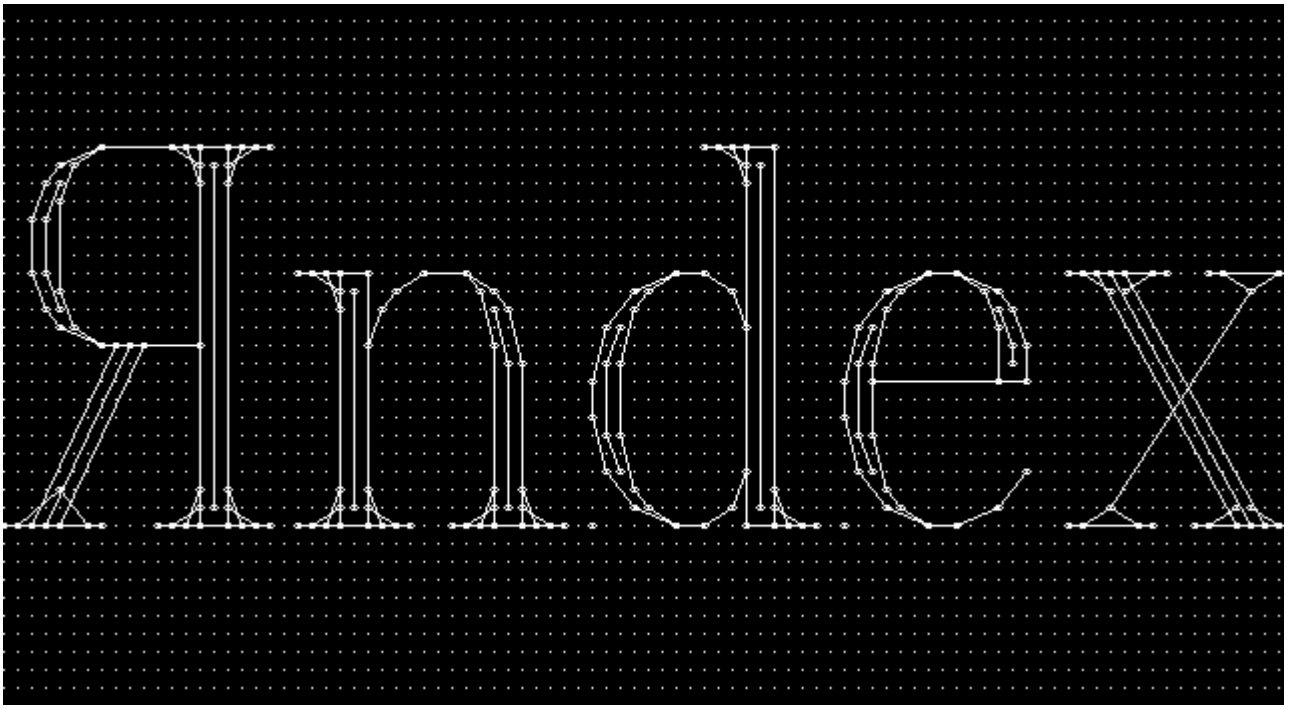
В DOS были доступны несколько режимов экрана, поддерживаемые драйвером видеоадаптера. Текстовый режим, как правило, содержал 25 строк по 80 столбцов, поддерживалось 16 цветов. В таком режиме можно было только выводить символы. Таблица символов содержала 256 символов в формате матриц 8 на 14 (или 8 на 16).

При печати строк с переводом строки в 24 и 25 строки все текстовые строки сдвигались вверх, содержимое первой и второй строки при этом не сохранялось.

Были и другие режимы: например, 50 строк на 80 столбцов. Графические режимы, как правило, работали в режиме пиксельного вывода (320x240, 640x480, 800x600), имитировали EGA- и VGA-адаптеры, обладали разной цветностью: монохромные, 4-цветные CGA, 16-цветные EGA и т.д. Режим можно было переключить запросом соответствующего прерывания. Несмотря на работу с точками, действовали те же самые функции печати, что и в текстовом режиме. Однако теперь в памяти не сохранялись коды символов, а сами символы выводились точечно. Используя посимвольную печать или изменив положение текстового курсора, возможно было напечатать символ там, где до этого символ уже был. При печати в 24 и 25 строке текст также сдвигался, при этом сдвигалась и графика с потерей верхних позиций не только текста, но и рисунка. Графический режим позволял работать с собственными методами печати – можно было не обращаться к операционной системе, а самостоятельно поточечно отрисовать символы: например, брать их из того же файла fnt или использовать нестандартные. Таким образом, технически можно было использовать даже Unicode, если подготовить соответствующий точечный шрифт.

Альтернативой было использование векторных шрифтов. Например, в Turbo Pascal использовались собственные драйверы для доступа к графическим режимам (в том числе полноцветным), такие, как egavga.bgi (640x200,640x350,640x480 16-цветные режимы), vga256.bgi (320x200 256-цветный режим),

ibm8514.bgi (640x480 и 1024x768 256-цветные режимы). Кроме того, была возможность использовать векторные шрифты.



Пример векторного шрифта в формате .chr от Borland

Источник: http://kokovkin.narod.ru/rus/portfolio.htm

В поставке Turbo Pascal имелись шрифты с засечками (Serif, типа Times New Roman), без засечек (Sans Serif, типа Arial и Calibri), моноширинные шрифты (типа Courier, используемые для отображения кода или имитации вывода печатной машинки – кстати, точечные шрифты для текстового режима по определению моноширинные), экзотические шрифты вроде рукописного (scri.chr) или готического (goth) написания.

Векторные шрифты из Turbo Pascal:

Номер	Название	Файл	Краткое описание
1	TriplexFont	trip.chr	Трехобводный, прямой, с засечками
2	SmallFont	small.chr	Однообводный, прямой, моноширинный
3	SansSerifFont	sans.chr	Двухобводный, прямой, моноширинный
4	GothicFont	goth.chr	Готический
5	Script.chr	scri.chr	«Рукописный» шрифт, курсив
6	SimpleFont	simp.chr	Одноштриховый шрифт типа Courier Двухобводный, прямой, пропорциональный
7	TSGRFont	tscr.chr	Красивый наклонный шрифт типа Times Italic Трехобводный, курсив, с засечками
8	LCOMFont	lcom.chr	Шрифт типа Times New Roman
9	EuroFont	euro.chr	Шрифт типа Courier увеличенного размера
10	BoldFont	bold.chr	Крупный двухштриховый шрифт

Векторные шрифты хранят начертания шрифтов не в виде точек, а в виде набора векторов, задаваемых координатами. Каждый векторный шрифт поддерживает своё начертание, но, кроме того,

для жирного, или курсивного, или прямого начертания должен использоваться собственный набор. Векторные шрифты хорошо масштабируются.

Современные векторные шрифты (True Type Font .ttf, Post Script .ps и т.д.) строятся на похожих принципах.

Вернёмся к точечным шрифтам. Возьмём, например, матрицу 8x8:

	Начертание символа									**Единицы и нули**									**Двоичная запись**		**Шестнадцатеричная запись**
	1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8				
1			■	■	■	■				0	0	1	1	1	1	0	0		0011 1100		3C
2		■					■			0	1	0	0	0	0	1	0		0100 0010		42
3	■							■		1	0	0	0	0	0	0	1		1000 0001		81
4	■		■			■		■		1	0	1	0	0	1	0	1		1010 0101		A5
5	■							■		1	0	0	0	0	0	0	1		1000 0001		81
6	■			■	■			■		1	0	0	1	1	0	0	1		1001 1001		99
7		■					■			0	1	0	0	0	0	1	0		0100 0010		42
8			■	■	■	■				0	0	1	1	1	1	0	0		0011 1100		3C

Таким образом, символ в файле fnt 8x8 в шестнадцатеричном виде будет представлен последовательностью: 3C 42 81 A5 81 99 42 3C

Если следующий символ представлял, допустим, кружок, его последовательность была следующей:

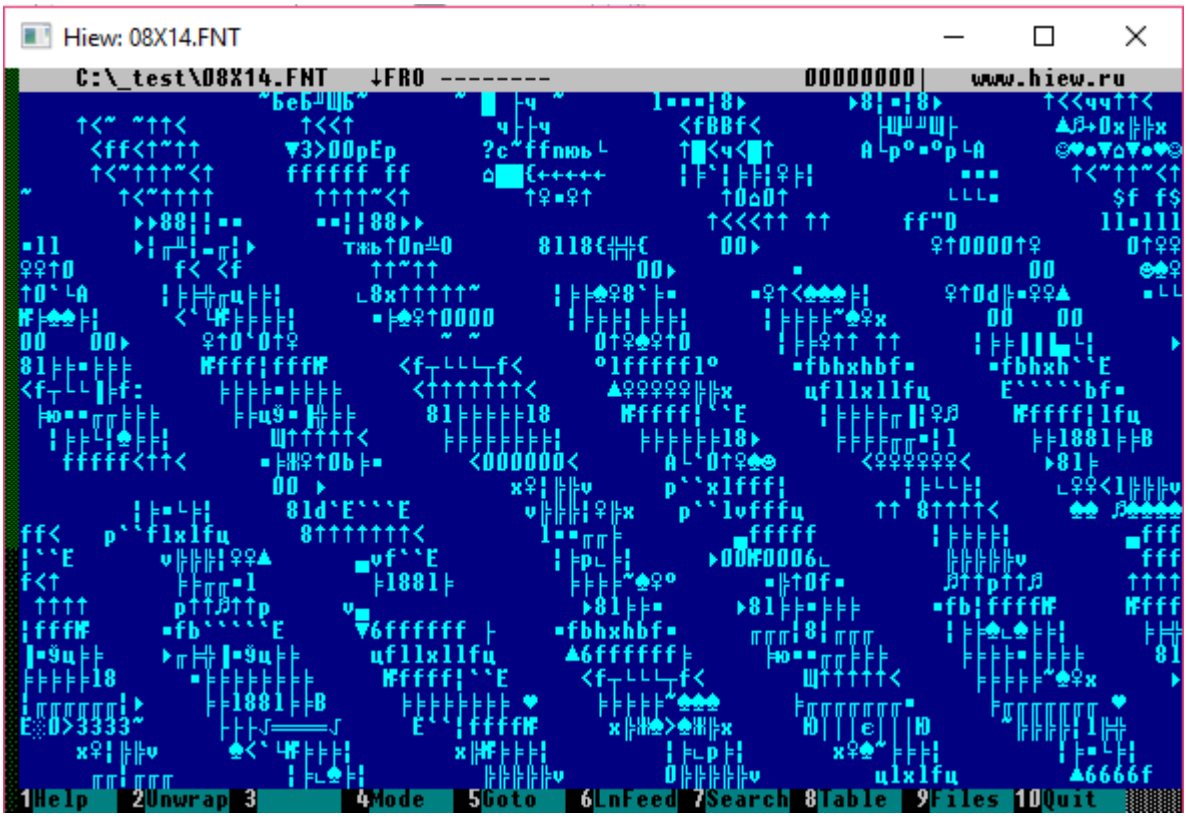
3C 42 81 81 81 81 42 3C

В файле все символы идут подряд: 3C 42 81 A5 81 99 42 3C 3C 42 81 81 81 81 42 3C.

Сам файл .fnt – двоичный, то есть в него записываются числа в двоичном виде.

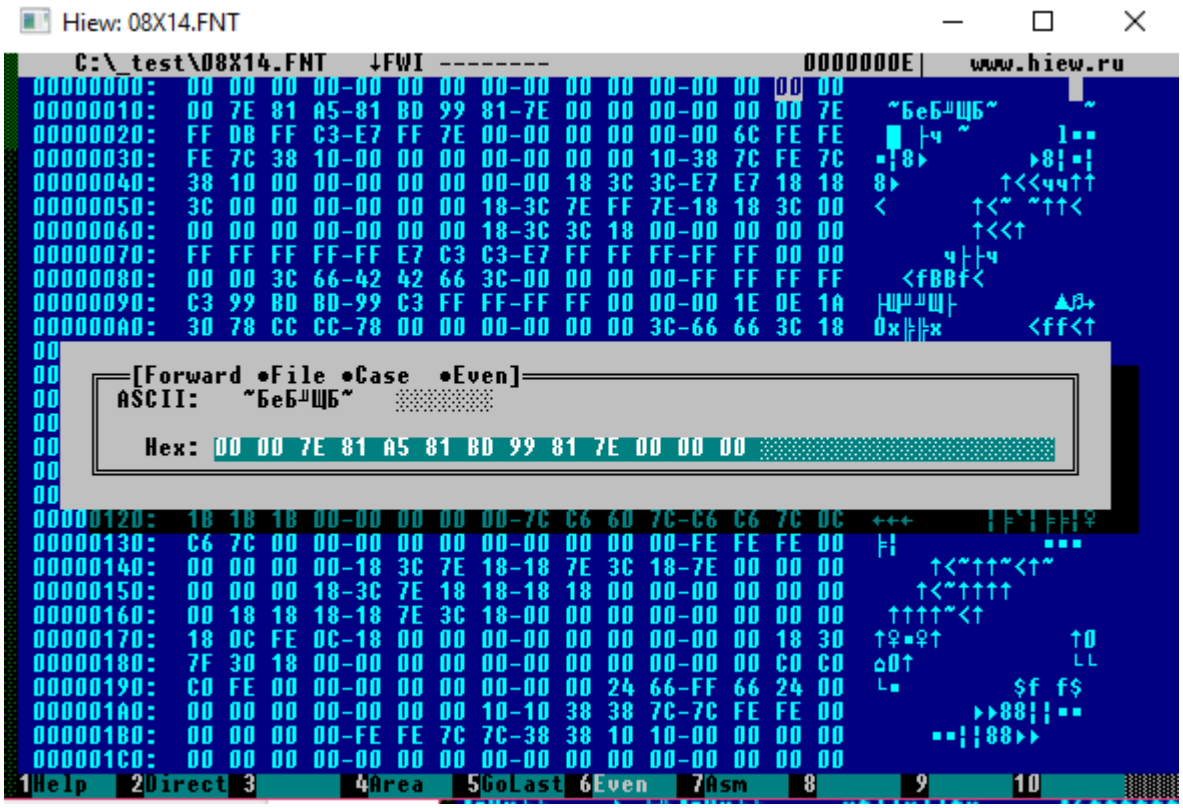
В MS DOS использовалась кодировка из 256 символов. Фактически заполнение файла fnt и определяло ту кодировку, которая использовалась. Но если такая ситуация была характерна для точечных и векторных шрифтов, она не обязательно могла соблюдаться во всех случаях. Вспомните: первые машины использовали для ввода-вывода информации печатную машинку. Некоторые устройства (калькуляторы, индикаторы на микроволновых печах, стиральных машинах и т.д.) используют семисегментные индикаторы. Каким образом используется кодирование в таких случаях?

