### 2.1. Код первоначальной загрузки boot

|  |
| --- |
|  |
|  | MBOOT\_PAGE\_ALIGN equ 1<<0 ; Load kernel and modules on a page boundary |
|  | MBOOT\_HEADER\_MAGIC equ 0x1BADB002 ; Сигнатура формата Multiboot |
|  | ; NOTE: We do not use MBOOT\_AOUT\_KLUDGE. It means that GRUB does not |
|  | ; pass us a symbol table. |
|  | MBOOT\_HEADER\_FLAGS equ MBOOT\_PAGE\_ALIGN | MBOOT\_MEM\_INFO ;Флаги, которые содержат |
|  | ; дополнительные требования к загрузке ядра и параметрам, передаваемым загрузчиком ядру |
|  | MBOOT\_CHECKSUM equ -(MBOOT\_HEADER\_MAGIC + MBOOT\_HEADER\_FLAGS) ;Контрольная сумма |
|  |  |
|  |  |
|  | [BITS 32] ; All instructions should be 32-bit. |
|  |  |
|  | [GLOBAL mboot] ; Делаем 'mboot' доступным из Си. |
|  | [EXTERN code] ; Start of the '.text' section. |
|  | [EXTERN bss] ; Start of the .bss section. |
|  | [EXTERN end] ; End of the last loadable section. |
|  |  |
|  | mboot: |
|  | dd MBOOT\_HEADER\_MAGIC ; GRUB will search for this value on each |
|  | ; 4-byte boundary in your kernel file |
|  | dd MBOOT\_HEADER\_FLAGS ; How GRUB should load your file / settings |
|  | dd MBOOT\_CHECKSUM ; To ensure that the above values are correct |
|  |  |
|  | dd mboot ; Location of this descriptor |
|  | dd code ; Start of kernel '.text' (code) section. |
|  | dd bss ; End of kernel '.data' section. |
|  | dd end ; End of kernel. |
|  | dd start ; Kernel entry point (initial EIP). |
|  |  |
|  | [GLOBAL start] ; Точка входа ядра (делаем видимым для линковщика) |
|  | [EXTERN main] ; Точка входа нашего Си кода |
|  |  |
|  |  |
|  | ;Grub через регистры %ebx и %eax передает указатель |
|  | ; на структуру multiboot Information и значение 0x1BADB002 |
|  | ; соответственно. Структура multiboot Information содержит |
|  | ; различную информацию, в том числе список загруженных модулей |
|  | ; и их расположение, что может понадобиться для дальнейшей загрузки системы. |
|  |  |
|  | start: |
|  | ; Загрузим multiboot информацию в стек, она будет первым параметром функции main на Си |
|  | push ebx |
|  |  |
|  | ; Запускаем ядро: |
|  | cli ; Отключаем прерывания |
|  | call main ; Вызываем нашу функцию main() |
|  | jmp $ ; Войдем в бесконечный цикл, чтобы остановить |
|  | ; выполнением процессора мусора в памяти |
|  | ; после нашего ядра! |

В модуле несколько строк кода:

push ebx

cli

call main

jmp $

Все остальное относится к заголовку мультзагрузки multiboot.

#### 2.2.1. Мультизагрузка

Мультизагрузка multiboot является стандартом, которому должно соответствовать ядро (предполагается в GRUB). Благодаря этому загрузчик может, во первых, точно знать, какая среда нужна ядру, когда происходит загрузка, во вторых, разрешить ядру запрашивать ту среду, в которой оно должно находиться.

Для того, чтобы ваше ядро было совместимо со стандартом multiboot, вам нужно где-нибудь в вашем ядре добавить структуру с заголовком (на самом деле, заголовок должен быть в первых 4КБ ядра). Удобно, что в NASM есть команда 'dd', которая позволит вам вставлять в код конкретные константы.

**MBOOT\_HEADER\_MAGIC** – число, которое указывает, что ядро совместимо со стандартом multiboot.

**MBOOT\_HEADER\_FLAGS** - поле флагов. Мы просим GRUB выполнить выравнивание секций по границам страниц (MBOOT\_PAGE\_ALIGN), а также предоставить нам некоторую информацию о памяти (MBOOT\_MEM\_INFO).

**MBOOT\_CHECKSUM** - это поле предназначено для проверки на наличие ошибок.

**mboot** - адрес структуры, в которую осуществляется запись. GRUB использует этот адрес для того, чтобы сообщить нам, куда была перемещена эта структура.

**code, bss, end, start** - эти символы определяются в компоновщике. Мы их используем для того, чтобы сообщить GRUB, где могут находиться различные секции нашего ядра.

### **Взаимодействие с Си**

 В GCC для архитектуры x86 при организации вызова используется следующее соглашение \_\_cdecl:

* Все параметры в функцию передаются через стек
* Параметры, перемещаемые в стек, перечисляются справа налево.
* Значение, возвращаемое функцией, помещается в EAX.

|  |
| --- |
| // main.c -- Определяет точку входа Cи-кода ядра |
|  |  |
|  | #include "monitor.h" |
|  | #include "multiboot.h" |
|  | #include "descriptor\_tables.h" |
|  | #include "timer.h" |
|  |  |
|  |  |
|  | int main(struct multiboot \*mboot\_ptr) |
|  | { |
|  | // Initialise all the ISRs and segmentation |
|  | init\_descriptor\_tables(); |
|  | // Initialise the screen (by clearing it) |
|  | monitor\_clear(); |
|  | // Write out a sample string |
|  | /\*for (size\_t i = 0; i < 16; i++) { |
|  | monitor\_color\_write("Hello, world!\n", i, 15); |
|  | }\*/ |
|  | monitor\_color\_write("MiniOS\n", 4, 15); |
|  | monitor\_write("\nDeveloped by Nikita Saharin, Pavel Sushko, Aleksandr Yakimovich\n\n"); |
|  |  |
|  | asm volatile("int $0x3"); |
|  | asm volatile("int $0x4"); |
|  |  |
|  | asm volatile("sti"); |
|  | init\_timer(10); |
|  |  |
|  | return 0; |
|  | } |

### 2.4. Компилируем, компонуем и запускаем!

Редактируем makefile:

SOURCES=boot.o main.o

CFLAGS=-nostdlib -nostdinc -fno-builtin -fno-stack-protector

Мы должны остановить компилятор GCC, когда он попытается скомпоновать наше ядро с своими библиотеками C для Linux - это ядро вообще (пока) не будет работать . Для этого используются флаги CFLAGS.

Компилируем, компонуем и запускаем ядро:

cd src

make clean # Здесь все ошибки игнорируются.

make

cd ..

./update\_image.sh

./run\_bochs.sh # Здесь от вас может быть запрошен пароль.

В результате должна начаться загрузка эмулятора bochs, вы увидите на несколько секунд загрузчик GRUB, затем будет работать ядро.

## 3. Экран

Ваше ядро с помощью GRUB загружается в текстовом режиме. То есть, для него выделяется фреймбуфер или кадровый буфер (область памяти), с помощью которого осуществляется управление экраном символов.

Область памяти фреймбуфер, доступна точно также, как и обычная оперативная память, расположенная по адресу 0xB8000. Однако важно заметить, что фрейбуфер фактически не является обычной оперативной памятью. Это часть специализированной видеопамяти контроллера, которая с помощью аппаратных средств отображается в ваше линейное адресное пространство.

Фреймбуфер является всего лишь простым массивом 16-битных слов, каждое 16-битное значение представляет собой отображение одного символа. Смещение слова от начала фреймбуфера, в котором находится символ, соответствующий позиции х, у на экране, рассчитывается следующим образом:

(y \* 80 + x) \* 2

Важно отметить, что множитель '\* 2' указан только потому, что каждый элемент занимает 2 байта (16 бит). Например, если вы индексируете массив 16-битных значений, то индексом будет просто у \* 80 + x.

http://rus-linux.net/MyLDP/kernel/toyos/the_screen_word_format.png

Формат слова

В кодировке ASCII (в текстовом режиме кодировка Unicode не поддерживается) для представления символов используются 8 бит. У нас остается еще 8 бит, которые не используются. Аппаратура VGA использует их для назначения цвета тексту и цвета фону (по 4 бит каждый). На рисунке показано деление на части этого 16-разрядного значения.

4 бита для цветового кода дает нам 15 возможных цветов, которые мы можем отобразить::

0:черный, 1:синий, 2:зеленый, 3:циан, 4:красный, 5:мажента, 6:коричневый, 7:светло серый, 8:темно серый, 9:светло синий, 10:светло зеленый, 11:светлый циан, 12:светло красный, 13:светлый мажента, 14: светло коричневый, 15: белый.

У контроллера VGA также есть несколько портов главной шины ввода/вывода, которые можно использовать для того, чтобы отправить контроллеру конкретные указания. (Среди прочего) есть регистр управления с адресом 0x3D4 и регистр данных с адресом 0x3D5. Мы будем использовать эти порты для того, чтобы передать в контроллер указание обновить позицию курсора.

#### Теория в действии:

Во-первых, нам нужно еще несколько обычно используемых глобальных функций. В файлах common.c и common.h есть функции записи данных в шину ввода/вывода и чтения их из шины, а также некоторые определения типов, что позволит вам проще писать переносимый код.