Так выглядит шрифт .fnt при открытии на чтение:



Видны повторяющиеся последовательности, разделенные нулевыми символами. Это и есть точечные образы шрифтов.

Повторяющиеся паттерны формируют вполне узнаваемый рисунок. Колонки смещаются вправо, так как число отображаемых символов не кратно 14 (каждый «символ» на самом деле двухбайтное число, которое кодирует строку точек символа. Кстати, порядок байтов здесь не действует строго справа налево, так, как это будет выводиться на экране).



Позиция первого символа имеет код 1 – в ASCII это управляющий символ, но если выводить символ на экран, отобразится значок смайла (сравните с представленным выше кодированием смайла).

Кодировки

Кодировки произошли от азбуки Морзе и кодов Бодо, в честь которого названа единица измерения передачи информации – бод. Сейчас все большее распространение получает Юникод (UTF-8, с переменной длиной, применяемый для передачи в сети Интернет, и UTF-16, с двумя байтами на символ, применяемый в Windows, а также со всеми проблемами, связанными с проблемой порядка байт и использованием специального символа Byte Order Mark – BOM), до этого применялись семибитные и восьмибитные кодировки.

Семибитная кодировка позволяла закодировать 2^7 =127 позиций, включая управляющие символы (перевод каретки, новая строка, звонок печатающей машинки, табуляция, возврат строки и т.д.) и алфавитно-цифровые символы одного языка (латинского).

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

В отечественных машинах использовался транслитерированный вариант семибитной кодировки ASCII – KOI-7, в котором символы латиницы были транслитерированы в кириллицу, что позволяло работать и обеспечивать некоторую языковую совместимость (из транслита можно было понять, о чём идет речь, хотя это и было неудобно). В семибитной кодировке пришлось бы использовать русскоязычные команды и управляющие конструкции.

Семибитная кодировка является не совсем экономной, так как 1 бит в ней остаётся незадействованным. При этом добавление всего лишь одного бита позволяет увеличить таблицу в два раза.

Восьмибитная кодировка позволяет закодировать 2^8 =256 символов.

Если особенности ASCII более или менее понятны – по существу, первые 127 символов существуют в неизменном порядке до сих пор в большинстве кодировок, – заполнение таблицы символов остаётся вопросом.

Например, можно заполнить таблицу недостающими символами европейских алфавитов (немецкого, исландского, французского, чешского) – всем хватит, кроме кириллических и иных шрифтов.

ANSI Extended ASCII (Windows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	□	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	□	□
9	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	□	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

cplusplus.com

Можно во вторую половину сложить то, что было в koi7 – получится koi8r:

─ 128	│ 129	┌ 130	┐ 131	└ 132	┘ 133	├ 134	┤ 135	┬ 136	┴ 137	┼ 138	▀ 139	▄ 140	█ 141	▌ 142	▐ 143
░ 144	▒ 145	▓ 146	⌠ 147	■ 148	∙ 149	√ 150	≈ 151	≤ 152	≥ 153	154	⌡ 155	° 156	² 157	· 158	÷ 159
═ 160	║ 161	╒ 162	ё 163	╓ 164	╔ 165	╕ 166	╖ 167	╗ 168	╘ 169	╙ 170	╚ 171	╛ 172	╜ 173	╝ 174	╞ 175
А 176	Б 177	В 178	Г 179	Д 180	Е 181	Ж 182	З 183	И 184	Й 185	К 186	Л 187	М 188	Н 189	О 190	П 191
Р 192	С 193	Т 194	У 195	Ф 196	Х 197	Ц 198	Ч 199	Ш 200	Щ 201	Ъ 202	Ы 203	Ь 204	Э 205	Ю 206	Я 207
а 208	б 209	в 210	г 211	д 212	е 213	ж 214	з 215	и 216	й 217	к 218	л 219	м 220	н 221	о 222	п 223
р 224	с 225	т 226	у 227	ф 228	х 229	ц 230	ч 231	ш 232	щ 233	ъ 234	ы 235	ь 236	э 237	ю 238	я 239
Ё 240	ё 241	╟ 242	╫ 243	╪ 244	╩ 245	┘ 246	┌ 247	■ 248	∙ 249	√ 250	ⁿ 251	² 252	■ 253	nbsp 254	255

Возможно, это не самый удобный вариант для работы: сортировка по алфавиту становится нетривиальной, а задача автоматической транслитерации – не самая необходимая.

Кодировка IBM (включая первые 127 символов):

[	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺ 1	☻ 2	♥ 3	♦ 4	♣ 5	♠ 6	• 7	8	○ 9	10	11	12	13	14	15
1	► 16	◄ 17	18	! 19	20	21	_ 22	23	↑ 24	↓ 25	→ 26	← 27	28	↔ 29	▲ 30	▼ 31
2	32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	}] 125	~ 126	⌂ 127
8	Ç 128	ü 129	é 130	â 131	ä 132	à 133	å 134	ç 135	ê 136	ë 137	è 138	ï 139	î 140	ì 141	Ä 142	Å 143
9	É 144	æ 145	Æ 146	ô 147	ö 148	ò 149	û 150	ù 151	ÿ 152	Ö 153	Ü 154	¢ 155	£ 156	¥ 157	₧ 158	ƒ 159
A	160	í 161	ó 162	ú 163	ñ 164	Ñ 165	ª 166	º 167	¿ 168	⌐ 169	¬ 170	½ 171	¼ 172	¡ 173	« 174	» 175
B	░ 176	▒ 177	▓ 178	| 179	┤ 180	╡ 181	╢ 182	╖ 183	╕ 184	╣ 185	║ 186	╗ 187	╝ 188	╜ 189	╛ 190	┐ 191
C	└ 192	┴ 193	┬ 194	├ 195	─ 196	┼ 197	╞ 198	╟ 199	╚ 200	╔ 201	╩ 202	╦ 203	╠ 204	═ 205	╬ 206	╧ 207
D	╨ 208	╤ 209	╥ 210	╙ 211	╘ 212	╒ 213	╓ 214	╫ 215	╪ 216	┘ 217	┌ 218	▉ 219	▄ 220	▌ 221	▐ 222	▀ 223
E	α 224	ß 225	Γ 226	π 227	Σ 228	σ 229	µ 230	τ 231	Φ 232	Θ 233	Ω 234	δ 235	∞ 236	φ 237	ε 238	∩ 239
F	≡ 240	± 241	≥ 242	≤ 243	⌠ 244	⌡ 245	÷ 246	≈ 247	° 248	∙ 249	· 250	√ 251	ⁿ 252	² 253	■ 254	□ 255

Кодировка символов, предложенная IBM (соответствует ASCII - кодировке)

В DOS применялась система кодовых страниц. Часть символов замещались национальными символами, часть включала псевдографику, которая не менялась.

Пример таблицы 866, используемой в MS DOS (её поддерживал драйвер keyrus):

А 128	Б 129	В 130	Г 131	Д 132	Е 133	Ж 134	З 135	И 136	Й 137	К 138	Л 139	М 140	Н 141	О 142	П 143
Р 144	С 145	Т 146	У 147	Ф 148	Х 149	Ц 150	Ч 151	Ш 152	Щ 153	Ъ 154	Ы 155	Ь 156	Э 157	Ю 158	Я 159
а 160	б 161	в 162	г 163	д 164	е 165	ж 166	з 167	и 168	й 169	к 170	л 171	м 172	н 173	о 174	п 175
░ 176	▒ 177	▓ 178	| 179	┤ 180	╡ 181	╢ 182	╖ 183	╕ 184	╣ 185	║ 186	╗ 187	╝ 188	╜ 189	╛ 190	┐ 191
└ 192	┴ 193	┬ 194	├ 195	─ 196	┼ 197	╞ 198	╟ 199	╚ 200	╔ 201	╩ 202	╦ 203	╠ 204	═ 205	╬ 206	╧ 207
╨ 208	╤ 209	╥ 210	╙ 211	╘ 212	╒ 213	╓ 214	╫ 215	╪ 216	┘ 217	┌ 218	▉ 219	▄ 220	▌ 221	▐ 222	▀ 223
р 224	с 225	т 226	у 227	ф 228	х 229	ц 230	ч 231	ш 232	щ 233	ъ 234	ы 235	ь 236	э 237	ю 238	я 239
Ё 240	ё 241	Є 242	є 243	Ї 244	ї 245	Ў 246	ў 247	° 248	∙ 249	· 250	√ 251	№ 252	¤ 253	■ 254	nbsp 255

В дальнейшем в Windows для графического отображения использовалась кодировка CP-1251, затем UTF-16. Если в CP-1251 не было символов псевдографики, то в UTF-16 есть не только псевдографические символы, но и графические изображения управляющих символов. К слову, они получили новые коды, а первые 32 символа по-прежнему используются как управляющие.

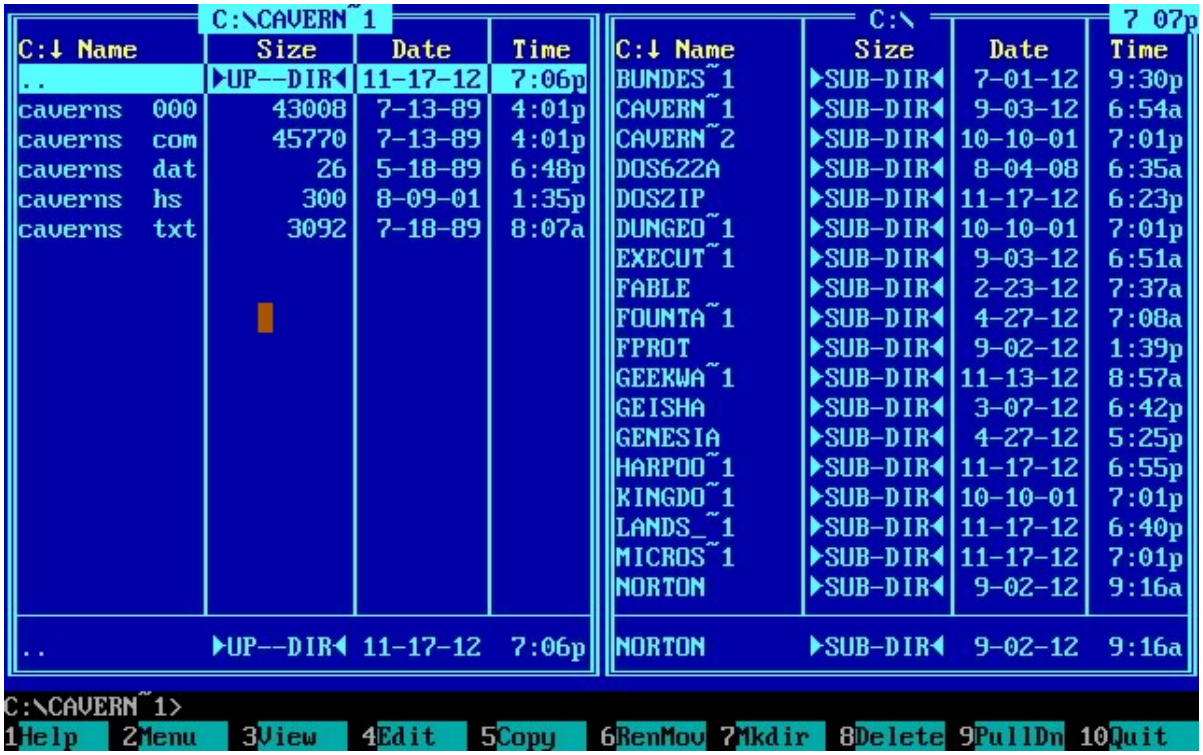
Управляющие и псевдографические символы

В досовских кодировках управляющие и псевдографические символы часто имели собственное начертание, но обработка символа осуществлялась в зависимости от используемого прерывания и функции. Так, при выводе строки печать символа 7 выводила гудок на динамик (на экран ничего не выводилось), печать символа 8 приводила к перемещению курсора назад, 9 (табуляция) – к перемещению курсора на 8 позиций вправо, символ 10 служил для обозначения новой строки, 13 – для перевода строки. Однако, например, утилита ndd (norton disk doctor) и подобные им (например, defrag) использовали 7, 8, 9, 10 вместе с символами 176-178 для отображения на экране секторов диска, а в других программах – для рисования полос прокрутки. Символ 13 печатал нотку. Символ 27→ использовался для изображения полосы прокрутки, его появление в файле могло означать конец, и, например, утилита copy без ключа /b обрезала бы данный файл до появления этого символа. Символ 28← на самом деле кодировал Escape. Служебные символы активно использовались и такими программами, как Lexicon, для задания новых страниц и других элементов разметки.

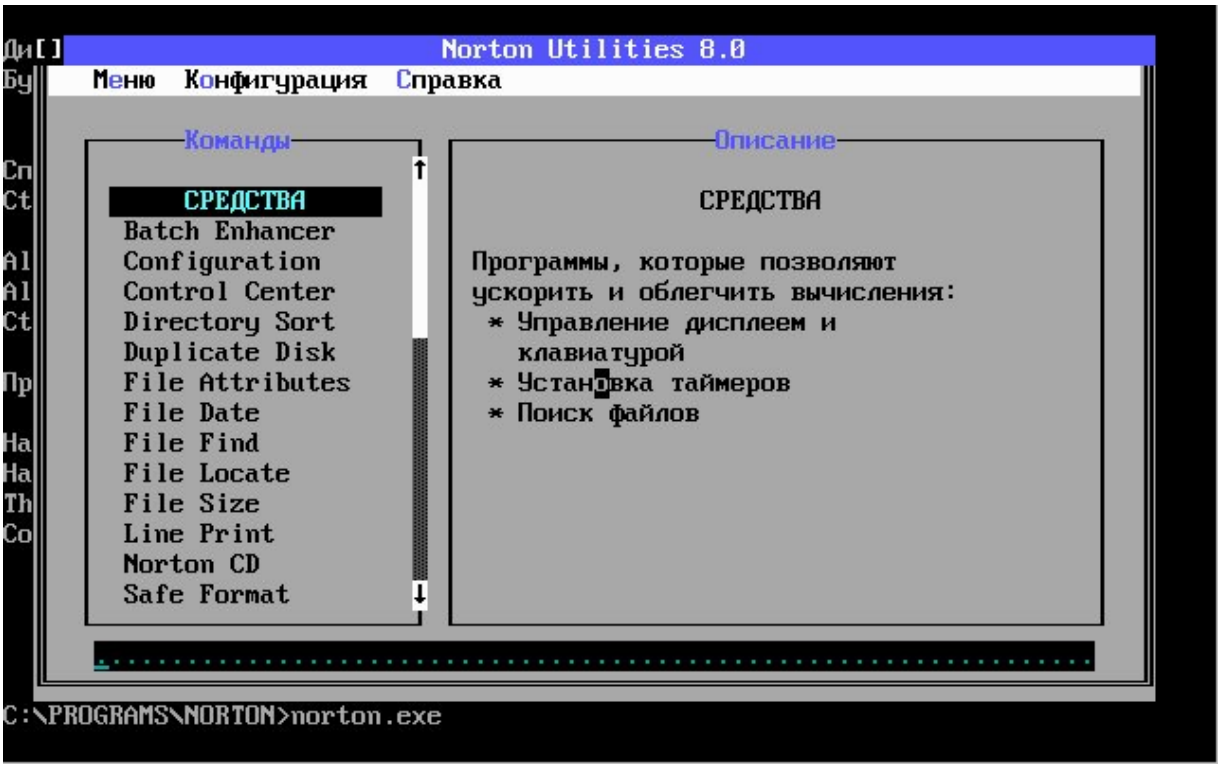
Формат файлов может различаться от операционной системы к операционной системе. Так, в Windows для новой строки использовались последовательность 13 и 10 символа (\r\n), а в linux – только 10ый (\n). То же самое касается и передачи строк в системные вызовы. Обычные варианты – передача длины и самого массива байт, однако возможны варианты и строк переменной длины. Например, при программировании в Си \0 (нулевой символ) может заканчивать строку, а некоторые функции MS DOS как конец строки воспринимали символ $. Фактически к управляющим символам относится и BOM, если в файле используется Юникод. К слову, ряд функциональных кнопок генерируют двухбайтный код (для F1-F12), а некоторые – однобайтный. Нажатие на кнопку Backspace отправляет программе символ с кодом 8, но вместо его печати программа перемещает курсор. Esc генерирует код. Enter генерирует код 13.

Возможность переопределения таблицы символов (в том числе и в реальном времени) вкупе с начертанием «управляющих символов» и псевдографических давали возможность рисовать псевдографические интерфейсы и даже псевдографический курсор мыши (вместо инвертированного

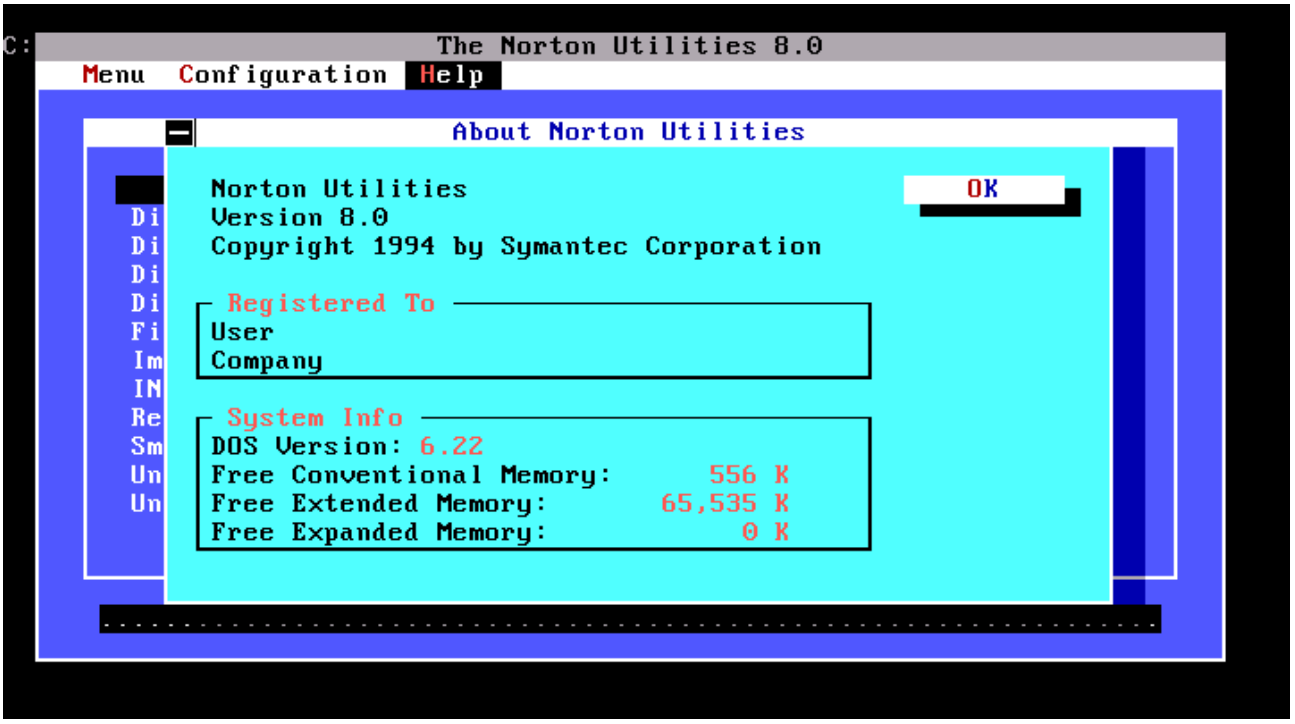
цвета и фона знакоместа). Фактически мышь работала в пиксельном режиме, и несколько символов псевдографики перерисовывались в реальном времени путём вычисления позиции перекрываемых символов и отрисовки на их месте новых из псевдографики с наложенным курсором мыши.



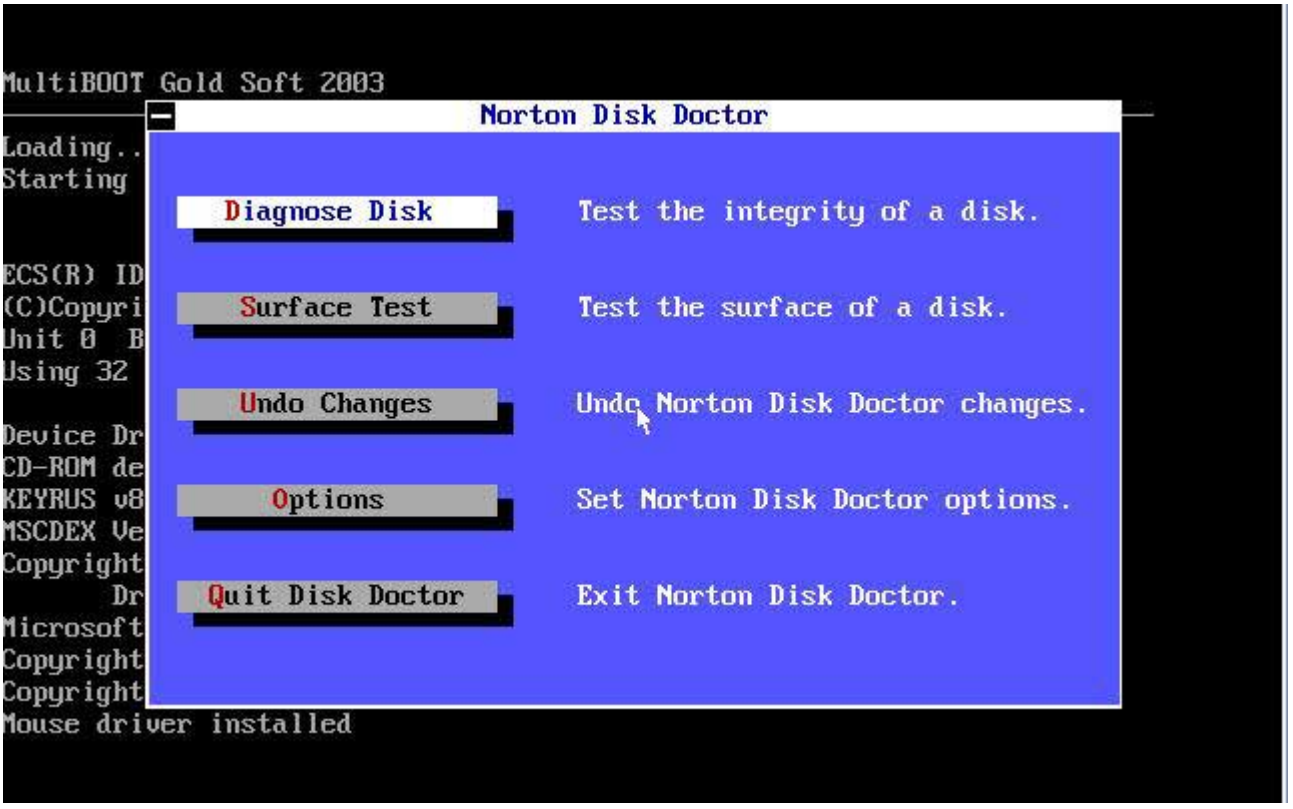
Псевдографические символы и начертания управляющих используются для формирования интерфейса в Norton Commander. Существует свободная библиотека ncurses, которая позволяет с использованием псевдографических символов формировать интерфейсы в консольных программах и встраиваемых системах (при наличии экрана).