// common.h

|  |
| --- |
| #ifndef COMMON\_H |
|  | #define COMMON\_H |
|  |  |
|  | // Some nice typedefs, to standardise sizes across platforms. |
|  | // These typedefs are written for 32-bit X86. |
|  | typedef unsigned int u32int; |
|  | typedef int s32int; |
|  | typedef unsigned short u16int; |
|  | typedef short s16int; |
|  | typedef unsigned char u8int; |
|  | typedef char s8int; |
|  | typedef unsigned int size\_t; |
|  |  |
|  | #define NULL 0; |
|  | #define EOF '\0'; |
|  |  |
|  | void outb(u16int port, u8int value); |
|  | u8int inb(u16int port); |
|  | u16int inw(u16int port); |
|  |  |
|  | /\*// Copy len bytes from src to dest. |
|  | void memcpy(u8int \*dest, const u8int \*src, u32int len); |
|  |  |
|  | // Write len copies of val into dest. |
|  | void memset(u8int \*dest, u8int val, u32int len); |
|  |  |
|  | // Compare two strings. Should return -1 if |
|  | // str1 < str2, 0 if they are equal or 1 otherwise. |
|  | int strcmp(char \*str1, char \*str2); |
|  |  |
|  | // Copy the NULL-terminated string src into dest, and |
|  | // return dest. |
|  | char \*strcpy(char \*dest, const char \*src); |
|  |  |
|  | // Concatenate the NULL-terminated string src onto |
|  | // the end of dest, and return dest. |
|  | char \*strcat(char \*dest, const char \*src); |
|  |  |
|  | int strlen(char \*src);\*/ |
|  |  |
|  | // REPLACED TO PANIC.H |
|  | /\* |
|  | #define PANIC(msg) panic(msg, \_\_FILE\_\_, \_\_LINE\_\_); |
|  | #define ASSERT(b) ((b) ? (void)0 : panic\_assert(\_\_FILE\_\_, \_\_LINE\_\_, #b)) |
|  |  |
|  | extern void panic(const char \*message, const char \*file, u32int line); |
|  | extern void panic\_assert(const char \*file, u32int line, const char \*desc); |
|  | \*/ |
|  |  |
|  | #define \_\_np\_anyptrlt(ptr1, ptr2)\ |
|  | ((ptr1) < (ptr2)) |
|  |  |
|  | void \*memchr(const void \*s, int c, register size\_t n); |
|  |  |
|  | int memcmp(const void \*s1, const void \*s2, register size\_t n); |
|  |  |
|  | void \*memcpy(void \* restrict s1, const void \* restrict s2, register size\_t n); |
|  |  |
|  | void \*memmove(void \*s1, const void \*s2, register size\_t n); |
|  |  |
|  | void \*memset(void \*s, register int c, register size\_t n); |
|  |  |
|  | char \*strcat(register char \* restrict s1, register const char \* restrict s2); |
|  |  |
|  | char \*strchr(register const char \*s, register int c); |
|  |  |
|  | int strcmp(register const char \*s1, register const char \*s2); |
|  |  |
|  | char \*strcpy(char \* restrict s1, const char \* restrict s2); |
|  |  |
|  | size\_t strcspn(register const char \*s1, register const char \*s2); |
|  |  |
|  | size\_t strlen(register const char \*s); |
|  |  |
|  | char \*strncat(register char \* restrict s1, register const char \* restrict s2, register size\_t n); |
|  |  |
|  | int strncmp(register const char \*s1, register const char \*s2, register size\_t n); |
|  |  |
|  | char \*strncpy(register char \* restrict s1, register const char \* restrict s2, register size\_t n); |
|  |  |
|  | char \*strpbrk(register const char \*s1, register const char \*s2); |
|  |  |
|  | char \*strrchr(register const char \*s, register int c); |
|  |  |
|  | size\_t strspn(register const char \*s1, register const char \*s2); |
|  |  |
|  | char \*strstr(register const char \*s1, register const char \*s2); //nikitasa1997 => Ok |
|  |  |
|  |  |
|  | #endif // COMMON\_H |

// common.c

|  |
| --- |
| #include "common.h" |
|  |  |
|  | // Write a byte out to the specified port. |
|  | void outb(u16int port, u8int value) |
|  | { |
|  | asm volatile ("outb %1, %0" : : "dN" (port), "a" (value)); |
|  | } |
|  |  |
|  | u8int inb(u16int port) |
|  | { |
|  | u8int ret; |
|  | asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port)); |
|  | return ret; |
|  | } |
|  |  |
|  | u16int inw(u16int port) |
|  | { |
|  | u16int ret; |
|  | asm volatile ("inw %1, %0" : "=a" (ret) : "dN" (port)); |
|  | return ret; |
|  | } |
|  |  |
|  | // Copy len bytes from src to dest. |
|  | /\*void memcpy(u8int \*dest, const u8int \*src, u32int len) |
|  | { |
|  | const u8int \*sp = (const u8int \*)src; |
|  | u8int \*dp = (u8int \*)dest; |
|  | for(; len != 0; len--) \*dp++ = \*sp++; |
|  | } |
|  |  |
|  | // Write len copies of val into dest. |
|  | void memset(u8int \*dest, u8int val, u32int len) |
|  | { |
|  | u8int \*temp = (u8int \*)dest; |
|  | for ( ; len != 0; len--) \*temp++ = val; |
|  | } |
|  |  |
|  | // Compare two strings. Should return -1 if |
|  | // str1 < str2, 0 if they are equal or 1 otherwise. |
|  | int strcmp(char \*str1, char \*str2) |
|  | { |
|  | int i = 0; |
|  | int failed = 0; |
|  | while(str1[i] != '\0' && str2[i] != '\0') |
|  | { |
|  | if(str1[i] != str2[i]) |
|  | { |
|  | failed = 1; |
|  | break; |
|  | } |
|  | i++; |
|  | } |
|  | // why did the loop exit? |
|  | if( (str1[i] == '\0' && str2[i] != '\0') || (str1[i] != '\0' && str2[i] == '\0') ) |
|  | failed = 1; |
|  |  |
|  | return failed; |
|  | } |
|  |  |
|  | // Copy the NULL-terminated string src into dest, and |
|  | // return dest. |
|  | char \*strcpy(char \*dest, const char \*src) |
|  | { |
|  | do |
|  | { |
|  | \*dest++ = \*src++; |
|  | } |
|  | while (\*src != 0); |
|  | } |
|  |  |
|  | // Concatenate the NULL-terminated string src onto |
|  | // the end of dest, and return dest. |
|  | char \*strcat(char \*dest, const char \*src) |
|  | { |
|  | while (\*dest != 0) |
|  | { |
|  | \*dest = \*dest++; |
|  | } |
|  |  |
|  | do |
|  | { |
|  | \*dest++ = \*src++; |
|  | } |
|  | while (\*src != 0); |
|  | return dest; |
|  | } |
|  |  |
|  | int strlen(char \*src) |
|  | { |
|  | int i = 0; |
|  | while (\*src++) |
|  | i++; |
|  | return i; |
|  | }\*/ |
|  |  |
|  | //REPLACED TO PANIC.C |
|  | /\* |
|  | extern void panic(const char \*message, const char \*file, u32int line) |
|  | { |
|  | // We encountered a massive problem and have to stop. |
|  | asm volatile("cli"); // Disable interrupts. |
|  |  |
|  | monitor\_write("PANIC("); |
|  | monitor\_write(message); |
|  | monitor\_write(") at "); |
|  | monitor\_write(file); |
|  | monitor\_write(":"); |
|  | monitor\_write\_dec(line); |
|  | monitor\_write("\n"); |
|  | // Halt by going into an infinite loop. |
|  | for(;;); |
|  | } |
|  |  |
|  | extern void panic\_assert(const char \*file, u32int line, const char \*desc) |
|  | { |
|  | // An assertion failed, and we have to panic. |
|  | asm volatile("cli"); // Disable interrupts. |
|  |  |
|  | monitor\_write("ASSERTION-FAILED("); |
|  | monitor\_write(desc); |
|  | monitor\_write(") at "); |
|  | monitor\_write(file); |
|  | monitor\_write(":"); |
|  | monitor\_write\_dec(line); |
|  | monitor\_write("\n"); |
|  | // Halt by going into an infinite loop. |
|  | for(;;); |
|  | } |
|  | \*/ |
|  |  |
|  | /\*Section\*/ |
|  | /\* |
|  | memchr() |
|  | memcmp() |
|  | memcpy() |
|  | memmove() |
|  | memset() |
|  | strcat() |
|  | strchr() |
|  | strcmp() |
|  | strcpy() |
|  | strcspn() |
|  | strlen() |
|  | strncat() |
|  | strncmp() |
|  | strncpy() |
|  | strpbrk() |
|  | strrchr() |
|  | strspn() |
|  | strstr() nikitasa1997 => Ok |
|  | strtok() nikitasa1997 => Ok |
|  | \*/ |
|  |  |
|  | #define \_\_np\_anyptrlt(ptr1, ptr2)\ |
|  | ((ptr1) < (ptr2)) |
|  |  |
|  | void \*memchr(const void \*s, int c, register size\_t n) |
|  | { |
|  | for (register const unsigned char \*ptr = s; n; ++ptr, --n) |
|  | { |
|  | if (\*ptr == (unsigned char) c) |
|  | { |
|  | return (void \*) ptr; |
|  | } |
|  | } |
|  |  |
|  | return NULL; |
|  | } |
|  |  |
|  | int memcmp(const void \*s1, const void \*s2, register size\_t n) |
|  | { |
|  | for (register const unsigned char \*ptr1 = s1, \*ptr2 = s2; n; ++ptr1, ++ptr2, --n) |
|  | { |
|  | if (\*ptr1 != \*ptr2) |
|  | { |
|  | return \*ptr1 - \*ptr2; |
|  | } |
|  | } |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | void \*memcpy(void \* restrict s1, const void \* restrict s2, register size\_t n) |
|  | { |
|  | register const char \*ptr2 = s2; |
|  | for (register char \*ptr1 = s1; n; ++ptr1, ++ptr2, --n) |
|  | { |
|  | \*ptr1 = \*ptr2; |
|  | } |
|  |  |
|  | return s1; |
|  | } |
|  |  |
|  | void \*memmove(void \*s1, const void \*s2, register size\_t n) |
|  | { |
|  | register unsigned char \*ptr1 = s1; |
|  | register const unsigned char \*ptr2 = s2; |
|  | if (\_\_np\_anyptrlt(ptr2, ptr1)) |
|  | { |
|  | for (ptr1 += n, ptr2 += n; --ptr1, --ptr2, n; --n) |
|  | { |
|  | \*ptr1 = \*ptr2; |
|  | } |
|  | } |
|  | else |
|  | { |
|  | for (; n; ++ptr1, ++ptr2, --n) |
|  | { |
|  | \*ptr1 = \*ptr2; |
|  | } |
|  | } |
|  |  |
|  | return s1; |
|  | } |
|  |  |
|  | void \*memset(void \*s, register int c, register size\_t n) |
|  | { |
|  | for (register unsigned char \*ptr = s; n; ++ptr, --n) |
|  | { |
|  | \*ptr = (unsigned char) c; |
|  | } |
|  |  |
|  | return s; |
|  | } |
|  |  |
|  | char \*strcat(register char \* restrict s1, register const char \* restrict s2) |
|  | { |
|  | char \*res = s1; |
|  | for (; \*s1; ++s1); |
|  | for (; (\*s1 = \*s2); ++s1, ++s2); |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | char \*strchr(register const char \*s, register int c) |
|  | { |
|  | for (; \*s != (char) c; ++s) |
|  | { |
|  | if (!\*s) |
|  | { |
|  | return NULL; |
|  | } |
|  | } |
|  |  |
|  | return (char \*) s; |
|  | } |
|  |  |
|  | int strcmp(register const char \*s1, register const char \*s2) |
|  | { |
|  | for (; \*s1 && \*s1 == \*s2; ++s1, ++s2); |
|  |  |
|  | return \*(const unsigned char \*) s1 - \*(const unsigned char \*) s2; |
|  | } |
|  |  |
|  | char \*strcpy(char \* restrict s1, const char \* restrict s2) |
|  | { |
|  | char \*res = s1; |
|  | for (; (\*s1 = \*s2); ++s1, ++s2); |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | size\_t strcspn(register const char \*s1, register const char \*s2) |
|  | { |
|  | size\_t res = 0u; |
|  | for (; \*s1; ++s1, ++res) |
|  | { |
|  | if (strchr(s2, \*s1)) |
|  | { |
|  | return res; |
|  | } |
|  | } |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | size\_t strlen(register const char \*s) |
|  | { |
|  | register size\_t i; |
|  | for (i = 0u; \*s; ++i); |
|  |  |
|  | return i; |
|  | } |
|  |  |
|  | char \*strncat(register char \* restrict s1, register const char \* restrict s2, register size\_t n) |
|  | { |
|  | char \*res = s1; |
|  |  |
|  | for (; \*s1; ++s1); |
|  | for (; \*s2 && n; ++s1, ++s2, --n) |
|  | { |
|  | \*s1 = \*s2; |
|  | } |
|  | \*s1 = '\0'; |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | int strncmp(register const char \*s1, register const char \*s2, register size\_t n) |
|  | { |
|  | for (;; ++s1, ++s2, --n) |
|  | { |
|  | if (!\*s1 || \*s1 != \*s2) |
|  | { |
|  | return \*(unsigned char \*) s1 - \*(unsigned char \*) s2; |
|  | } |
|  | } |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | char \*strncpy(register char \* restrict s1, register const char \* restrict s2, register size\_t n) |
|  | { |
|  | char \*res = s1; |
|  | for (; n; ++s1, ++s2, --n) |
|  | { |
|  | \*s1 = \*s2; |
|  | } |
|  | for (; n; ++s1, --n) |
|  | { |
|  | \*s1 = '\0'; |
|  | } |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | char \*strpbrk(register const char \*s1, register const char \*s2) |
|  | { |
|  | for (; \*s1; ++s1) |
|  | { |
|  | if (strchr(s2, \*s1)) |
|  | { |
|  | return (char \*) s1; |
|  | } |
|  | } |
|  |  |
|  | return NULL; |
|  | } |
|  |  |
|  | char \*strrchr(register const char \*s, register int c) |
|  | { |
|  | char \*res = NULL; |
|  | for (; \*s; ++s) |
|  | { |
|  | if (\*s == (char) c) |
|  | { |
|  | res = (char \*) s; |
|  | } |
|  | } |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | size\_t strspn(register const char \*s1, register const char \*s2) // |
|  | { |
|  | size\_t res = 0u; |
|  | for (; \*s1 && strchr(s2, \*s1); ++s1, ++res); |
|  |  |
|  | return res; |
|  | } |
|  |  |
|  | char \*strstr(register const char \*s1, register const char \*s2) //nikitasa1997 => Ok |
|  | { |
|  | for (; \*s1; ++s1) |
|  | { |
|  | register size\_t i; |
|  | for (i = 0u; s1[i] && s1[i] == s2[i]; ++i); |
|  | if (!s2[i]) |
|  | { |
|  | return (char \*) s1; |
|  | } |
|  | } |
|  |  |
|  | return NULL; |
|  | } |
|  |  |
|  | char \*strtok(register char \* restrict s1, register const char \* restrict s2) //nikitasa1997 => Ok |
|  | { |
|  | static char \*ptr = NULL; |
|  |  |
|  | if (s1) |
|  | { |
|  | ptr = s1; |
|  | } |
|  | else if (!ptr) |
|  | { |
|  | return NULL; |
|  | } |
|  |  |
|  | s1 = ptr + strspn(ptr, s2); |
|  | ptr = s1 + strcspn(s1, s2); |
|  | if (ptr == s1) |
|  | { |
|  | ptr = NULL; |
|  | return NULL; |
|  | } |
|  |  |
|  | if (\*ptr) |
|  | { |
|  | \*ptr = '\0'; |
|  | ++ptr; |
|  | } |
|  | else |
|  | { |
|  | ptr = NULL; |
|  | } |
|  |  |
|  | return s1; |
|  | } |

#### 3.2.2. Код монитора

Пример заголовочного файла:

// monitor.h

|  |
| --- |
| #ifndef MONITOR\_H |
|  | #define MONITOR\_H |
|  |  |
|  | #include "common.h" |
|  |  |
|  | /\* |
|  | ====== |
|  | COLORS |
|  | ====== |
|  | 0 Black |
|  | 1 Blue |
|  | 2 Green |
|  | 3 Cyan |
|  | 4 Red |
|  | 5 Magenta |
|  | 6 Brown |
|  | 7 Light gray |
|  | 8 Dark gray |
|  | 9 Light blue |
|  | 10 Light green |
|  | 11 Light cyan |
|  | 12 Pink |
|  | 13 Light magenta |
|  | 14 Yellow |
|  | 15 White |
|  | \*/ |
|  |  |
|  | // Writes a single character out to the screen. |
|  | void monitor\_color\_put(char c, u8int backColour, u8int foreColour); |
|  | void monitor\_put(char c); |
|  |  |
|  | // Clears the screen, by copying lots of spaces to the framebuffer. |
|  | void monitor\_clear(); |
|  |  |
|  | // Outputs a null-terminated ASCII string to the monitor. |
|  | void monitor\_write(const char \*c); |
|  |  |
|  | void monitor\_write\_hex(u32int n); |
|  |  |
|  | void monitor\_write\_dec(u32int n); |
|  |  |
|  | // Color outputs a null-terminated ASCII string to the monitor. |
|  | void monitor\_color\_write(const char \*c, u8int backColour, u8int foreColour); |
|  |  |
|  | void monitor\_color\_write\_hex(u32int n, u8int backColour, u8int foreColour); |
|  |  |
|  | void monitor\_color\_write\_dec(u32int n, u8int backColour, u8int foreColour); |
|  |  |
|  |  |
|  | #endif // MONITOR\_H |

###### 3.2.2.1. Перемещение курсора

Рассчитываем линейное смещение координат х и у курсора с помощью приведенного выше уравнения, передаем это смещение в контроллер VGA. По ряду причин, он принимает 16-битное значение местоположения курсора, которое должно быть представлено в виде двух байтов. Для того, чтобы сообщить колнтроллеру, что мы посылаем ему старший байт, мы посылаем в порт контроллера (0x3D4) команду 14 , а затем отправляем этот байт в порт 0x3D5. Затем мы все повторяем с младшим байтом, но теперь с командой 15.

// Обновляем аппаратный курсор.

static void move\_cursor()

{

// The screen is 80 characters wide...

u16int cursorLocation = cursor\_y \* 80 + cursor\_x;

outb(0x3D4, 14); // Сообщаем плате VGA о том, что мы посылаем старший байт курсора.

outb(0x3D5, cursorLocation >> 8); // Посылаем старший байт курсора.

outb(0x3D4, 15); // Сообщаем плате VGA о том, что мы посылаем младший байт курсора.

outb(0x3D5, cursorLocation); // Посылаем младший байт курсора.

}

###### 3.2.2.2. Скроллинг экрана

Прокрутка изображения на 1 строку вверх, при заполнении текстом всего экрана.

// Скроллинг текста на экране на одну строку вверх.

static void scroll()

{

// Берем символ пробела с атрибутами цвета, заданными по умолчанию.

u8int attributeByte = (0 /\*black\*/ << 4) | (15 /\*white\*/ & 0x0F);

u16int blank = 0x20 /\* space \*/ | (attributeByte << 8);

// Строка 25 находится внизу, это означает, что нам нужно выполнить скроллинг вверх

if(cursor\_y >= 25)

{

// Перемещаем кусок текста, отображаемого в текущий момент,

// обратно в буфер, сдвинув его на одну строку

int i;

for (i = 0\*80; i < 24\*80; i++)

{

video\_memory[i] = video\_memory[i+80];

}

// Последняя строка должна теперь быть пустой. Это осуществляется

// записью в нее 80 символов пробела.

for (i = 24\*80; i < 25\*80; i++)

{

video\_memory[i] = blank;

}

// Теперь курсор должен находиться на последней строке.

cursor\_y = 24;

}

}

###### 3.2.2.3. Запись символа на экран

Расчет положения курсора.

// Writes a single character out to the screen.

void monitor\_put(char c)

{

// Цвет фона - черный (0), цвет текста - белый (15).

u8int backColour = 0;

u8int foreColour = 15;

// Байт атрибута состоит из двух полубайтов - младший является цветом,

// используемым для отображения текста, а старший - фоновым цветом.

u8int attributeByte = (backColour << 4) | (foreColour & 0x0F);

// Байт атрибута представляет собой 8 старших битов слова, которое мы должны

// послать на плату VGA.

u16int attribute = attributeByte << 8;

u16int \*location;

// Обработка символа backspace - перемещаем курсор на одну позицию обратно

if (c == 0x08 && cursor\_x)

{

cursor\_x--;

}

// Обработка символа табуляции - увеличиваем координату X курсора,

// но так, чтобы она делилась на 8

else if (c == 0x09)

{

cursor\_x = (cursor\_x+8) & ~(8-1);

}

// Обрабатываем возврат каретки

else if (c == '\r')

{

cursor\_x = 0;

}

// Обрабатываем переход на новую строку - перемещаем курсор обратно

// влево и увеличиваем номер строки

else if (c == '\n')

{

cursor\_x = 0;

cursor\_y++;

}

// Обрабатываем другие символы, которые можно выводить на экран.

else if(c >= ' ')

{

location = video\_memory + (cursor\_y\*80 + cursor\_x);

\*location = c | attribute;

cursor\_x++;

}

// Проверяем, нужно ли нам добавлять новую строку из-за того, что

// мы достигли конца экрана.

if (cursor\_x >= 80)

{

cursor\_x = 0;

cursor\_y ++;

}

// Скроллинг экрана, если это необходимо.

scroll();

// Перемещение аппаратного курсора.

move\_cursor();

}

###### 3.2.2.4. Очистка экрана

Заполнение пробелами.

// Очистка экрана - записываем пробелы во фреймбуфер.

void monitor\_clear()

{

// Устанавливаем в байте атрибутов значения цветов, используемые по умолчанию

u8int attributeByte = (0 /\*black\*/ << 4) | (15 /\*white\*/ & 0x0F);

u16int blank = 0x20 /\* space \*/ | (attributeByte << 8);

int i;

for (i = 0; i < 80\*25; i++)

{

video\_memory[i] = blank;

}

// Перемещаем аппаратный курсор в начало.

cursor\_x = 0;

cursor\_y = 0;

move\_cursor();

}

###### 3.2.2.5. Запись строки

// Выдаем на монитор строку ASCII, завершающуюся символом null.

void monitor\_write(char \*c)

{

int i = 0;

while (c[i])

{

monitor\_put(c[i++]);

}

}

## 4. Таблицы GDT и IDT

#### 4.1. Таблица глобальных дескрипторов GDT (теория)

Способы защиты памяти и реализации виртуальной памяти в архитектуре x86

Сегментация

Страничная организация памяти

При сегментации доступ к памяти осуществляется внутри сегмента. То есть адрес памяти добавляется к базовому адресу сегмента и проверяется длина сегмента. Вы можете рассматривать сегмент как окно в адресном пространстве. Процесс не знает, что это окно, все, что он видит - линейное адресное пространство, начиная с нуля и до длины сегмента.