Псевдографика в Norton Utilites:

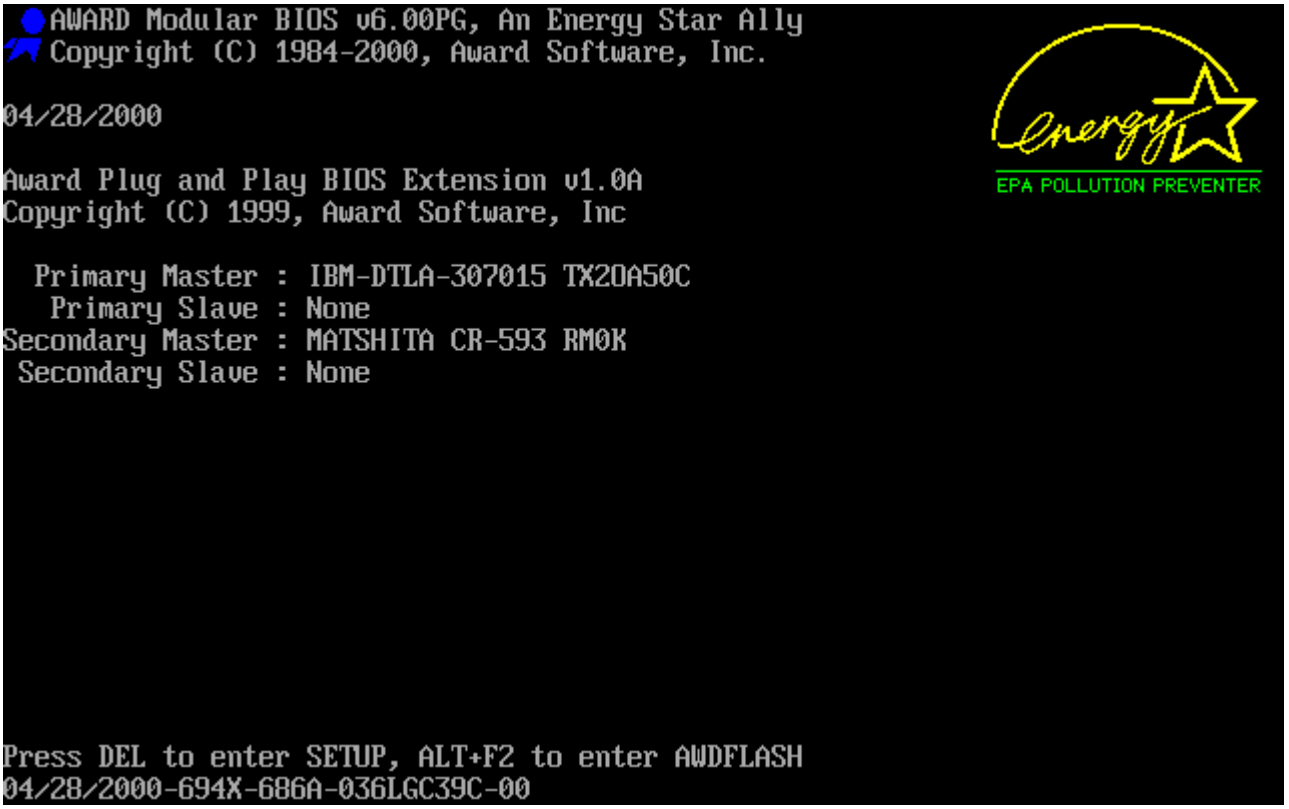


Псевдографика в Norton Utilites в текстовом режиме с переопределением знакогенератора (сравните с верхней картинкой):



Перед нами пример переопределения символов для более красивого интерфейса в NDD. Обратите внимание на псевдографический курсор мыши: он перемещался попиксельно. На месте е на экране напечатан не символ е, а псевдографический символ, полученный из матрицы символа е, с наложением фрагмента курсора в соответствии с его положением.

Переопределение символов позволяет включать в текстовом режиме картинки (не путать с ASCII-арт – там используется существующий набор символов).



Обратите внимание на картинки: они отрисованы в текстовом режиме благодаря переопределению таблиц символов.

Модульность DOS

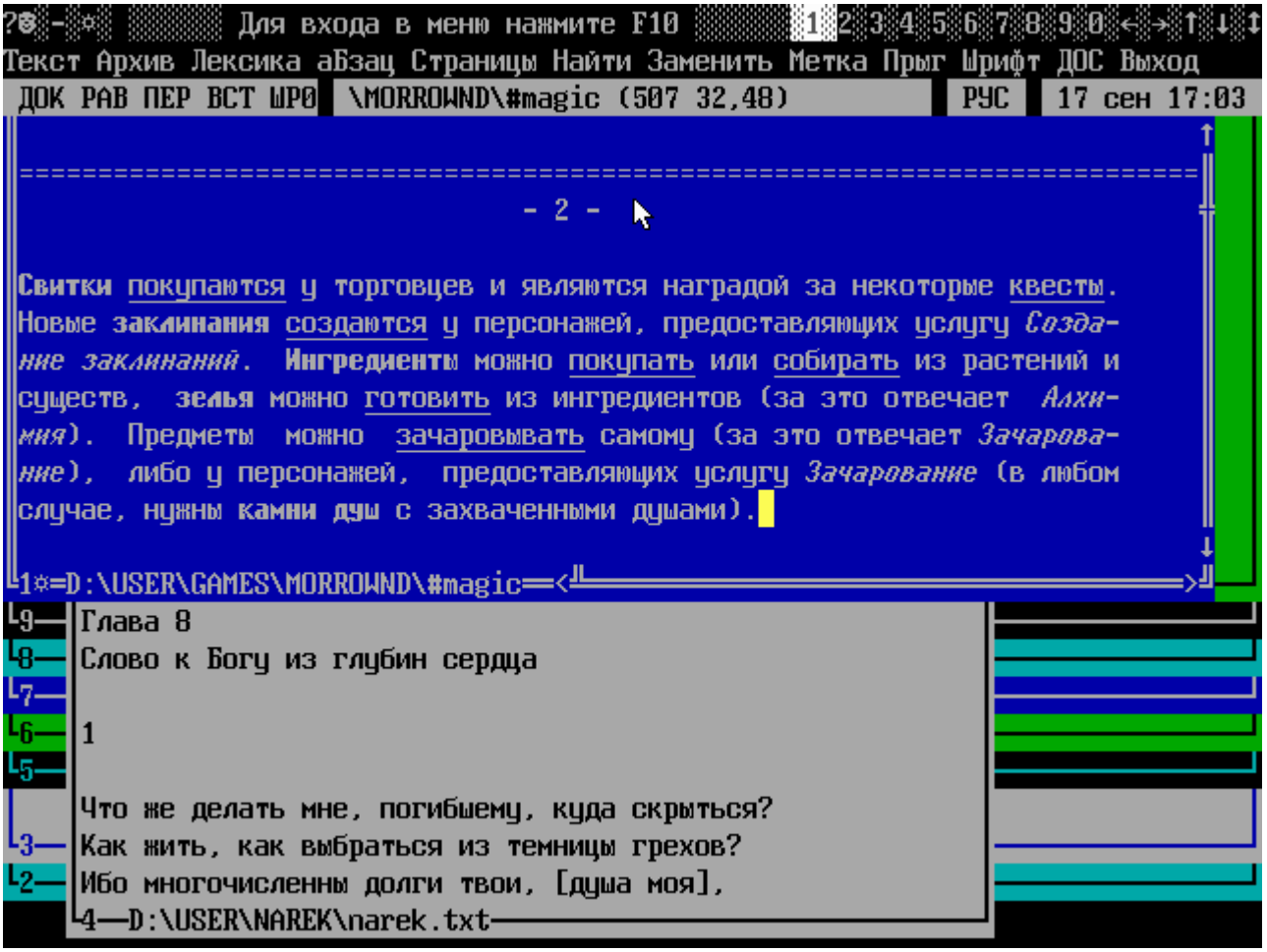
Итак, DOS – не столько операционная система с ярко выраженными системными и пользовательскими интерфейсами, сколько конструктор. Программы в ней могли работать с оборудованием напрямую, могли использовать собственные драйверы и переопределять прерывания.

Текстовый процессор Lexicon использовал собственные драйверы для экрана и шрифтов, Norton Utilites – собственные наборы шрифтов и драйвер мыши, игры могли полностью самостоятельно взаимодействовать с графикой на аппаратном уровне. Эти программы сами были мини-операционными системами, оставляя DOS роль загрузчика или операционной системы в самой базовой части.

Такие программы, как Dosshell, Desqview и Windows 1, в дальнейшем сами эволюционировали в операционные системы. Windows вплоть до 3.11 (а в некоторой степени и 9x), с одной стороны, представляли собой программу-оболочку для DOS, с другой стороны – использовали значительный набор собственных драйверов, системных вызовов и т.д., что позволяет говорить об операционной системе в операционной системе. Впрочем, DOS и впоследствии играла роль загрузчика Windows и консоли восстановления. Более того, такие загрузчики, как loadlin, позволяют загрузить из DOS даже Linux.

В списке драйверов мы не упомянули himem.sys и emm386.exe. Формально это драйверы, которые загружаются, если упомянуты в config.sys, фактически же ситуация с ними обстоит несколько сложнее.

Так, начиная с версий MS DOS 4.01 – 5.0 при использовании менеджера памяти EMM386 (и его аналогов других разработчиков) операционная система MS-DOS работает как задача в виртуальном режиме. EMM386 в этом случае является подобием операционной системы защищённого режима, передавая большинство системных прерываний ядру MS-DOS в виртуальной задаче. Более того, виртуальный режим процессора 8086 позволил уже и в Windows выполнять приложения MS DOS. Himem.sys, предназначенный для поддержки HMA, был необходим и для запуска Windows 9x (что интересно, впервые поддержка High Memory Area появилась в Windows 2.0, стартовавшей из DOS, а затем уже механизм загрузки MS DOS в HMA был реализован в MS DOS 5.0). В приложениях использовались и такие 32-битные расширители, как QEMM, DOS/4GW и другие, позволяющие использовать до 64 Мбайт расширенной памяти.



Многооконный текстовый редактор Лексикон (использует собственные драйверы шрифтов).

Однозадачность

Что такое многозадачность? Возможны несколько её реализаций. Простейший случай – невытесняющая многозадачность. В MS DOS многие сложные программы, такие, как Lexicon, Quick Basic, Turbo Pascal, позволяли сделать «Выход в DOS»: программа останавливалась, запускалась новая копия command.com, можно было запустить другую программу (обычно на Norton Commander памяти не хватало, но можно было использовать аналогичный, но более экономичный Volkov Commander), а после завершения управление передавалось в вызвавшую программу – это и есть однозадачный режим. Если в системе реализована возможность остановки и передачи управления другой программе с возвратом в остановленную – это невытесняющая многозадачность. Её не так сложно реализовать и в MS DOS, написав драйвер, управляющий переключением задачами (по прерыванию с клавиатуры вызвать менеджер задач и передать управление в нужную задачу). Однако

во время выполнения одной задачи другие задачи простаивают. Вытесняющая многозадачность позволяет выделять программе уже не все время процессора полностью, а лишь квант времени, останавливая его не по нажатию специальной кнопки, а по таймеру или получению прерывания от оборудования.

Интересна реализация многозадачности в MS DOS: фактически её нет, но она может быть реализована благодаря механизму прерываний.

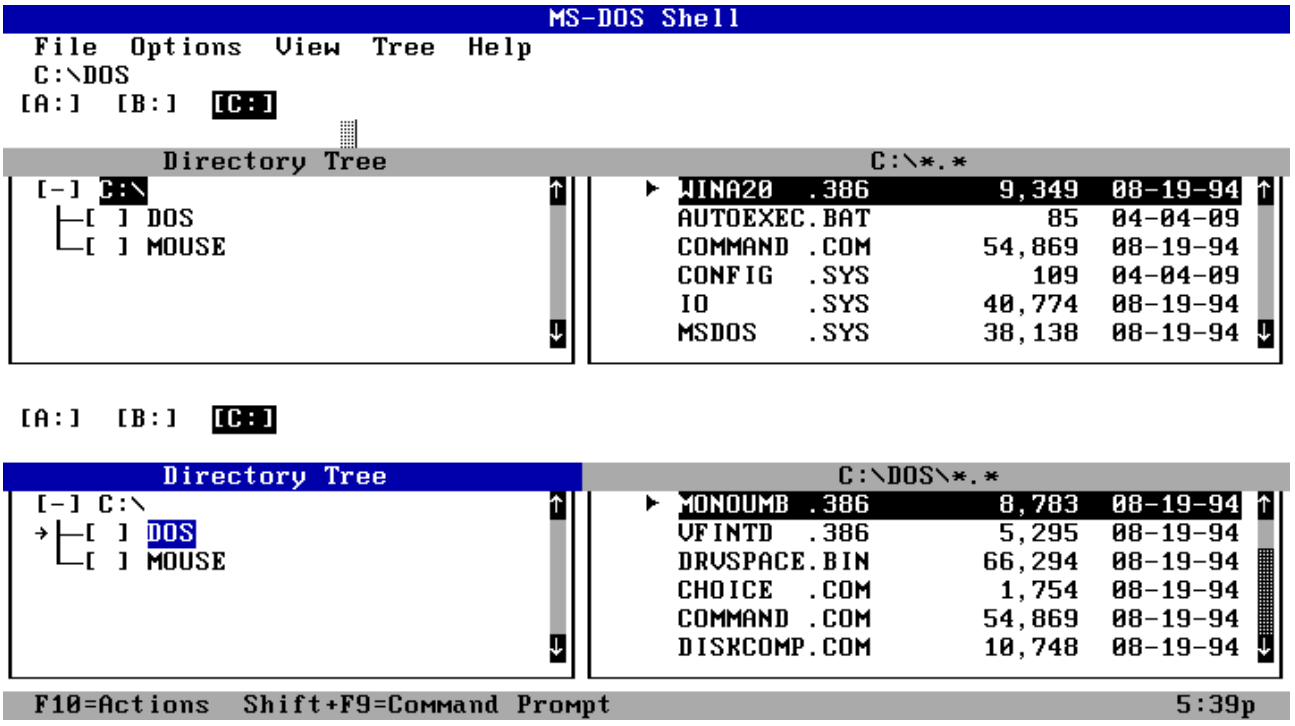
В 1988 году в Microsoft разрабатывалась MS DOS 4.0 с вытесняющей многозадачностью. Она в реальном режиме обладала вытесняющей многозадачностью, предназначенной для семейства процессоров 8086 (впоследствии эта возможность была удалена), включала перемещаемые и выгружаемые сегменты памяти для кода и перемещаемые сегменты данных – менеджер памяти Windows был версией менеджера памяти DOS 4, – и имела возможность динамического переключения экранов. Версия была экспериментальной и в массовую продажу не пошла: под именем MS DOS 4.0 была выпущена ОС без этих возможностей. Впрочем, данные наработки, как уже было сказано, были использованы в Windows.

В версии 4.01, которая также вышла под именем MS DOS 4.0, имелась графическая среда DOS Shell с невытесняющей многозадачностью.