Что касается страничной организации памяти, то адресное пространство делится на блоки (обычно размером 4 Кб) - страницы. Любая страница может быть отображена в физическую память - отображается на то, что называется "фрейм". Либо отображения может не быть. И таким образом, вы можете создавать виртуальные пространства памяти.

У каждого из этих методов есть свои собственные преимущества, но страничная организация памяти намного лучше. Сегментация, хотя все еще используется, быстро устаревает как способ защиты памяти и виртуальной памяти. На самом деле, в архитектуре x86-64 для того, чтобы некоторые ее команды работали должным образом, требуется плоская модель памяти (один сегмент с базой с адресом 0 и границей с адресом 0xFFFFFFFF).

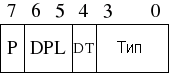
Сегментация встроена непосредственно в архитектуру x86. Ее обойти невозможно. Итак, мы создадим собственную таблицу глобальных дескрипторов - список дескрипторов сегментов.

Мы создаем плоскую модель памяти. Окно сегмента должно начинаться с адреса 0x00000000 и продолжаться до адреса 0xFFFFFFFF (конец памяти). Однако есть одна вещь, которую при сегментации делать можно, а при страничной организации памяти - нет, это задание уровня кольца.

Кольцо определяет уровень привилегий - нулевое кольцо является наиболее привилегированным и еще есть, как минимум, три кольца. Говорят, что процессы, запущенные в нулевом кольце, работают в режиме ядра или в режиме супервизора, поскольку они могут использовать такие инструкции, как sti и cli, которые большинство процессов использовать не может. Как правило, кольца 1 и 2 не используются. Технически в них можно получить доступ к большему подмножеству инструкций режима супервизора, чем это можно сделать в кольце 3. В некоторых микроядерных архитектурах эти кольца используются для запуска серверных процессов или драйверов.

Внутри дескриптора сегмента есть число, указывающее уровень кольца, который используется. Чтобы изменить уровень кольца потребуются сегменты, представляющие как кольцо 0, так и кольцо 3.

#### 4.2. Таблица глобальных дескрипторов GDT (практика)



GRUB сам настраивает таблицу GDT. Проблема в том, что вы не знаете, где находится GDT, или то, что в ней задано. Так что если вы ее случайно перезапишите, после трехкратного повторения ошибки ваш компьютер перезагрузится.

В архитектуре x86 есть 6 регистров сегментации. В каждом хранится смещение, указываемое в GDT. Это: cs (сегмент кода), ds (сегмент данных), es (дополнительный сегмент), fs, gs, ss (сегмент стека). В сегменте кода должна быть ссылка на дескриптор, который установлен как 'code segment' (сегмент кода). Для этого в байте доступе существует флаг. Остальные ссылки должны указывать на дескриптор, который определен как 'data segment' (сегмент данных).

#### 4.2.1. Файл descriptor\_tables.h

Запись в таблице GDT выглядит следующим образом:

// В этой структуре хранится содержимое одной записи GDT.

// Мы используем атрибут 'packed', которые указывает компилятору GCC,

// что в этой структуре выравнивание не выполняется.

struct gdt\_entry\_struct

{

u16int limit\_low; // Младшие 16 битов граничного значения limit.

u16int base\_low; // Младшие 16 битов адресной базы.

u8int base\_middle; // Следующие 8 битов адресной базы.

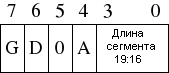
u8int access; // Флаги доступа, определяющие в каком кольце можно использовать этот сегмент.

u8int granularity;

u8int base\_high; // Последние 8 битов адресной базы.

} \_\_attribute\_\_((packed));

typedef struct gdt\_entry\_struct gdt\_entry\_t;



Большинство этих полей должны быть понятны сами по себе. Формат байта доступа показан на рисунке, приведенном ранее, а здесь показан формат байта гранулярности.

**P** - Сегмент присутствует? (1 = Да)

**DPL** - Дескриптор уровня привилегий - кольцо 0 - 3

**DT** - Тип дескриптора

**Type** - Тип сегмента - сегмент кода/сегмент данных

**G** - Гранулярность (0 = 1 байт, 1 = 1 кбайт)

**D** - Размер операнда (0 = 16 бит, 1 = 32 бита)

**0** - Всегда должно быть нулевое значение

**A** - Возможность доступа из системы (всегда должно быть нулевым)

Для того, чтобы сообщить процессору, где найти нашу таблицу GDT, мы передаем ему адрес специальной структуры с указателями:

struct gdt\_ptr\_struct

{

u16int limit; // Верхние 16 битов всех предельных значений селектора.

u32int base; // Адрес первой структуры gdt\_entry\_t.

}

\_\_attribute\_\_((packed));

typedef struct gdt\_ptr\_struct gdt\_ptr\_t;

Базой является адрес первой записи в нашей таблице GDT, предельным значением будет размер таблицы минус один (последний допустимый адрес в таблице).

Определения этих структур вместе с прототипами должны находиться в заголовочном файле descriptor\_tables.h.

// Общедоступная функция инициализации.

void init\_descriptor\_tables();

#### 4.2.2. Файл descriptor\_tables.c

В файле descriptor\_tables.c у нас находятся несколько объявлений:

// descriptor\_tables.c

|  |
| --- |
| // descriptor\_tables.c - Initialises the GDT and IDT, and defines the |
|  | // default ISR and IRQ handler. |
|  | // Based on code from Bran's kernel development tutorials. |
|  | // Rewritten for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #include "common.h" |
|  | #include "descriptor\_tables.h" |
|  | #include "isr.h" |
|  |  |
|  | // Lets us access our ASM functions from our C code. |
|  | extern void gdt\_flush(u32int); |
|  | extern void idt\_flush(u32int); |
|  |  |
|  | // Internal function prototypes. |
|  | static void init\_gdt(); |
|  | static void init\_idt(); |
|  | static void gdt\_set\_gate(s32int,u32int,u32int,u8int,u8int); |
|  | static void idt\_set\_gate(u8int,u32int,u16int,u8int); |
|  |  |
|  | gdt\_entry\_t gdt\_entries[5]; |
|  | gdt\_ptr\_t gdt\_ptr; |
|  | idt\_entry\_t idt\_entries[256]; |
|  | idt\_ptr\_t idt\_ptr; |
|  |  |
|  | // Extern the ISR handler array so we can nullify them on startup. |
|  | extern isr\_t interrupt\_handlers[]; |
|  |  |
|  | // Initialisation routine - zeroes all the interrupt service routines, |
|  | // initialises the GDT and IDT. |
|  | void init\_descriptor\_tables() |
|  | { |
|  |  |
|  | // Initialise the global descriptor table. |
|  | init\_gdt(); |
|  | // Initialise the interrupt descriptor table. |
|  | init\_idt(); |
|  | // Nullify all the interrupt handlers. |
|  | memset(&interrupt\_handlers, 0, sizeof(isr\_t)\*256); |
|  | } |
|  |  |
|  | static void init\_gdt() |
|  | { |
|  | gdt\_ptr.limit = (sizeof(gdt\_entry\_t) \* 5) - 1; |
|  | gdt\_ptr.base = (u32int)&gdt\_entries; |
|  |  |
|  | gdt\_set\_gate(0, 0, 0, 0, 0); // Null segment |
|  | gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment |
|  | gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment |
|  | gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code segment |
|  | gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data segment |
|  |  |
|  | gdt\_flush((u32int)&gdt\_ptr); |
|  | } |
|  |  |
|  | // Set the value of one GDT entry. |
|  | static void gdt\_set\_gate(s32int num, u32int base, u32int limit, u8int access, u8int gran) |
|  | { |
|  | gdt\_entries[num].base\_low = (base & 0xFFFF); |
|  | gdt\_entries[num].base\_middle = (base >> 16) & 0xFF; |
|  | gdt\_entries[num].base\_high = (base >> 24) & 0xFF; |
|  |  |
|  | gdt\_entries[num].limit\_low = (limit & 0xFFFF); |
|  | gdt\_entries[num].granularity = (limit >> 16) & 0x0F; |
|  |  |
|  | gdt\_entries[num].granularity |= gran & 0xF0; |
|  | gdt\_entries[num].access = access; |
|  | } |
|  |  |
|  | static void init\_idt() |
|  | { |
|  | idt\_ptr.limit = sizeof(idt\_entry\_t) \* 256 -1; |
|  | idt\_ptr.base = (u32int)&idt\_entries; |
|  |  |
|  | memset(&idt\_entries, 0, sizeof(idt\_entry\_t)\*256); |
|  |  |
|  | // Remap the irq table. |
|  | outb(0x20, 0x11); |
|  | outb(0xA0, 0x11); |
|  | outb(0x21, 0x20); |
|  | outb(0xA1, 0x28); |
|  | outb(0x21, 0x04); |
|  | outb(0xA1, 0x02); |
|  | outb(0x21, 0x01); |
|  | outb(0xA1, 0x01); |
|  | outb(0x21, 0x0); |
|  | outb(0xA1, 0x0); |
|  |  |
|  | idt\_set\_gate( 0, (u32int)isr0 , 0x08, 0x8E); |
|  | idt\_set\_gate( 1, (u32int)isr1 , 0x08, 0x8E); |
|  | idt\_set\_gate( 2, (u32int)isr2 , 0x08, 0x8E); |
|  | idt\_set\_gate( 3, (u32int)isr3 , 0x08, 0x8E); |
|  | idt\_set\_gate( 4, (u32int)isr4 , 0x08, 0x8E); |
|  | idt\_set\_gate( 5, (u32int)isr5 , 0x08, 0x8E); |
|  | idt\_set\_gate( 6, (u32int)isr6 , 0x08, 0x8E); |
|  | idt\_set\_gate( 7, (u32int)isr7 , 0x08, 0x8E); |
|  | idt\_set\_gate( 8, (u32int)isr8 , 0x08, 0x8E); |
|  | idt\_set\_gate( 9, (u32int)isr9 , 0x08, 0x8E); |
|  | idt\_set\_gate(10, (u32int)isr10, 0x08, 0x8E); |
|  | idt\_set\_gate(11, (u32int)isr11, 0x08, 0x8E); |
|  | idt\_set\_gate(12, (u32int)isr12, 0x08, 0x8E); |
|  | idt\_set\_gate(13, (u32int)isr13, 0x08, 0x8E); |
|  | idt\_set\_gate(14, (u32int)isr14, 0x08, 0x8E); |
|  | idt\_set\_gate(15, (u32int)isr15, 0x08, 0x8E); |
|  | idt\_set\_gate(16, (u32int)isr16, 0x08, 0x8E); |
|  | idt\_set\_gate(17, (u32int)isr17, 0x08, 0x8E); |
|  | idt\_set\_gate(18, (u32int)isr18, 0x08, 0x8E); |
|  | idt\_set\_gate(19, (u32int)isr19, 0x08, 0x8E); |
|  | idt\_set\_gate(20, (u32int)isr20, 0x08, 0x8E); |
|  | idt\_set\_gate(21, (u32int)isr21, 0x08, 0x8E); |
|  | idt\_set\_gate(22, (u32int)isr22, 0x08, 0x8E); |
|  | idt\_set\_gate(23, (u32int)isr23, 0x08, 0x8E); |
|  | idt\_set\_gate(24, (u32int)isr24, 0x08, 0x8E); |
|  | idt\_set\_gate(25, (u32int)isr25, 0x08, 0x8E); |
|  | idt\_set\_gate(26, (u32int)isr26, 0x08, 0x8E); |
|  | idt\_set\_gate(27, (u32int)isr27, 0x08, 0x8E); |
|  | idt\_set\_gate(28, (u32int)isr28, 0x08, 0x8E); |
|  | idt\_set\_gate(29, (u32int)isr29, 0x08, 0x8E); |
|  | idt\_set\_gate(30, (u32int)isr30, 0x08, 0x8E); |
|  | idt\_set\_gate(31, (u32int)isr31, 0x08, 0x8E); |
|  | idt\_set\_gate(32, (u32int)irq0, 0x08, 0x8E); |
|  | idt\_set\_gate(33, (u32int)irq1, 0x08, 0x8E); |
|  | idt\_set\_gate(34, (u32int)irq2, 0x08, 0x8E); |
|  | idt\_set\_gate(35, (u32int)irq3, 0x08, 0x8E); |
|  | idt\_set\_gate(36, (u32int)irq4, 0x08, 0x8E); |
|  | idt\_set\_gate(37, (u32int)irq5, 0x08, 0x8E); |
|  | idt\_set\_gate(38, (u32int)irq6, 0x08, 0x8E); |
|  | idt\_set\_gate(39, (u32int)irq7, 0x08, 0x8E); |
|  | idt\_set\_gate(40, (u32int)irq8, 0x08, 0x8E); |
|  | idt\_set\_gate(41, (u32int)irq9, 0x08, 0x8E); |
|  | idt\_set\_gate(42, (u32int)irq10, 0x08, 0x8E); |
|  | idt\_set\_gate(43, (u32int)irq11, 0x08, 0x8E); |
|  | idt\_set\_gate(44, (u32int)irq12, 0x08, 0x8E); |
|  | idt\_set\_gate(45, (u32int)irq13, 0x08, 0x8E); |
|  | idt\_set\_gate(46, (u32int)irq14, 0x08, 0x8E); |
|  | idt\_set\_gate(47, (u32int)irq15, 0x08, 0x8E); |
|  |  |
|  | idt\_flush((u32int)&idt\_ptr); |
|  | } |
|  |  |
|  | static void idt\_set\_gate(u8int num, u32int base, u16int sel, u8int flags) |
|  | { |
|  | idt\_entries[num].base\_lo = base & 0xFFFF; |
|  | idt\_entries[num].base\_hi = (base >> 16) & 0xFFFF; |
|  |  |
|  | idt\_entries[num].sel = sel; |
|  | idt\_entries[num].always0 = 0; |
|  | // We must uncomment the OR below when we get to using user-mode. |
|  | // It sets the interrupt gate's privilege level to 3. |
|  | idt\_entries[num].flags = flags /\* | 0x60 \*/; |
|  | } |

Обратите внимание на функцию gdt\_flush - она определена в ассемблерном файле и для нас загружен указатель на нашу таблицу GDT.

Сначала init\_gdt настраивает структуру указателей GDT - предельное значение limit является размером записи GDT, умноженной на 5, - у нас 5 записей, так как есть дескрипторы сегмента кода и сегмента данных для ядра, дескрипторы сегмента кода и сегмента данных для пользовательского режима и пустая запись null. Она должна присутствовать, иначе могут произойти неприятные вещи.

Затем gdt\_init настраивает 5 дескрипторов с помощью обращения к gdt\_set\_gate. Единственное различие в дескрипторах четырех сегментов - это байт доступа 0x9A, 0x92, 0xFA, 0xF2. DPL является дескриптором уровня привилегий: 3 - для пользовательского кода и 0 - для кода ядра. Type определяет, является ли сегментом кода или сегментом данных.

Наконец, у нас есть функция на языке ассемблера, которая запишет указатель GDT.

[GLOBAL gdt\_flush] ; Позволяет коду на C обращаться gdt\_flush().

gdt\_flush:

mov eax, [esp+4] ; Берет указатель на таблицу GDT, переданную в качестве параметра.

lgdt [eax] ; Загружает новый указатель GDT

mov ax, 0x10 ; 0x10 является смещением, находящимся в таблице GDT и указываемым на наш сегмент данных

mov ds, ax ; Загрузка переключателей всех сегментов данных

mov es, ax

mov fs, ax

mov gs, ax

mov ss, ax

jmp 0x08:.flush ; 0x08 является смещением на наш сегмент кода: Длинный переход!

.flush:

ret

Эта функция берет первый параметр, переданный ей (в esp+4), загружает в GDT значение указателей (с помощью инструкции lgdt), затем загружает переключатель сегментов для сегментов кода и сегментов данных. Заметьте, что длина каждой записи GDT составляет 8 байтов, а дескриптором кода ядра является второй сегмент, так что смещение будет равно 0x08. Аналогичным образом дескриптор данных ядра является третьим, так что в этом случае смещение будет 16 = 0x10. Здесь мы переносим значение 0x10 в регистры сегмента данных ds,es,fd,gs,ss. Чтобы изменить сегмент кода, нужно поступить немного по-другому; мы должны выполнить длинный переход. Это приводит к неявному изменению значения CS.

#### 4.3. Таблица дескрипторов прерываний IDT (теория)

Иногда вы хотите прервать работу процессора. Прерывание похоже на сигнал POSIX - он сообщает вам, что произошло что-то интересное. Процессор может регистрировать обработчики прерываний, которые имеют дело с прерываниями, затем передать управление в код, который обработает прерывание. Прерывания могут обрабатываться через внешние ресурсы - через IRQ, или с использованием внутренних ресурсов, т.е. с помощью инструкции 'int n'.

Таблица дескрипторов прерываний *The Interrupt Descriptor Table* указывает процессору, где найти обработчики для каждого прерывания. Эта таблица очень похожа на таблицу GDT. Это просто массив записей, каждая из которых соответствует номеру прерывания. Есть 256 допустимых номеров прерываний, так что должно быть 256 записей. Если происходит прерывание, а для него нет записи (подходит даже запись со значением NULL), то процессор перейдет в режим panic и произойдет перезагрузка системы.

#### 4.3.1. Отказы, ловушки и исключения

Процессору иногда нужно будет передать сигналы в ваше ядро. Может случиться что-то важное, например, деление на ноль или ошибка при подкачке страницы. Для того, чтобы это сделать, используются первые 32 прерывания. Поэтому вдвойне важно, чтобы все они не отображались в NULL, в противном случае после троекратной ошибки процессор выполнит перезагрузку (эмулятор bochs перейдет в режим panic и выдаст ошибку 'unhandled exception').

Ниже приведены специальные прерывания, связанные с работой процессора:

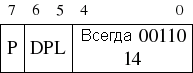
* 0 - Прерывание деления на ноль
* 1 - Прерывание отладки (пошагового исполнения)
* 2 - Немаскируемое прерывание
* 3 - Прерывание точки останова
* 4 - Переполнение при выполнении команды Into
* 5 - Прерывание выхода за границы данных
* 6 - Прерывание неправильного кода операции
* 7 - Прерывание отсутствие сопроцессора
* 8 - Прерывание двойной ошибки (код ошибки помещается в стек)
* 9 - Нарушение сегментации памяти сопроцессором
* 10 - Неправильный TSS (код ошибки помещается в стек)
* 11 - Отсутствие сегмента (код ошибки помещается в стек)
* 12 - Ошибка стека (код ошибки помещается в стек)
* 13 - Ошибка общей защиты (код ошибки помещается в стек)
* 14 - Ошибка системы страничной организации памяти (код ошибки помещается в стек)
* 15 - Неизвестное прерывание
* 16 - Ошибка сопроцессора
* 17 - Прерывание контроля выравнивания
* 18 - Прерывание, связанное с общей работой процессора
* 19-31 – Зарезервировано

#### 4.4. Таблица дескрипторов прерываний IDT (практика)

#### 4.4.1. Файл descriptor\_tables.h

// descriptor\_tables.h

|  |
| --- |
| // descriptor\_tables.h - Defines the interface for initialising the GDT and IDT. |
|  | // Also defines needed structures. |
|  | // Based on code from Bran's kernel development tutorials. |
|  | // Rewritten for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #include "common.h" |
|  | #include "panic.h" |
|  |  |
|  | // Initialisation function is publicly accessible. |
|  | void init\_descriptor\_tables(); |
|  |  |
|  | // This structure contains the value of one GDT entry. |
|  | // We use the attribute 'packed' to tell GCC not to change |
|  | // any of the alignment in the structure. |
|  | struct gdt\_entry\_struct |
|  | { |
|  | u16int limit\_low; // The lower 16 bits of the limit. |
|  | u16int base\_low; // The lower 16 bits of the base. |
|  | u8int base\_middle; // The next 8 bits of the base. |
|  | u8int access; // Access flags, determine what ring this segment can be used in. |
|  | u8int granularity; |
|  | u8int base\_high; // The last 8 bits of the base. |
|  | } \_\_attribute\_\_((packed)); |
|  |  |
|  | typedef struct gdt\_entry\_struct gdt\_entry\_t; |
|  |  |
|  | // This struct describes a GDT pointer. It points to the start of |
|  | // our array of GDT entries, and is in the format required by the |
|  | // lgdt instruction. |
|  | struct gdt\_ptr\_struct |
|  | { |
|  | u16int limit; // The upper 16 bits of all selector limits. |
|  | u32int base; // The address of the first gdt\_entry\_t struct. |
|  | } \_\_attribute\_\_((packed)); |
|  |  |
|  | typedef struct gdt\_ptr\_struct gdt\_ptr\_t; |
|  |  |
|  | // A struct describing an interrupt gate. |
|  | struct idt\_entry\_struct |
|  | { |
|  | u16int base\_lo; // The lower 16 bits of the address to jump to when this interrupt fires. |
|  | u16int sel; // Kernel segment selector. |
|  | u8int always0; // This must always be zero. |
|  | u8int flags; // More flags. See documentation. |
|  | u16int base\_hi; // The upper 16 bits of the address to jump to. |
|  | } \_\_attribute\_\_((packed)); |
|  |  |
|  | typedef struct idt\_entry\_struct idt\_entry\_t; |
|  |  |
|  | // A struct describing a pointer to an array of interrupt handlers. |
|  | // This is in a format suitable for giving to 'lidt'. |
|  | struct idt\_ptr\_struct |
|  | { |
|  | u16int limit; |
|  | u32int base; // The address of the first element in our idt\_entry\_t array. |
|  | } \_\_attribute\_\_((packed)); |
|  |  |
|  | typedef struct idt\_ptr\_struct idt\_ptr\_t; |
|  |  |
|  | // These extern directives let us access the addresses of our ASM ISR handlers. |
|  | extern void isr0 (); |
|  | extern void isr1 (); |
|  | extern void isr2 (); |
|  | extern void isr3 (); |
|  | extern void isr4 (); |
|  | extern void isr5 (); |
|  | extern void isr6 (); |
|  | extern void isr7 (); |
|  | extern void isr8 (); |
|  | extern void isr9 (); |
|  | extern void isr10(); |
|  | extern void isr11(); |
|  | extern void isr12(); |
|  | extern void isr13(); |
|  | extern void isr14(); |
|  | extern void isr15(); |
|  | extern void isr16(); |
|  | extern void isr17(); |
|  | extern void isr18(); |
|  | extern void isr19(); |
|  | extern void isr20(); |
|  | extern void isr21(); |
|  | extern void isr22(); |
|  | extern void isr23(); |
|  | extern void isr24(); |
|  | extern void isr25(); |
|  | extern void isr26(); |
|  | extern void isr27(); |
|  | extern void isr28(); |
|  | extern void isr29(); |
|  | extern void isr30(); |
|  | extern void isr31(); |
|  | extern void irq0 (); |
|  | extern void irq1 (); |
|  | extern void irq2 (); |
|  | extern void irq3 (); |
|  | extern void irq4 (); |
|  | extern void irq5 (); |
|  | extern void irq6 (); |
|  | extern void irq7 (); |
|  | extern void irq8 (); |
|  | extern void irq9 (); |
|  | extern void irq10(); |
|  | extern void irq11(); |
|  | extern void irq12(); |
|  | extern void irq13(); |
|  | extern void irq14(); |
|  | extern void irq15(); |