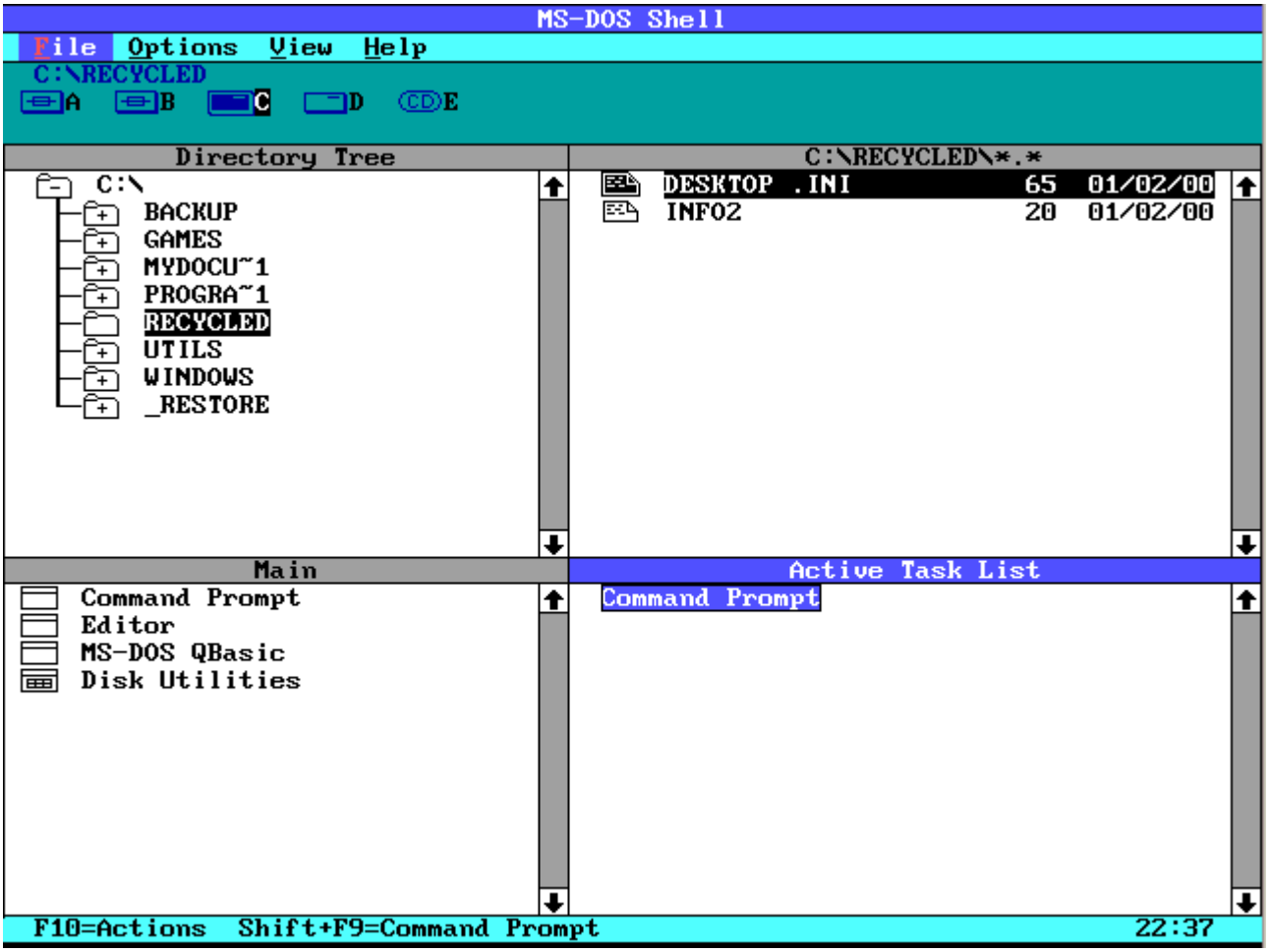
Существовали и альтернативные реализации многозадачности в MS DOS.

Интересно, что в MS DOS-совместимых операционных системах DR DOS 6.0 (1991 года) и Novell DOS 7.0 переключаемая многозадачность существовала. Ctrl-1, Ctrl-2 и т.д. позволяли переключаться между задачами (это напоминает переключение между терминалами в Linux: Alt-F1, Alt-F2), Ctrl-Esc – выйти на список задач. В Novell DOS 7.0 уже присутствовала вытесняющая многозадачность.

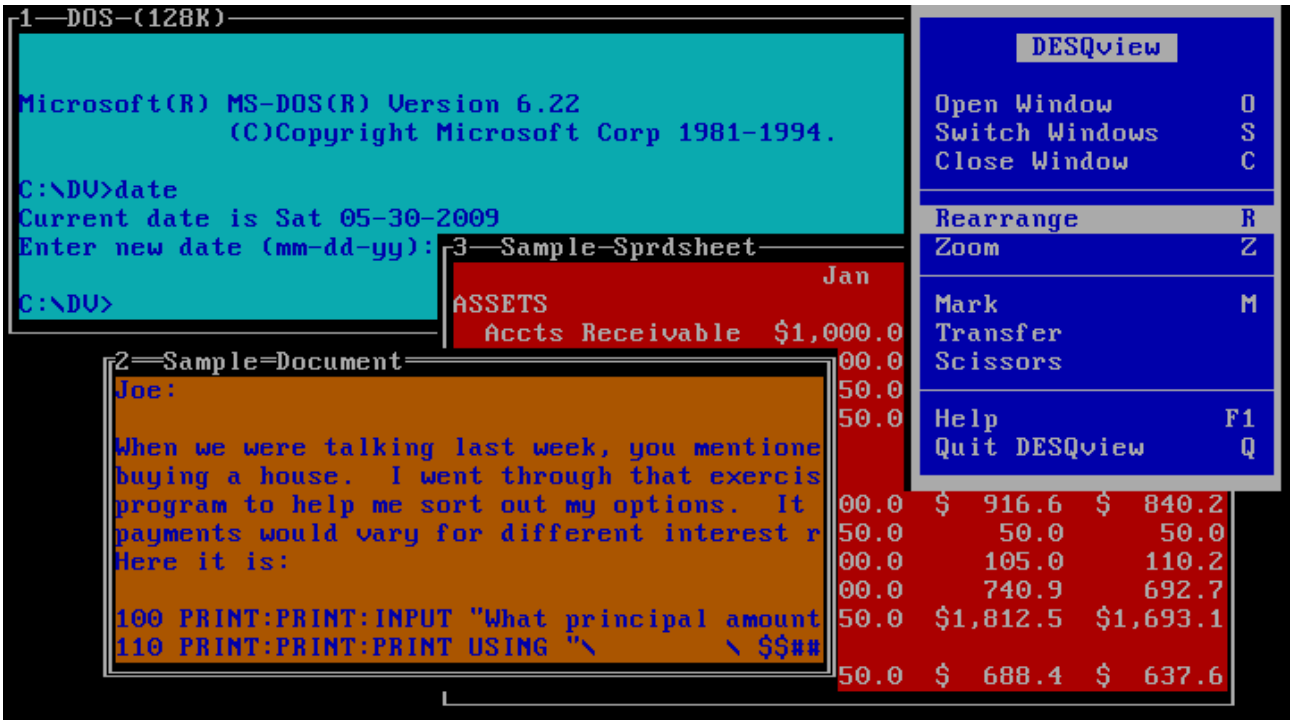
В 1980 сторонней компанией Quarterdeck Office Systems разрабатывалась оболочка для DOS DESQView, релиз которой последовал в 1985 году. Благодаря драйверу QEMM-386 (QEMM.COM) той же компании программа могла задействовать всю память. DESQView была сравнительно популярна, пока ее не вытеснили сначала DR DOS 6.0, а затем Microsoft Windows 3.11. В DESQView не было графического GUI, а попытка создать на её основе DV/X с использованием X Window System успеха не принесла.



DOS Shell

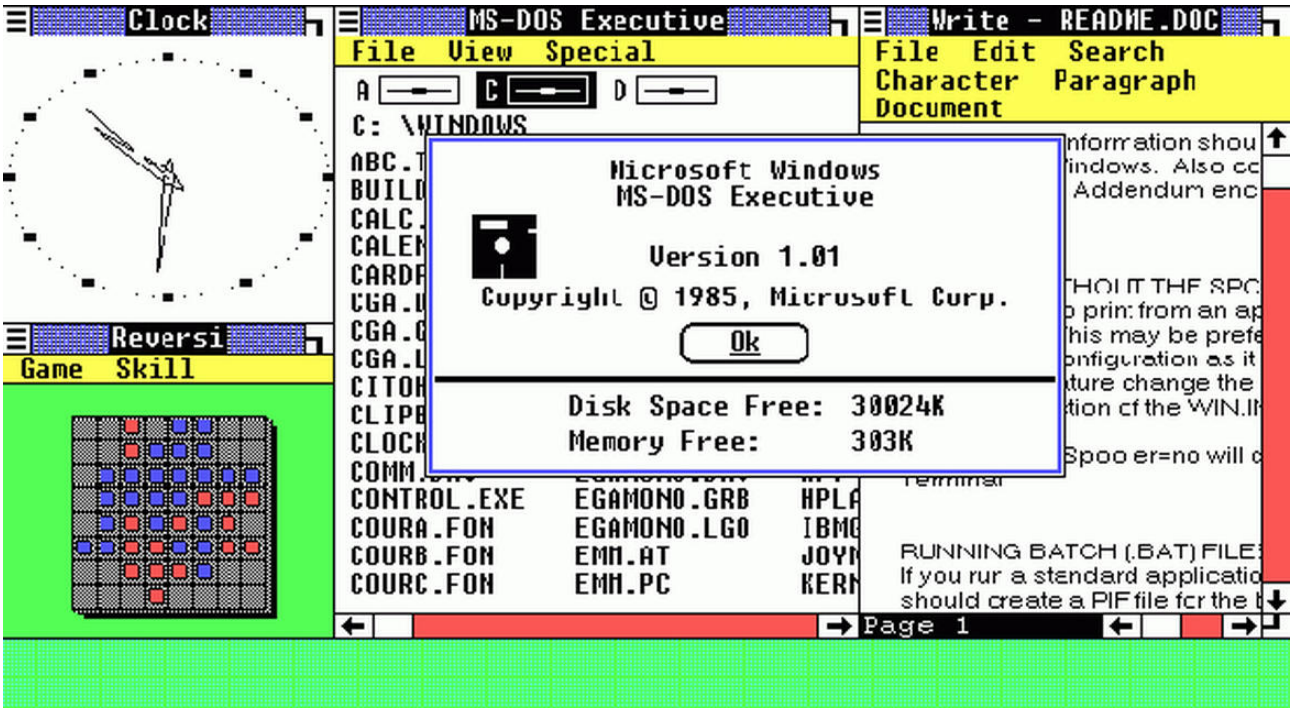


DOS Shell



DESQView – многооконность и кооперативная многозадачность.

Microsoft Windows 1.0 появилась из необходимости иметь графический интерфейс и реализовать многозадачность. На это повлияли Xerox Star и Apple Lisa (1981-82 годы), а также необходимость создать продукт для конкуренции с DESQView. Фактически это было приложение win.com, которое запускало оболочку MS-DOS Executive. MS-DOS Executive вытеснил и без того не пользовавшийся популярностью DOSshell.



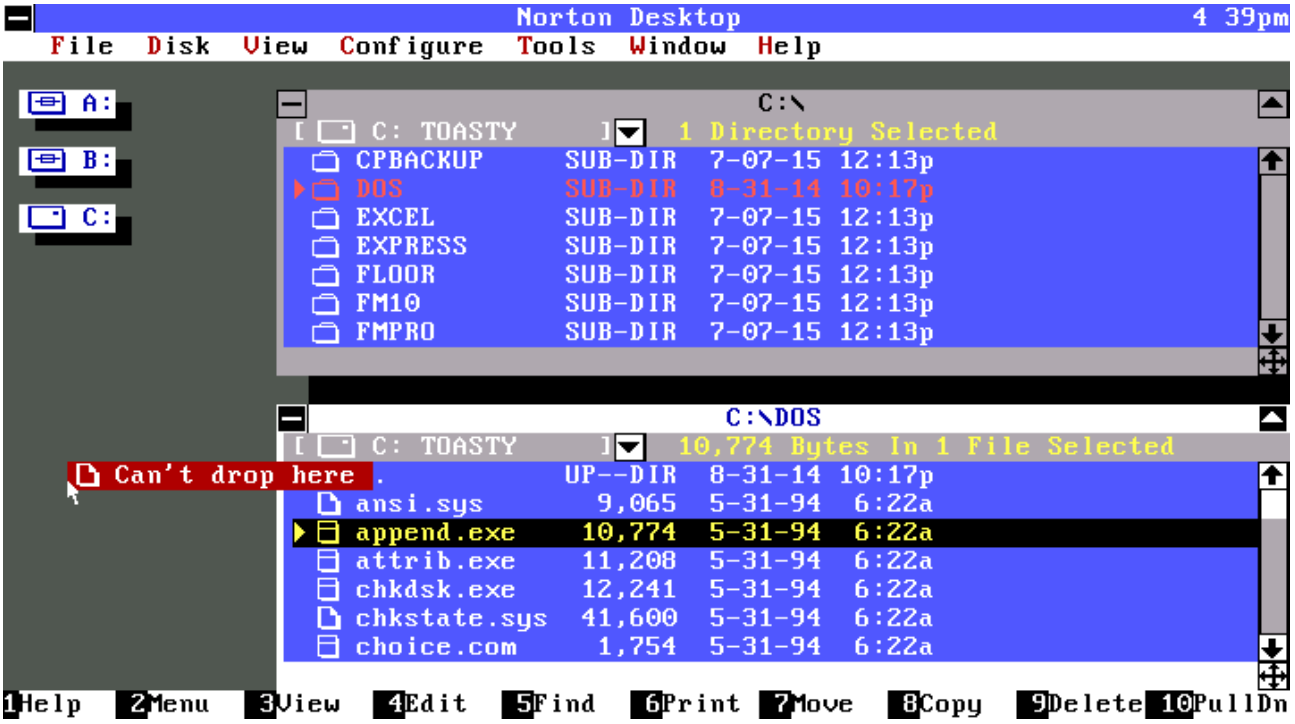
Windows 1.01 (сравните с DOSShell)

Из MS DOS Executive можно было выйти, вернув управление в однозадачный режим DOS. Подобная ситуация сохранялась довольно долго, в то время как Windows из приложения компонента MS DOS развивалась в отдельную систему, фактически поглощая оставшиеся компоненты DOS.

Оболочки

Оболочки для ОС могут представлять собой интерпретатор командной строки (Бейсик в многочисленных микрокомпьютерах на заре их появления, COMMAND.COM, bash), текстовый менеджер файлов с возможностью навигации по файлам, редактирования файлов и запуска программ. Популярен был такой интерфейс, как у Norton Commander: например, Norton Commander, Volkov Commander, DOS Navigator – для DOS, mc – Midnight Commander – для Linux, FAR для Windows. Оболочки могут иметь и альтернативную, зачастую псевдографическую концепцию. Отдельно стоит рассматривать более сложные и многокомпонентные графические оболочки (Windows с Explorer, X11 Server с реализацией среды рабочего стола Gnome, KDE или Unity с оболочкой, например, Nautilus). Оболочки сами по себе не обязаны реализовывать многозадачность, хотя DOSShell с этим справлялся, а Windows в принципе в чистом виде представляет собой не одну оболочку, а набор компонентов. В качестве примера оболочки без многозадачности наряду с Norton Commander можно привести оболочку от Symantec. Кроме того, были попытки создать графическую версию для замещения встроенной оболочки в Windows 3.11. В принципе в MS DOS также вместо coommand.com можно запустить иную программу, от оболочки до альтернативных интерпретаторов командной строки, так как строка shell=C:\ndos.com C:\ в config.sys позволяла загрузить ndos.com – альтернативу командному интерпретатору от Norton Utilites. Так же и в Windows 3.11 вместо

Диспетчера программ и в Windows 95 вместо Explorer, запускающихся по умолчанию, можно настроить запуск иной программы.



Norton Desktop

На наш взгляд, Windows 3.11 стоит считать отдельной от MS DOS системы с учётом того, что MS DOS фактически становился её частью. Похожая ситуация сложилась с X Windows Server в Linux, хотя там немного иначе выглядит баланс между исходной операционной системой и графической системой (в Windows значительная часть функциональности тесно интегрирована с графической средой)

Утилиты

Утилиты – это полностью пользовательские программы, незаменимый компонент системы, напрямую не имеющий отношения к ядру. В папке msdos операционной системы DOS содержалось большое число служебных программ. Сама MS DOS оказалась в какой-то степени гибридом CP/M и Unix, позаимствовав ряд решений, утилит, файловых конвейеров из UNIX. Ряд возможностей командного интерпретатора встроен в command.com, но большинство из них, как и в UNIX и Linux, – отдельные программы, имеющие расширение .com или .exe. Среди них были утилиты fdisk, attrib, sys, label, chkdsk, undelete, subst, defrag.

Иногда встроенных утилит было недостаточно, и устанавливались дополнительные утилиты, такие, как Norton Utilites, DOS Tools (утилиты для парковки головок винчестера и остановки системы – park.com, shutdown.com) и другие программы: упомянутые выше оболочки, редакторы (в том числе и такие, как hiew), языки программирования.

Примечательно, что в состав MS DOS входил редактор и интерпретатор диалекта языка Бейсик QBASIC. Он представлял собой урезанную версию коммерческого продукта Microsoft Quick Basic 4.5 без возможности компиляций и некоторых функций для работе с прерываниями – тем не менее, подобные функции можно было реализовать самостоятельно благодаря возможности непосредственно выполнять Inline-код. Еще более интересно, что QBASIC.EXE был гораздо более важным компонентом MS DOS последних версий, чем это может показаться. Программы EDIT.COM и

HELP.COM использовали встроенные редактор и справочную систему программы QBASIC.EXE, фактически просто запуская его с определёнными ключами.

Архитектура ядра операционной системы

Кольца защиты процессора

В MS DOS не было никакого разграничения процессов: ошибка в приложении могла остановить всю операционную систему. Более того, реальный режим, в котором изначально работал MS DOS, не предоставлял механизмов защиты. Тем не менее, уже в Intel 80386 такие возможности возникли, появился защищённый режим. Несмотря на то, что многозадачность реализовывала операционная система (в принципе возможна и аппаратная реализация), для защиты процессов процессор предоставлял механизм колец защиты и страничную организацию памяти. Программы, предназначенные для незащищённого (так называемого реального) режима, в котором работала DOS, и защищённого режима, в общем случае не совместимы. В защищённом режиме невозможно использование прерываний BIOS и запуск программ для реального времени. Тем не менее такую возможность даёт виртуальный режим.

Существуют четыре уровня привилегий – четыре кольца защиты, пронумерованные от 0 (наиболее привилегированный уровень), до 3 (наименее привилегированный уровень) по отношению к ресурсам, на которые распространяется действие механизмов защиты процессора: среди них память, порты ввода / вывода и возможность выполнения некоторых инструкций. Каждую команду процессор x86 выполняет, находясь на том или ином уровне привилегий: от этого зависит, что может и чего не может сделать код. Некоторые инструкции могут быть выполнены на уровне привилегий 0 и при попытке выполнения их на ином уровне привилегий приведут к вызову исключения. Также исключение будет вызвано при недопустимом с точки зрения защиты обращению к памяти или устройствам.

Несмотря на наличие четырех уровней, обычно используется всего два уровня – 0 и 3, предназначенные для ядра и для пользовательского пространства. Фактически программа не может производить никакие действия с файлами, сетью или устройствами сама: для этого она обращается к функциям ядра. Возможность реализации тех или иных функций в уровне 0 или уровне 3 определяет архитектуру ядра. Возможность доступа приложений к уровню 0, пусть и в целях совместимости, делает систему нестабильной – для сравнения соотнесём её с Windows 95.

Архитектуры ядер

О вызове функций ядра мы уже имеем представление. Важно, что в этом случае процессор переходит на другой уровень защиты – в целом это замедляет быстродействие.

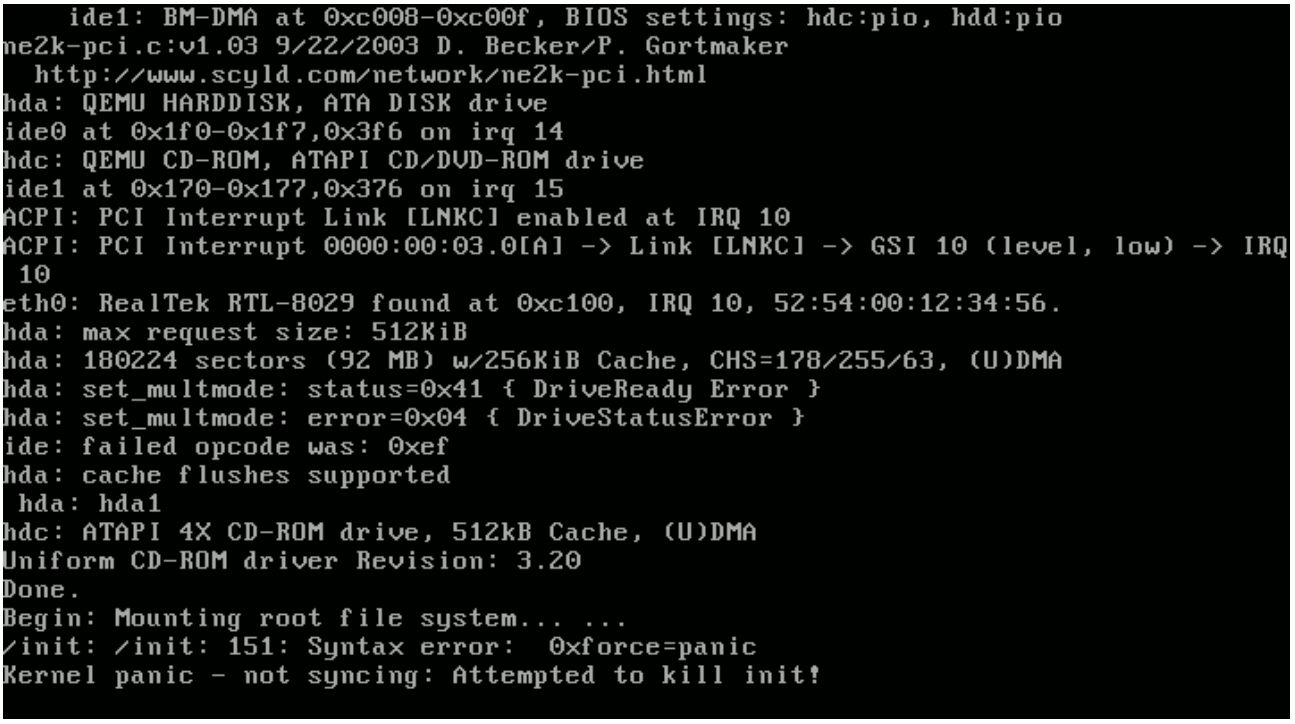
При выборе между быстродействием и безопасностью приходится следовать той или иной архитектуре ядра.

Рассмотрим её основные вариации.

Монолитное ядро

Все драйверы выполняются в пространстве ядра. Пример монолитного ядра – ядро Linux. Изначально драйверы требовалось скомпилировать в ядро, после чего дальнейшее развитие привело к появлению модульных ядер.

Если ошибка на уровне приложения приводит к остановке самого приложения, ошибка в драйвере приводит к краху всей системы. В Linux такое событие называется Kernel panic.



Модульное ядро

Модульное ядро – разновидность монолитного ядра, позволяющая подгружать и выгружать драйверы для выполнения в пространство ядра без компиляции. Современные дистрибутивы Linux используют модульное ядро.

Команды insmod и modprobe позволяют загружать и выгружать модули ядра, lsmod – посмотреть список загруженных модулей. Insmod работает с файлом ядра, lsmod оперирует с зависимостями.

Микроядро

Микроядро – концепция, призванная решить проблему падения системы при крахе драйвера. В концепции микроядра драйверы выполняются в отдельном пространстве. Падение драйвера не приводит к краху ядра. По существу, ядро в данном случае служит лишь связующим звеном между множеством компонентов. Достоинством микроядра является его высокая надёжность, недостатком – дополнительные затраты на переключение контекста.

Примерами микроядер можно считать ядро Minix 3 от Эндрю Таненбаума, ядро Mach, на котором базируется ряд известных ОС, исследовательская Microsoft Singularity. Также одной из лучших реализаций концепции микроядерной ОС является POSIX-совместимая ОС реального времени QNX (?).

На практике обычно применяются те или иные вариации на основе микроядра.

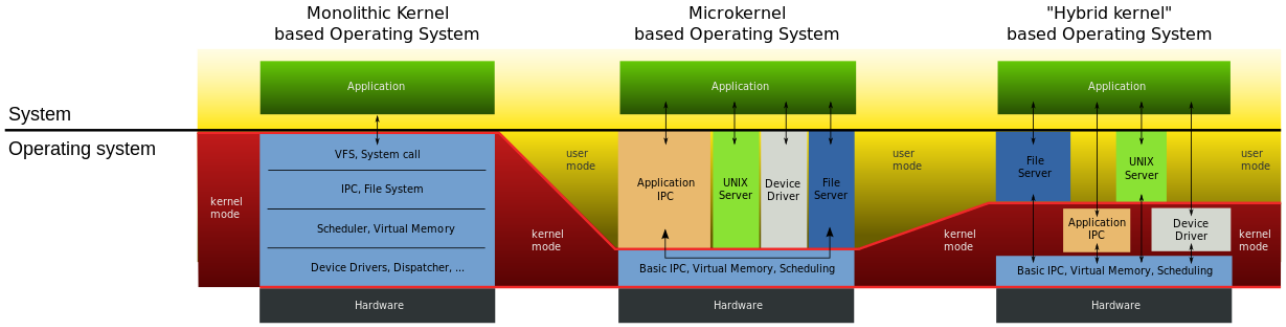
Экзоядро

Экзоядро представляет собой крайний случай микроядра, когда экзоядро занимается только взаимодействием процессов. Все функции работы с памятью, сетью, устройствами при этом вынесены в пользовательские библиотеки (libOS)

Наноядро

Наноядро – это разновидность ядра, реализующего только обработку прерываний. Его концепция близка к Hardware Abstraction Layer (HAL, Слой аппаратных абстракций). Такое ядро часто используется для виртуализации. Наноядро гостевой операционной системы может работать как процесс или модуль ядра другой операционной системы, позволяя запускать «неродные» приложения.