На рисунке показан формат поля флагов. Младшие 5 битов должны всегда иметь одно и то же значение 0b0110, т.е.14 в десятичной системе. DPL описывает уровень привилегий, с какими, как мы ожидаем, будет осуществляться. Бит P означает, что запись есть. Любой дескриптор со сброшенным этим битом явно указывает на исключительную ситуацию "прерывание не обрабатывается".

#### 4.4.2. Файл descriptor\_tables.c

Мы должны изменить этот файл и добавить в него наш новый код:

// descriptor\_tables.c

|  |
| --- |
| // descriptor\_tables.c - Initialises the GDT and IDT, and defines the |
|  | // default ISR and IRQ handler. |
|  | // Based on code from Bran's kernel development tutorials. |
|  | // Rewritten for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #include "common.h" |
|  | #include "descriptor\_tables.h" |
|  | #include "isr.h" |
|  |  |
|  | // Lets us access our ASM functions from our C code. |
|  | extern void gdt\_flush(u32int); |
|  | extern void idt\_flush(u32int); |
|  |  |
|  | // Internal function prototypes. |
|  | static void init\_gdt(); |
|  | static void init\_idt(); |
|  | static void gdt\_set\_gate(s32int,u32int,u32int,u8int,u8int); |
|  | static void idt\_set\_gate(u8int,u32int,u16int,u8int); |
|  |  |
|  | gdt\_entry\_t gdt\_entries[5]; |
|  | gdt\_ptr\_t gdt\_ptr; |
|  | idt\_entry\_t idt\_entries[256]; |
|  | idt\_ptr\_t idt\_ptr; |
|  |  |
|  | // Extern the ISR handler array so we can nullify them on startup. |
|  | extern isr\_t interrupt\_handlers[]; |
|  |  |
|  | // Initialisation routine - zeroes all the interrupt service routines, |
|  | // initialises the GDT and IDT. |
|  | void init\_descriptor\_tables() |
|  | { |
|  |  |
|  | // Initialise the global descriptor table. |
|  | init\_gdt(); |
|  | // Initialise the interrupt descriptor table. |
|  | init\_idt(); |
|  | // Nullify all the interrupt handlers. |
|  | memset(&interrupt\_handlers, 0, sizeof(isr\_t)\*256); |
|  | } |
|  |  |
|  | static void init\_gdt() |
|  | { |
|  | gdt\_ptr.limit = (sizeof(gdt\_entry\_t) \* 5) - 1; |
|  | gdt\_ptr.base = (u32int)&gdt\_entries; |
|  |  |
|  | gdt\_set\_gate(0, 0, 0, 0, 0); // Null segment |
|  | gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment |
|  | gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment |
|  | gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code segment |
|  | gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data segment |
|  |  |
|  | gdt\_flush((u32int)&gdt\_ptr); |
|  | } |
|  |  |
|  | // Set the value of one GDT entry. |
|  | static void gdt\_set\_gate(s32int num, u32int base, u32int limit, u8int access, u8int gran) |
|  | { |
|  | gdt\_entries[num].base\_low = (base & 0xFFFF); |
|  | gdt\_entries[num].base\_middle = (base >> 16) & 0xFF; |
|  | gdt\_entries[num].base\_high = (base >> 24) & 0xFF; |
|  |  |
|  | gdt\_entries[num].limit\_low = (limit & 0xFFFF); |
|  | gdt\_entries[num].granularity = (limit >> 16) & 0x0F; |
|  |  |
|  | gdt\_entries[num].granularity |= gran & 0xF0; |
|  | gdt\_entries[num].access = access; |
|  | } |
|  |  |
|  | static void init\_idt() |
|  | { |
|  | idt\_ptr.limit = sizeof(idt\_entry\_t) \* 256 -1; |
|  | idt\_ptr.base = (u32int)&idt\_entries; |
|  |  |
|  | memset(&idt\_entries, 0, sizeof(idt\_entry\_t)\*256); |
|  |  |
|  | // Remap the irq table. |
|  | outb(0x20, 0x11); |
|  | outb(0xA0, 0x11); |
|  | outb(0x21, 0x20); |
|  | outb(0xA1, 0x28); |
|  | outb(0x21, 0x04); |
|  | outb(0xA1, 0x02); |
|  | outb(0x21, 0x01); |
|  | outb(0xA1, 0x01); |
|  | outb(0x21, 0x0); |
|  | outb(0xA1, 0x0); |
|  |  |
|  | idt\_set\_gate( 0, (u32int)isr0 , 0x08, 0x8E); |
|  | idt\_set\_gate( 1, (u32int)isr1 , 0x08, 0x8E); |
|  | idt\_set\_gate( 2, (u32int)isr2 , 0x08, 0x8E); |
|  | idt\_set\_gate( 3, (u32int)isr3 , 0x08, 0x8E); |
|  | idt\_set\_gate( 4, (u32int)isr4 , 0x08, 0x8E); |
|  | idt\_set\_gate( 5, (u32int)isr5 , 0x08, 0x8E); |
|  | idt\_set\_gate( 6, (u32int)isr6 , 0x08, 0x8E); |
|  | idt\_set\_gate( 7, (u32int)isr7 , 0x08, 0x8E); |
|  | idt\_set\_gate( 8, (u32int)isr8 , 0x08, 0x8E); |
|  | idt\_set\_gate( 9, (u32int)isr9 , 0x08, 0x8E); |
|  | idt\_set\_gate(10, (u32int)isr10, 0x08, 0x8E); |
|  | idt\_set\_gate(11, (u32int)isr11, 0x08, 0x8E); |
|  | idt\_set\_gate(12, (u32int)isr12, 0x08, 0x8E); |
|  | idt\_set\_gate(13, (u32int)isr13, 0x08, 0x8E); |
|  | idt\_set\_gate(14, (u32int)isr14, 0x08, 0x8E); |
|  | idt\_set\_gate(15, (u32int)isr15, 0x08, 0x8E); |
|  | idt\_set\_gate(16, (u32int)isr16, 0x08, 0x8E); |
|  | idt\_set\_gate(17, (u32int)isr17, 0x08, 0x8E); |
|  | idt\_set\_gate(18, (u32int)isr18, 0x08, 0x8E); |
|  | idt\_set\_gate(19, (u32int)isr19, 0x08, 0x8E); |
|  | idt\_set\_gate(20, (u32int)isr20, 0x08, 0x8E); |
|  | idt\_set\_gate(21, (u32int)isr21, 0x08, 0x8E); |
|  | idt\_set\_gate(22, (u32int)isr22, 0x08, 0x8E); |
|  | idt\_set\_gate(23, (u32int)isr23, 0x08, 0x8E); |
|  | idt\_set\_gate(24, (u32int)isr24, 0x08, 0x8E); |
|  | idt\_set\_gate(25, (u32int)isr25, 0x08, 0x8E); |
|  | idt\_set\_gate(26, (u32int)isr26, 0x08, 0x8E); |
|  | idt\_set\_gate(27, (u32int)isr27, 0x08, 0x8E); |
|  | idt\_set\_gate(28, (u32int)isr28, 0x08, 0x8E); |
|  | idt\_set\_gate(29, (u32int)isr29, 0x08, 0x8E); |
|  | idt\_set\_gate(30, (u32int)isr30, 0x08, 0x8E); |
|  | idt\_set\_gate(31, (u32int)isr31, 0x08, 0x8E); |
|  | idt\_set\_gate(32, (u32int)irq0, 0x08, 0x8E); |
|  | idt\_set\_gate(33, (u32int)irq1, 0x08, 0x8E); |
|  | idt\_set\_gate(34, (u32int)irq2, 0x08, 0x8E); |
|  | idt\_set\_gate(35, (u32int)irq3, 0x08, 0x8E); |
|  | idt\_set\_gate(36, (u32int)irq4, 0x08, 0x8E); |
|  | idt\_set\_gate(37, (u32int)irq5, 0x08, 0x8E); |
|  | idt\_set\_gate(38, (u32int)irq6, 0x08, 0x8E); |
|  | idt\_set\_gate(39, (u32int)irq7, 0x08, 0x8E); |
|  | idt\_set\_gate(40, (u32int)irq8, 0x08, 0x8E); |
|  | idt\_set\_gate(41, (u32int)irq9, 0x08, 0x8E); |
|  | idt\_set\_gate(42, (u32int)irq10, 0x08, 0x8E); |
|  | idt\_set\_gate(43, (u32int)irq11, 0x08, 0x8E); |
|  | idt\_set\_gate(44, (u32int)irq12, 0x08, 0x8E); |
|  | idt\_set\_gate(45, (u32int)irq13, 0x08, 0x8E); |
|  | idt\_set\_gate(46, (u32int)irq14, 0x08, 0x8E); |
|  | idt\_set\_gate(47, (u32int)irq15, 0x08, 0x8E); |
|  |  |
|  | idt\_flush((u32int)&idt\_ptr); |
|  | } |
|  |  |
|  | static void idt\_set\_gate(u8int num, u32int base, u16int sel, u8int flags) |
|  | { |
|  | idt\_entries[num].base\_lo = base & 0xFFFF; |
|  | idt\_entries[num].base\_hi = (base >> 16) & 0xFFFF; |
|  |  |
|  | idt\_entries[num].sel = sel; |
|  | idt\_entries[num].always0 = 0; |
|  | // We must uncomment the OR below when we get to using user-mode. |
|  | // It sets the interrupt gate's privilege level to 3. |
|  | idt\_entries[num].flags = flags /\* | 0x60 \*/; |
|  | } |

Мы также намерены добавить в gdt.s следующий код:

[GLOBAL idt\_flush] ; Allows the C code to call idt\_flush().

idt\_flush:

mov eax, [esp+4] ; Берет указатель на IDT, передаваемый в качестве параметра.

lidt [eax] ; Загружает указатель IDT.

Ret

#### 4.4.3. Файл interrupt.s

Когда процессор получает прерывание, он сохраняет в стеке содержимое важных регистров (указателя команд, указателя стека, сегментов кода и данных, регистра флагов). Затем он по таблице IDT находит место, где расположен обработчик прерываний, и передает ему управление.

Мы избежим большого количества повторяющегося кода, если напишем много обработчиков, которые просто помещают в стек код прерывания (жестко прописано в ассемблере) и вызывают общую функцию обработчика.

Некоторые прерывания также помещают в стек код ошибки. Мы не можем вызвать общие функцию до тех пор, пока не обеспечим единообразное состояние стека, поэтому для тех прерываний, которые не помещают в стек код ошибки, мы записываем в стек еще одно фиктивное значение с тем, чтобы состояние стека было одинаковое.

[GLOBAL isr0]

isr0:

cli ; Сброс прерываний

push byte 0 ; Помещаем в стек фиктивный код ошибки (в случае, если ISR0 не помещает в стек свой собственный код ошибки)

push byte 0 ; Помещаем в стек номер прерывания (0)

jmp isr\_common\_stub ; Переходим к общей части обработчика.

Это пример работающей подпрограммы, но сделать ее 32 версии - для этого потребуется написать много кода. Впрочем, мы можем использовать макросредство NASM с тем, чтобы уменьшить количество работы:

%macro ISR\_NOERRCODE 1 ; define a macro, taking one parameter

[GLOBAL isr%1] ; %1 доступ к первому параметру.

isr%1:

cli

push byte 0

push byte %1

jmp isr\_common\_stub

%endmacro

%macro ISR\_ERRCODE 1

[GLOBAL isr%1]

isr%1:

cli

push byte %1

jmp isr\_common\_stub

%endmacro

Теперь мы можем просто написать вызовы макрофункции

ISR\_NOERRCODE 0

ISR\_NOERRCODE 1

...

Просмотр руководства Intel укажет вам, что только в прерываниях 8, 10-14 код ошибки помещается в стек. Остальные требуют использовать фиктивные коды ошибок.

1. создать на ассемблере общую функцию обработки.

2. создать высокоуровневую функцию обработки на языке C.

; В файле isr.c

[EXTERN isr\_handler]

; общая часть ISR сохраняет состояние процессора, настраивает

; сегменты на использование в режиме ядра, вызывает обработчик отказов, написанный на C

; и, наконец, восстанавливает состояние стека.

isr\_common\_stub:

push ; Помещает в стек содержимое регистров edi,esi,ebp,esp,ebx,edx,ecx,eax

mov ax, ds ; Младшие 16 битов регистра eax = ds.

push eax ; Сохранение дескриптора сегмента данных

mov ax, 0x10 ; Загрузка сегмента данных ядра

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

call isr\_handler

pop eax ; Перезагрузка оригинального дескриптора сегмента данных

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

popa ; Выталкиваем из стека значения edi,esi,ebp...

add esp, 8 ; Очищаем из стека код ошибки, помещаем в стек номер ISR

sti

iret ; Выталкиваем из стека следующие пять значений: CS, EIP, EFLAGS, SS и ESP

Этот фрагмент кода является нашим общим обработчиком прерывания. В нем, во-первых, используется команда 'pusha', которая помещает в стек всех регистры общего назначения. В нем в конце для восстановления регистров используется команда 'popa'. В нем также берется и помещается в стек переключатель текущего сегмента данных, все регистры сегмента устанавливаются в режим ядра, а после всего они восстанавливаются. В данный момент от этого действия проку не будет, он появится, когда мы переключимся в пользовательский режим. Обратите также внимание, что вызывается обработчик прерывания высокого уровня - *isr\_handler*.

Когда возникает прерывание, процессор автоматически помещает информацию о своем состоянии в стек. В стек помещаются сегмент кода, указатель команд, регистр флагов, сегмент стека и указатель стека. Инструкция IRET специально предназначена для выхода из прерывания. Она выталкивает из стека эти значения и возвращает процессор в первоначальное состояние.

#### 4.4.4. Файл isr.c

|  |
| --- |
| // isr.c -- High level interrupt service routines and interrupt request handlers. |
|  | // Part of this code is modified from Bran's kernel development tutorials. |
|  | // Rewritten for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #include "common.h" |
|  | #include "isr.h" |
|  | #include "monitor.h" |
|  |  |
|  | isr\_t interrupt\_handlers[256]; |
|  |  |
|  | void register\_interrupt\_handler(u8int n, isr\_t handler) |
|  | { |
|  | interrupt\_handlers[n] = handler; |
|  | } |
|  |  |
|  | // This gets called from our ASM interrupt handler stub. |
|  | void isr\_handler(registers\_t regs) |
|  | { |
|  | monitor\_write("recieved interrupt: "); |
|  | monitor\_write\_dec(regs.int\_no); |
|  | monitor\_put('\n'); |
|  |  |
|  | if (interrupt\_handlers[regs.int\_no] != 0) |
|  | { |
|  | isr\_t handler = interrupt\_handlers[regs.int\_no]; |
|  | handler(regs); |
|  | } |
|  | } |
|  |  |
|  | // This gets called from our ASM interrupt handler stub. |
|  | void irq\_handler(registers\_t regs) |
|  | { |
|  | // Send an EOI (end of interrupt) signal to the PICs. |
|  | // If this interrupt involved the slave. |
|  | if (regs.int\_no >= 40) |
|  | { |
|  | // Send reset signal to slave. |
|  | outb(0xA0, 0x20); |
|  | } |
|  | // Send reset signal to master. (As well as slave, if necessary). |
|  | outb(0x20, 0x20); |
|  |  |
|  | if (interrupt\_handlers[regs.int\_no] != 0) |
|  | { |
|  | isr\_t handler = interrupt\_handlers[regs.int\_no]; |
|  | handler(regs); |
|  | } |
|  |  |
|  | } |

Обработчик прерывания выдает на экран сообщение, указывая номер обрабатываемого прерывания. Здесь используется структура registers\_t, куда мы помещаем содержимое регистров и которая была определена в isr.h:

#### 4.4.5. Файл isr.h

|  |
| --- |
| // isr.h -- Interface and structures for high level interrupt service routines. |
|  | // Part of this code is modified from Bran's kernel development tutorials. |
|  | // Rewritten for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #include "common.h" |
|  | #include "panic.h" |
|  |  |
|  | // A few defines to make life a little easier |
|  | #define IRQ0 32 |
|  | #define IRQ1 33 |
|  | #define IRQ2 34 |
|  | #define IRQ3 35 |
|  | #define IRQ4 36 |
|  | #define IRQ5 37 |
|  | #define IRQ6 38 |
|  | #define IRQ7 39 |
|  | #define IRQ8 40 |
|  | #define IRQ9 41 |
|  | #define IRQ10 42 |
|  | #define IRQ11 43 |
|  | #define IRQ12 44 |
|  | #define IRQ13 45 |
|  | #define IRQ14 46 |
|  | #define IRQ15 47 |
|  |  |
|  | typedef struct registers |
|  | { |
|  | u32int ds; // Data segment selector |
|  | u32int edi, esi, ebp, esp, ebx, edx, ecx, eax; // Pushed by pusha. |
|  | u32int int\_no, err\_code; // Interrupt number and error code (if applicable) |
|  | u32int eip, cs, eflags, useresp, ss; // Pushed by the processor automatically. |
|  | } registers\_t; |
|  |  |
|  | // Enables registration of callbacks for interrupts or IRQs. |
|  | // For IRQs, to ease confusion, use the #defines above as the |
|  | // first parameter. |
|  | typedef void (\*isr\_t)(registers\_t); |
|  | void register\_interrupt\_handler(u8int n, isr\_t handler); |

#### 4.4.6. Проверка и вывод результата

Добавим следующий код к вашей функции main():

asm volatile ("int $0x3");

asm volatile ("int $0x4");

В результат будет вызвано два программных прерываний: 3 и 4.

## 5. Запросы на прерывания IRQ и таймер PIT

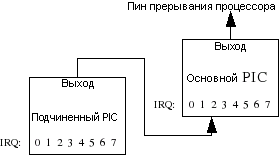
### 5.1. Запросы прерываний (теория)

Есть несколько способов связи с внешними устройствами. Двумя из наиболее часто используемых и популярных являются опрос устройств и использование прерываний.

###### Опрос устройств: выполняется в цикле, случай от случая и проверяя, готово ли устройство.

Использование прерываний: используется много устройств. Когда конкретное устройство будет готово, оно вызовет прерывание процессора, в результате чего будет запущен обработчик прерывания.

Опрос устройств используется во многих случаях - в некоторых процессорах нет механизма работы с прерываниями, либо у вас может быть много устройств, либо, возможно, вам просто не нужно проверять слишком часто, что это не стоит делать с помощью прерываний. По любому, использование прерываний очень полезный способ взаимодействия аппаратного обеспечения. Прерывания используются клавиатурой при нажатии клавиш, а также программируемым интервальным таймером (PIT).



Низкоуровневые концепции, лежащие за понятием внешних прерываний, не очень сложны. Во всех устройствах, которые способны вызывать прерывания, есть цепь, соединяющая их с устройством PIC (программируемый контроллер прерываний). PIC является единственным устройством, которое напрямую связано с выводом процессора, на который поступает прерывание. PIC может расставлять приоритеты между устройствами, посылающими прерывания. Во всех современных компьютерах у вас есть 2 устройства PIC, основное и подчиненное, позволяющие обслуживать в общей сложности 15 устройств, использующих прерывания (одна линия используется для связи с подчиненным устройством PIC).

Вы можете изменить номер прерывания, которое поступает по каждой линии IRQ. Это называется переназначением PIC и является чрезвычайно полезным свойством.