Гибридное ядро



Сравнение архитектуры

Гибридное ядро позволяет сгладить недостатки монолитного ядра и микроядра. При этом часть критичных для производительности функций реализована внутри пространства ядра, а часть, как сервисы, в пользовательском пространстве.

Как примеры гибридных можно привести ядра, основанные на проекте Mach:

- проект MkLinux (предшественник Mac OS X) – ядро Linux запускалось поверх ядра Mach 3.0;
- проект (до сих пор не завершённый) GNU/Hurd для проекта GNU;
- XNU – гибридное ядро на базе микроядра Mach версии 2.5, компонентов от 4.3BSD и объектно-ориентированного интерфейса драйверов Driver Kit, разработанное для NextStep и в дальнейшем для Mac OS X.

Также отметим следующие примеры:

- RTLinux – микроядро, которое выполняет Linux как вытесняемый процесс (операционная система реального времени);
- MinWin – гибридное ядро, применяемое в современных Windows; состоит из NT-ядра и ряда компонентов пользовательского режима и драйверов.

Применение гибридного ядра приводит к тому, что, например, крах драйвера NVidia не вызывает перезагрузку драйвера операционной системой. С другой стороны, существует много других отказов, приводящих к появлению Blue Screen of Death (BSOD) – своего рода визитной карточки Windows.

Что-то пошло не так...

Архитектура ядра Linux

Цифры и факты

- монолитное (модульное) ядро;
- около 30 тыс. файлов;
- около 8 млн. строк кода, не считая комментариев;
- репозиторий занимает около 1 Гб;
- linux-2.6.33.tar.bz2: 63 Mb;
- patch-2.6.33.bz2: 10Mb, около 1.7 млн изменённых строк;
- около 6000 человек, чей код есть в ядре.
- Ядро создал в 1991 году студент университета Хельсинки Линус Торвальдс.
- В качестве платформы он использовал ОС Minix, запущенную на персональном компьютере с процессором Intel 80386.
- В качестве примера для подражания Линус Торвальдс использовал ОС семейства Unix, а в качестве путеводителя – сначала стандарт POSIX, а затем просто исходные коды программ из комплекта GNU (bash, gcc и пр).
- Для работы над ядром Линус Торвальдс также создал git.

Системные вызовы

Системные вызовы возможны как через вызов прерывания int (и выходом из него iret), так и с помощью новых инструкций sysenter (32 бита) и syscall (64 бита) и функции выхода из обработчика sysexit.

Системные вызовы предоставляют интерфейс, используемый прикладными программами – это API ядра. Большинство системных вызовов Linux взяты из стандарта POSIX, но есть и специфические для Linux системные вызовы.

Отметим различие между архитектурой системных вызовов Linux и UNIX с одной стороны и Windows - с другой. Дизайнеры Unix предпочитают предоставить десять системных вызовов с одним параметром вместо одного системного вызова с двадцатью параметрами. Классический пример – создание процесса. В Windows функция для создания процесса – CreateProcess() – принимает 10 аргументов, из которых 5 – структуры. В противоположность этому Unix-системы предоставляют два системных вызова (fork() и exec()), при этом первый – без параметров, второй – с тремя параметрами.

Управление памятью

В Linux используется плоская (flat) модель памяти, когда код и данные используют одно и то же адресное пространство.

Ядро Linux использует в качестве минимальной единицы памяти страницу. Размер страницы может зависеть от оборудования: на x86 это 4Кб. Для хранения информации о странице физической памяти (её физический адрес, принадлежность, режим использования и пр) используется специальная структура page размером в 40 байт.

Ядро использует возможности современных процессоров для организации виртуальной памяти. Благодаря манипуляциям с каталогами страниц виртуальной памяти каждый процесс получает адресное пространство размером в 4Гб (на 32х-разрядных архитектурах). Часть этого пространства доступна процессу только на чтение или исполнение: туда отображаются интерфейсы ядра.

Важно, что процесс, работающий в пространстве пользователя, в большинстве случаев «не знает», где находятся его данные: в ОЗУ или в разделе подкачки (swap). Процесс может запросить у системы выделить ему память именно в ОЗУ, но система не обязана удовлетворять такую просьбу.

Управление процессами

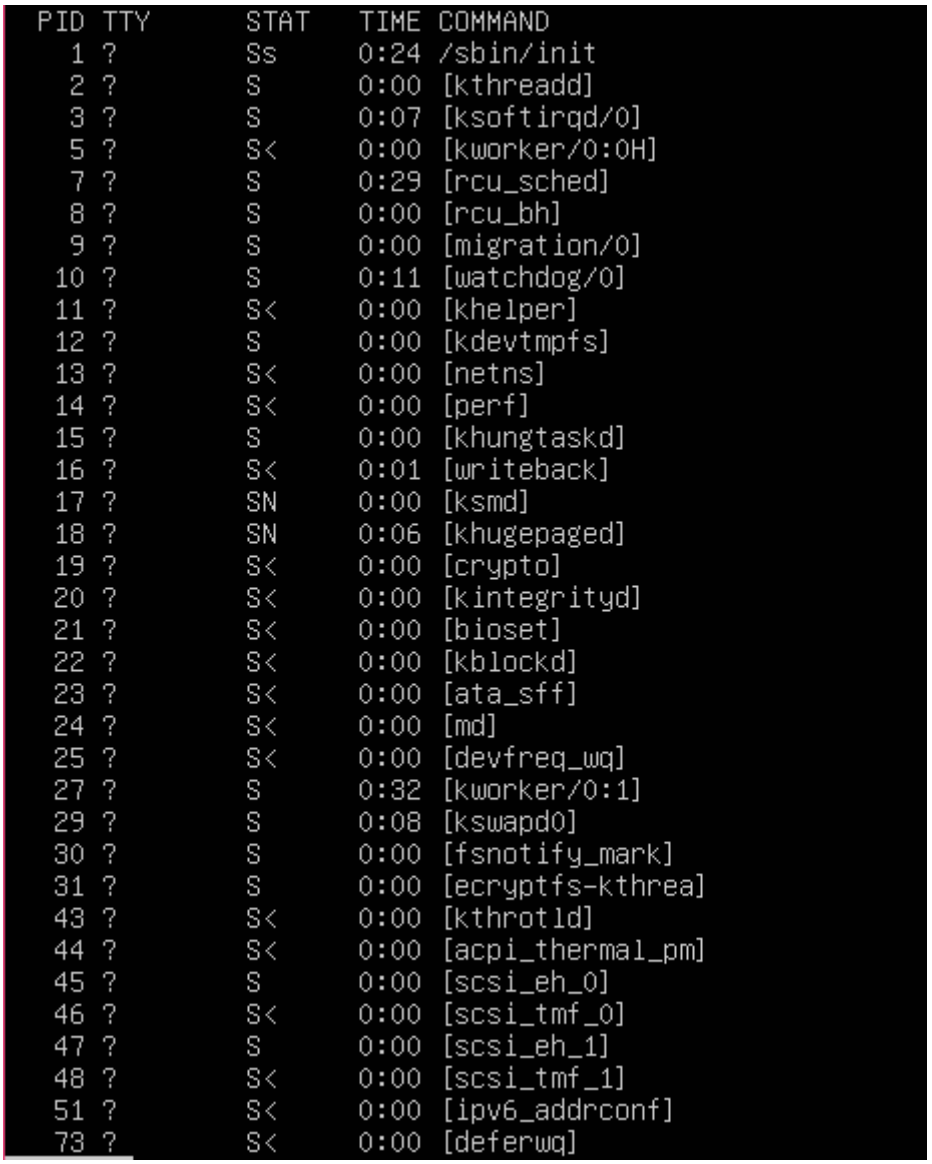
Ядро Linux изначально имело поддержку вытесняющей многозадачности. В реализации вытесняющей многозадачности ядро выделяет каждому процессу определённый квант процессорного времени и «насильно» передаёт управление другому процессу по истечении этого кванта. С одной стороны, это создаёт накладные расходы на переключение режимов процессора и расчёт приоритетов, с другой – повышает надежность и производительность.

Переключение процессов в linux может производиться по наступлении двух событий: аппаратного прерывания или прерывания от таймера. Частота прерываний таймера устанавливается при компиляции ядра в диапазоне от 100 Гц до 1000 Гц. Как правило, аппаратные прерывания вызываются при нажатии клавиши на клавиатуре, движении мыши, поступлении сигналов от других устройств. Начиная с версии 2.6.23 появилась возможность собрать ядро, не использующее переключение процессов по таймеру. Это позволяет снизить энергопотребление в режиме простоя компьютера.

Планировщик процессов использует довольно сложный алгоритм, основанный на расчёте приоритетов процессов. Среди процессов выделяются те, что требуют много процессорного времени,

и те, что тратят больше времени на ввод-вывод. На основе этого регулярно пересчитываются приоритеты процессов. Кроме того, в системе используются задаваемые пользователем значения nice для отдельных процессов.

Помимо многозадачности в режиме пользователя, ядро Linux использует многозадачность в режиме ядра: само ядро многопоточно. Эти задачи видны при выводе ps ax: имена процессов отображаются в квадратных скобках.



Вывод команды ps ax. В квадратных скобках – потоки ядра.

Традиционные ядра Unix-систем не были вытесняемыми. Так, если процесс /usr/bin/cat запрашивает системный вызов open() для открытия файла /media/cdrom/file.txt, управление передаётся ядру. Ядро обнаруживает, что файл расположен на CD-диске, и начинает инициализацию привода (раскручивание диска и пр). Это занимает время, в процессе которого управление не возвращается пользовательским процессам, т.к. планировщик неактивен в то время, когда выполняется код ядра. Все пользовательские процессы ждут завершения этого вызова open().

Современное ядро Linux, напротив, полностью вытесняемо. Планировщик отключается лишь на короткие промежутки времени, когда ядро никак нельзя прервать – например, на время инициализации устройств, требующих действий с фиксированными задержками. В любое другое

время поток ядра может быть вытеснен, и тогда управление передаётся другому потоку ядра или пользовательскому процессу.

Сетевая подсистема

Несмотря на то, что сетевая подсистема могла бы выполняться в пространстве пользователя, на практике стек TCP/IP реализуется на уровне ядра. Сделано это для увеличения производительности, так как множественные обращения к ядру из пространства пользователя, неизбежные при формировании и анализе пакетов, требовали значительного числа накладных расходов.

Сетевая подсистема Linux обеспечивает функционал, в который входят:

- абстракция сокетов;
- стеки сетевых протоколов (TCP/IP, UDP/IP, IPX/SPX, AppleTalk и мн. др);
- маршрутизация (routing);
- пакетный фильтр (модуль Netfilter);
- абстракция сетевых интерфейсов.

Несмотря на то, что архитектурно Linux ближе к System V, в которой использовался Transport Layer Interface (TLI), в Linux используется механизм сокетов Беркли из BSD.

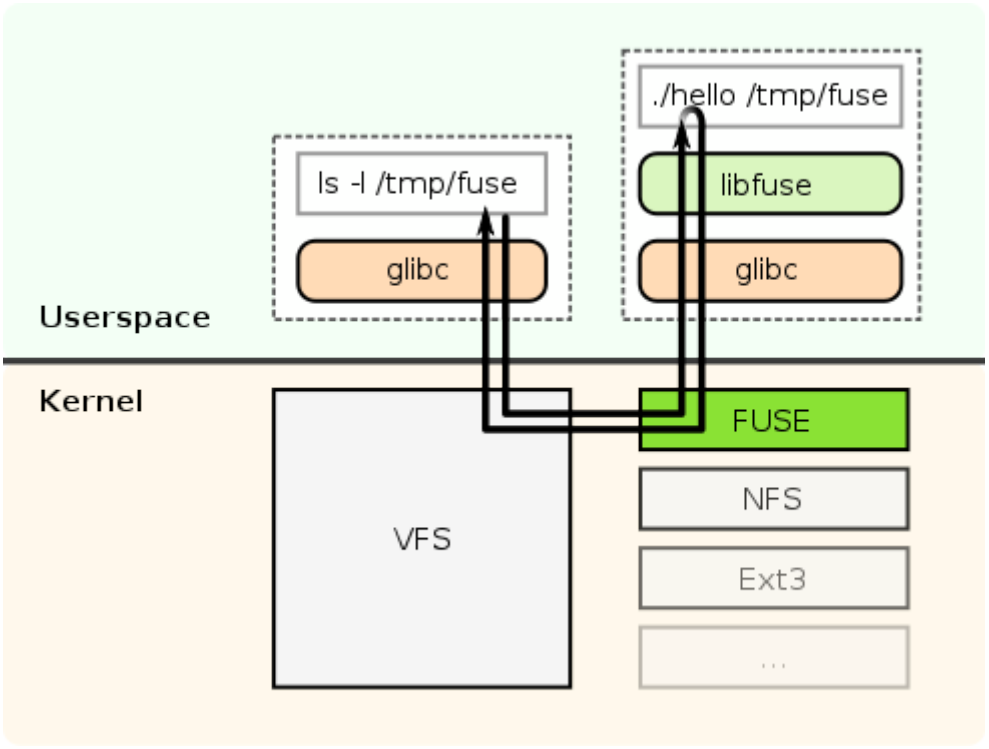
VFS – Виртуальная файловая система

В отличие от Windows, в Unix-подобных ОС существует только одна файловая система, представляющая собой дерево директорий, растущее из «корня». Приложениям в большинстве случаев неважно, на каком носителе расположены данные файлов; они могут находиться на жёстком диске, оптическом диске, флеш-носителе или вообще на другом компьютере, доступном через механизмы сети. Подсистема, реализующая такой уровень абстрактности, называется виртуальной файловой системой (VFS).

В виртуальной файловой системе файловые системы других носителей монтируются в одну из директорий VFS. С другой стороны, благодаря файловой системе proc, смонтированной в /proc, любое приложение или скрипт может получить важные данные от ядра системы. Так, например, команда cat /proc/version позволяет получить информацию о версии ядра Linux в консоли или bash-скрипте.

Драйверы файловых систем

Благодаря архитектуре ядра Linux драйверы файловых систем относятся к более высокому уровню, чем драйверы устройств. Это связано с тем, что драйверы файловых систем не сообщаются ни с какими устройствами: драйвер файловой системы лишь реализует функции, предоставляемые им через интерфейс VFS. При этом данные пишутся и читаются в/из страницы памяти; какие из них и когда будут записаны на носитель – решает более низкий уровень. Благодаря этому был разработан драйвер FUSE (filesystem in userspace – «файловая система в пользовательском пространстве»), который делегирует функциональность драйвера файловой системы в модули, исполняемые в пространстве пользователя. Существует множество реализаций, позволяющих монтировать к VFS удалённые файловые системы, распределённые файловые системы, архивы и т.д.



Механизм работы модуля FUSE

Автор: This file was made by User:SvenTranslation CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3009564

Страничный кэш

Страничный кэш – это подсистема ядра, которая оперирует страницами виртуальной памяти, организованными в виде базисного дерева (radix tree). Когда происходит чтение данных с носителя, данные читаются в выделяемую в кэше страницу, и страница остается в кэше, а драйвер файловой системы читает из нее данные. Драйвер файловой системы пишет данные в страницы памяти, находящиеся в кэше. При этом эти страницы помечаются как «грязные» (dirty). Специальный поток ядра, pdflush, регулярно обходит кэш и формирует запросы на запись грязных страниц. Записанная на носитель грязная страница вновь помечается как чистая.

Уровень блочного ввода-вывода

Уровень блочного ввода-вывода – это подсистема ядра, которая оперирует очередями (queues), состоящими из структур bio. Каждая такая структура описывает одну операцию ввода-вывода (условно говоря, запрос вида «записать эти данные в блоки ##141-142 устройства /dev/hda1»). Для каждого процесса, осуществляющего ввод-вывод, формируется своя очередь. Из этого множества очередей создаётся одна очередь запросов к драйверу каждого устройства.

Планировщик ввода-вывода

Если выполнять запросы на дисковый ввод-вывод от приложений в том порядке, в котором они поступают, производительность системы в среднем будет очень низкой. Это связано с тем, что операция поиска нужного сектора на жёстком диске – очень медленная. Поэтому планировщик обрабатывает очереди запросов, выполняя две операции:

- сортировка: планировщик старается ставить подряд запросы, обращающиеся к находящимся близко секторам диска;

- объединение: если в результате сортировки рядом оказались несколько запросов, обращающихся к последовательно расположенным секторам, их нужно объединить в один запрос.

В современном ядре доступно несколько планировщиков: Anticipatory, Deadline, CFQ, noop. Существует версия ядра от Con Kolivas с ещё одним планировщиком — BFQ. Планировщики могут выбираться при компиляции ядра либо при его запуске.

Отметим планировщик noop: он не выполняет ни сортировку, ни слияние запросов, а перенаправляет запросы драйверам устройств в порядке поступления. На системах с обычными жёсткими дисками этот планировщик показал бы очень плохую производительность. Однако сейчас становятся распространены системы, в которых вместо жёстких дисков используются флэш-носители. Для таких носителей время поиска сектора равно нулю, поэтому операции сортировки и слияния не нужны. В таких системах при использовании планировщика noop производительность не меняется, а потребление ресурсов несколько снижается.

Обработка прерываний

Практически все актуальные архитектуры оборудования используют для общения устройств с программным обеспечением концепцию прерываний. Выглядит это следующим образом: процессор исполняет некоторый процесс (неважно, поток ядра или пользовательский процесс), в ходе чего происходит прерывание от устройства. Процессор отвлекается от выполняемых задач и переключает управление на адрес памяти, сопоставленный данному номеру прерывания в специальной таблице прерываний. По этому адресу находится обработчик прерывания. Обработчик выполняет необходимые действия в зависимости от того, что именно произошло с этим устройством, затем управление передаётся планировщику процессов, который, в свою очередь, решает, кому передать управление дальше.

Во время работы обработчика прерывания планировщик задач неактивен. Это неудивительно: предполагается, что обработчик прерывания работает непосредственно с устройством, а устройство может требовать выполнения каких-то действий в жестких временных рамках. Поэтому, если обработчик прерывания будет работать долго, все остальные процессы и потоки ядра будут ждать, а это обычно недопустимо.

В ядре Linux в результате любого аппаратного прерывания управление передается в функцию do_IRQ(). Эта функция использует отдельную таблицу зарегистрированных в ядре обработчиков прерываний, чтобы определить, куда передавать управление дальше.

Чтобы обеспечить минимальное время работы в контексте прерывания, в ядре Linux используется разделение обработчиков на верхние и нижние половины. Верхняя половина – это функция, которая регистрируется драйвером устройства в качестве обработчика определённого прерывания. Она выполняет только ту работу, которая должна быть выполнена немедленно. Затем она регистрирует другую функцию (свою нижнюю половину) и возвращает управление. Планировщик задач передаст управление зарегистрированной верхней половине, как только это будет возможно. При этом в большинстве случаев управление передаётся нижней половине сразу после завершения работы верхней половины. При этом нижняя половина работает как обычный поток ядра и может быть прервана в любой момент, а потому она имеет право исполняться сколь угодно долго.

В качестве примера рассмотрим обработчик прерывания от сетевой карты, сообщающего, что принят ethernet-пакет. Обработчик обязан выполнить два действия:

- взять пакет из буфера сетевой карты и сигнализировать ей, что пакет получен операционной системой – это нужно сделать немедленно по получении прерывания, поскольку через миллисекунду в буфере будут уже совсем другие данные;

- поместить этот пакет в какие-либо структуры ядра, выяснить, к какому протоколу он относится, и передать его в соответствующие функции обработки – это нужно сделать как можно быстрее, чтобы обеспечить максимальную производительность сетевой подсистемы, но не обязательно немедленно.

Первое действие выполняет верхняя половина обработчика, а второе – нижняя.

Драйверы устройств

Большинство драйверов устройств обычно компилируются в виде модулей ядра. Драйвер устройства получает запросы с двух сторон:

- от устройства – через зарегистрированные драйвером обработчики прерываний;
- от различных частей ядра – через API, который определяется конкретной подсистемой ядра и самим драйвером.

Практика

Простейший модуль ядра для Linux – написание

«Hello, world!»

Напишем на Си простейший модуль, который будет выводить сообщение при загрузке и выгрузке модуля.

hello-1.c

```
/*
 *  hello-1.c - Простейший модуль ядра.
 */
#include <linux/module.h>       /* Необходим для любого модуля ядра */
#include <linux/kernel.h>        /* Здесь находится определение KERN_ALERT */
int init_module(void)
{
        printk("<1>Hello world 1.\n");
  /*
   * Если вернуть ненулевое значение, это будет воспринято как признак ошибки,
   * возникшей в процессе работы init_module; в результате модуль не будет
загружен.
   */
        return 0;
}
void cleanup_module(void)
{
        printk(KERN_ALERT "Goodbye world 1.\n");
}
```

Модуль ядра должен иметь по меньшей мере хотя бы две функции: функцию инициализации модуля – в данном случае init_module(), которую вызывает insmod во время загрузки модуля, – и функцию завершения работы модуля – в данном случае cleanup_module(), которую вызывает rmmod.

Функция init_module(), как правило, выполняет регистрацию обработчика какого-либо события или замещает какую-либо функцию в ядре своим кодом, который, выполнив некие специфические действия, вызывает оригинальную версию функции в ядре.

Функция cleanup_module(), напротив, выполняет отмену всех изменений, сделанных в init_module(), что делает выгрузку модуля безопасной.

Любой модуль ядра должен подключать заголовочный файл linux/module.h.

Также мы подключали ещё один файл – linux/kernel.h, но лишь для того, чтобы получить доступ к определению KERN_ALERT.

Функция printk()

Разработка модулей ядра обладает особенностями: помимо стандарта оформления функций ядра приходится использовать определённые способы взаимодействия с пользователем. Так, здесь не получится использовать printf (хотя можно сделать комбинацию sprintf и printk), а printk() предназначен для регистрации событий и предупреждений – по существу, для ведения лога. Тем не менее, именно эту функцию мы будем использовать для печати на экран.

Соответствуя назначению логирования, каждый вызов printk() сопровождается указанием приоритета: в нашем примере это <1> и KERN_ALERT. Всего в ядре определено 8 различных уровней приоритета для функции printk(), и каждый из них имеет свое макроопределение – таким образом, нет необходимости писать числа, лишённые смысла: имена уровней приоритета и их числовые значения находятся в файле linux/kernel.h. Если уровень приоритета не указывается, по умолчанию он принимается равным DEFAULT_MESSAGE_LOGLEVEL.

Выше упомянуты два способа задания приоритета – числом и именем. На практике считается дурным тоном указание уровней приоритета числовым значением – например, так: <4>. Для этих целей лучше пользоваться именами макроопределений, например: KERN_WARNING.

Если задан уровень ниже, чем int console_loglevel, сообщение выводится на экран. Если запущены и syslog, и klogd, сообщение попадёт также и в системный журнал /var/log/messages – при этом оно может быть выведено на экран, а может и не выводиться. Мы использовали достаточно высокий уровень приоритета KERN_ALERT, чтобы гарантировать вывод сообщения на экран функцией printk().

Компиляция и запуск

Makefile:

```
obj-m += hello-1.o
```

Компиляция:

```
# make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

Запускаем:

```
# insmod ./hello-1.ko
```

Проверяем, что модуль загружен:

```
# cat /proc/modules
```

Выгружаем:

```
# rmmod hello-1
```

Практическое задание

1. Разобрать код загрузчика MS DOS (см. первый источник литературы).
2. Разобрать с помощью hiew точечный файл (из вашей операционной системы или найденный в Интернете). Расшифровать пиксельную матрицу какого-нибудь символа по аналогии с тем, как делали мы, но в обратном направлении (подсказка: переводите шестнадцатеричные числа в двоичные). Можно использовать 2-й источник литературы.
3. Реализовать простейший модуль ядра, проверить его запуск и остановку (использовать одну из вариаций из литературы 8, 9).
4. * Реализовать драйвер символьного устройства (из источников 8, 9), изучить работу модуля ядра.
5. * Реализовать любой учебный пример (из источников 8, 9), изучить работу модуля ядра.

Примечание. Задания со звёздочкой предназначены для тех, кому недостаточно заданий 1-3 и требуются более сложные задачи.

Дополнительные материалы

1. Код загрузчика MSDOS 6.22 http://www.tburke.net/info/ntldr/bootsect.txt
2. Драйвер KeyRus на мемориальной странице Дмитрия Гуртяка http://gurtjak.skif.net/pages/programs.htm
3. Кольца, уровни защиты https://habrahabr.ru/company/smart_soft/blog/184174/
4. Работаем с модулями ядра в Linux https://habrahabr.ru/post/117654/
5. Интервью с Эндрю Таненбаумом https://geektimes.ru/post/36936/
6. Еще интервью с Эндрю Таненбаумом http://www.ylsoftware.com/news/222
7. Представление процессов в ядре Linux http://www.net4me.net/docs/pdf/Linux/ps.pdf
8. Peter Jay Salzman, Michael Burian, Ori Pomerantz The Linux Kernel Module Programming Guide http://www.linuxcenter.ru/lib/books/lkmpg.phtml
9. Олег Цирюлик Модули Linux ядра http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/KERN-modul-4.95.pdf

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Пример Hello world на ассемблере для DOS (COM, EXE) и Windows http://learnprogramm.ucoz.ru/index/hello_world_na_assemblere_dlja_dos_com_exe_i_windows/0-101
2. Устройство драйверов в MS DOS http://www.xserver.ru/computer/os/msdos/1/6.shtml
3. Процесс загрузки драйверов DOS http://www.frolov-lib.ru/books/bsp/v18/ch6_2.html

4. Драйвер KeyRus на мемориальной странице Дмитрия Гуртяка http://gurtjak.skif.net/pages/programs.htm
5. Шрифты chr от Borland в Turbo Pascal http://www.hardline.ru/selfteachers/Info/Programming/!TurboPascal/gl14/gl14_9.html
6. Шрифты в графическом режиме Free Pascal http://src-code.net/teksty-na-graficheskom-ekrane-free-pascal/
7. Управление семисегментным индикатором http://cxem.net/beginner/beginner131.php
8. https://ru.wikipedia.org/wiki/80386
9. https://ru.wikipedia.org/wiki/MS-DOS
10. https://ru.wikipedia.org/wiki/Защищенный_режим
11. Кольца защиты https://habrahabr.ru/company/smart_soft/blog/184174/
12. https://en.wikipedia.org/wiki/MinWin
13. Ядро Linux за десять минут http://iportnov.blogspot.ru/2010/03/linux-10.html?m=1