#### 5.2. Запросы прерываний (практика)

Взаимодействие устройств PIC осуществляется через шину ввода/вывода. В каждом устройстве есть порт команд и порт данных:

* Главное устройство: команда - 0x20, данные - 0x21
* Подчиненное устройство: команда - 0xA0, данные - 0xA1

static void init\_idt()

{

...

// Remap the irq table.

outb(0x20, 0x11);

outb(0xA0, 0x11);

outb(0x21, 0x20);

outb(0xA1, 0x28);

outb(0x21, 0x04);

outb(0xA1, 0x02);

outb(0x21, 0x01);

outb(0xA1, 0x01);

outb(0x21, 0x0);

outb(0xA1, 0x0);

...

idt\_set\_gate(32, (u32int)irq0, 0x08, 0x8E);

...

idt\_set\_gate(47, (u32int)irq15, 0x08, 0x8E);

}

Заметьте, что сейчас мы также задали в шлюзе IDT номера 32-47 для нашего обработчика IRQ. Поэтому мы также должны добавить в файл interrupt.s фрагменты кода для каждого прерывания. Также нам нужен новый макрос в interrupt.s - во фрагментах кода для каждого прерывания указываются два числа связанные с этим прерыванием - это номер IRQ (0-15) и номер прерывания (32-47):

; Этот макрос создает кусок кода для IRQ - первым параметром

; является номер IRQ, вторым - номер ISR, на который осуществляется переназначение.

%macro IRQ 2

global irq%1

irq%1:

cli

push byte 0

push byte %2

jmp irq\_common\_stub

%endmacro

...

IRQ 0, 32

IRQ 1, 33

...

IRQ 15, 47

У нас также есть новый общий код - irq\_common\_stub. Это связано с тем, что прерывания будут вести себя немного по-другому - прежде чем вернуться из обработчика IRQ, вы должны сообщить устройству PIC о том, что обработка закончена, для того чтобы можно было быстро перейти к следующему прерыванию (если оно находится в состоянии ожидания). Эта операция известна как о как EOI (end of interrupt - завершение прерывания). Однако, есть небольшая сложность. Если главное устройство PIC посылает прерывание IRQ (номер 0-7), мы должны (что очевидно) отправить EOI в главное устройство. Если подчиненное устройство PIC посылает IRQ (8-15), мы должны отправить EOI как в главное, так и в подчиненное устройства.

Первый наш фрагмент общего кода на ассемблере. Он почти идентичен isr\_common\_stub.

; В файле isr.c

[EXTERN irq\_handler]

; Это наш общий фрагмент для обработки IRQ. Он сохраняет состояние процессора, настраивает

; сегменты режима ядра, вызывает обработчик прерываний уровня языка C и, в конце концов,

; восстанавливает состояние стека.

irq\_common\_stub:

pusha ; Помещает в стек edi,esi,ebp,esp,ebx,edx,ecx,eax

mov ax, ds ; Младшие 16 битов регистра eax = ds.

push eax ; сохраняет дескриптор сегмента данных

mov ax, 0x10 ; Загрузка дескриптора сегмента данных ядра

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

call irq\_handler

pop ebx ; Перезагрузка исходного дескриптора сегмента данных

mov ds, bx

mov es, bx

mov fs, bx

mov gs, bx

popa ; Выталкивает из стека значения edi,esi,ebp...

add esp, 8 ; Очищает код ошибки, помещенный в стек, и помещает в стек номер ISR

sti

iret ; выталкивает из стека следующие пять значений: CS, EIP, EFLAGS, SS и ESP

Теперь код на языке C (переходим в файл isr.c):

// Этот фрагмент вызывается из кода обработчика прерывания, написанного на ассемблере.

void irq\_handler(registers\_t regs)

{

// Посылает сигнал EOI (завершение прерывания) в устройства PIC.

// Если к возникновению прерывания причастно подчиненное устройство.

if (regs.int\_no >= 40)

{

// Send reset signal to slave.

outb(0xA0, 0x20);

}

// Посылает сигнал перезагрузки в главное устройство (а также в подчиненное устройство, если это необходимо).

outb(0x20, 0x20);

if (interrupt\_handlers[regs.int\_no] != 0)

{

isr\_t handler = interrupt\_handlers[regs.int\_no];

handler(regs);

}

}

Здесь все довольно очевидно - если IRQ было больше 7 (номер прерывания > 40), мы посылаем сигнал перезагрузки в подчиненное устройство. В любом случае мы посылаем также сигнал в главное устройство.

#### 5.2.1. Файл isr.h

// Несколько определений, которые сделаю жизнь немного проще

#define IRQ0 32

...

#define IRQ15 47

// Разрешает регистрировать обратные вызовы (callbacks) для прерываний или IRQ.

// Чтобы избежать неразберихи используйте для IRQ в качестве первого параметра

// определения #define, укаазанные выше.

typedef void (\*isr\_t)(registers\_t);

void register\_interrupt\_handler(u8int n, isr\_t handler);

#### 5.2.2. Файл isr.c

isr\_t interrupt\_handlers[256];

void register\_interrupt\_handler(u8int n, isr\_t handler)

{

interrupt\_handlers[n] = handler;

}

Теперь мы можем обрабатывать запросы прерываний от внешних устройств, а также перенаправлять их специальным обработчикам.

#### 5.3. Таймер PIT

Программируемый интервальный таймер - микросхема, подключенная к IRQ0. Он может прервать работу процессора с частотой, определяемой пользователем (между 18,2 Hz и 11931 МГц). PIT является основным способом, используемым для реализации системных часов и единственным доступным методом реализации многозадачности.

В PIT есть внутренний генератор частоты. Этот тактовый сигнал подается через делитель частоты для получения в результате модуляции окончательной выходной частоты. В таймере есть 3 канала, каждый со своим собственным делителем частоты.

Канал 0 является единственным полезным.

Настроим PIT так, чтобы он прерывал через регулярные промежутки времени с частотой f. Для того, чтобы установить частоту f, посылаем в PIT делитель 'divisor' число, на которое нужно разделить его основную частоту (19131 MHz). Он определяется следующим образом:

divisor = 1193180 Hz / частота (в Hz)

В PIT есть 4 регистра в пространстве ввода/вывода. 0x40-0x42 являются портами данных для каналов 0-2 соответственно, а 0x43 является портом команды.

#### 5.4. Таймер PIT

В файле Timer.h находятся только объявления:

|  |
| --- |
| // timer.h -- Defines the interface for all PIT-related functions. |
|  | // Written for JamesM's kernel development tutorials. |
|  |  |
|  | #ifndef TIMER\_H |
|  | #define TIMER\_H |
|  |  |
|  | #include "common.h" |
|  | #include "panic.h" |
|  |  |
|  | void init\_timer(u32int frequency); |
|  |  |
|  | #endif |

файл timer.c:

|  |
| --- |
| // timer.c -- Initialises the PIT, and handles clock updates. |
|  | // Written for JamesM's kernel development tutorials. |
|  |  |
|  | #include "timer.h" |
|  | #include "isr.h" |
|  | #include "monitor.h" |
|  |  |
|  | u32int tick = 0; |
|  |  |
|  | static void timer\_callback(registers\_t regs) |
|  | { |
|  | tick++; |
|  | monitor\_write("Tick: "); |
|  | monitor\_write\_dec(tick); |
|  | monitor\_write("\n"); |
|  | } |
|  |  |
|  | void init\_timer(u32int frequency) |
|  | { |
|  | // Firstly, register our timer callback. |
|  | register\_interrupt\_handler(IRQ0, &timer\_callback); |
|  |  |
|  | // The value we send to the PIT is the value to divide it's input clock |
|  | // (1193180 Hz) by, to get our required frequency. Important to note is |
|  | // that the divisor must be small enough to fit into 16-bits. |
|  | u32int divisor = 1193180 / frequency; |
|  |  |
|  | // Send the command byte. |
|  | outb(0x43, 0x36); |
|  |  |
|  | // Divisor has to be sent byte-wise, so split here into upper/lower bytes. |
|  | u8int l = (u8int)(divisor & 0xFF); |
|  | u8int h = (u8int)( (divisor>>8) & 0xFF ); |
|  |  |
|  | // Send the frequency divisor. |
|  | outb(0x40, l); |
|  | outb(0x40, h); |
|  | } |

init\_timer сообщает механизму прерывания что обрабатываем IRQ0 с помощью функции timer\_callback. Вызывается всякий раз, когда от таймера будет поступать прерывание. Рассчитываем делитель, который будет отправлен в PIT. Посылаем байт команды в командный порт PIT. Байт (0x36) устанавливает PIT в режим повторителя (когда значение счетчика достигнет нуля, он автоматически будет обновлен) и сообщаем ему, что мы хотим установить значение делителя.

Затем мы отправляем значение делителя. Обратите внимание, что он должен отсылаться ​​в виде двух отдельных байтов, а не как одно 16-битное значение.

Добавить в файл main.c следующую строчку:

init\_timer(50); // Initialise timer to 50Hz

В результату увидим на экране тики таймера.

## 6. Страничная организация памяти

#### 6.1. Виртуальная память

То, что программа «видит», когда читает из памяти и пишет в память, является виртуальным адресным пространством. Часть виртуального адресного пространства отображается в физическую память, а часть - не отображается. Если вы пытаетесь получить доступ к неотображаемой части, то процессор выдаст сигнал о некорректном обращении к памяти page fault, операционная система ловит его, и в POSIX-системе выдает сигнал SIGSEGV, за которым непосредственно следует сигнал SIGKILL.

Эта абстракция является чрезвычайно полезной. Это означает, что с помощью компиляторов можно создавать программы, которые будут, когда запускаются, размещать код в конкретном месте в памяти. При использовании виртуальной памяти процесс считает, что он, например, находится по адресу 0x080482a0, но на самом деле это может быть место в физической памяти с адресом 0x1000000. В результате процессы не могут случайно (или намеренно) испортить данные других процессов.

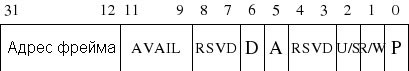
В архитектуре x86 есть возможность эмулировать программно виртуальную память. Этот модуль называется MMU (Memory Management Unit- Устройство управление памятью) и он обрабатывает все операции по отображению памяти, связанные с сегментацией и страничной организацией памяти, образуя слой между процессором и памятью.

### 6.2. Страничная организация памяти как конкретная реализация виртуальной памяти

Виртуальная память использует принцип абстракции. Как таковой, он требует конкретизации с помощью некоторых систем и алгоритмов. Страничная организация памяти является новой, лучшей альтернативой для архитектуры x86.

Страничная организация представляет собой разделение виртуального адресного пространства.

#### 6.2.1. Запись, описывающая страницу



В каждом процессе обычно имеются различные наборы отображения страниц, так что пространства виртуальной памяти не зависят друг от друга. В архитектуре x86 (32-разрядной) размеры страниц зафиксированы в 4 Кб по размеру. У каждой страницы есть соответствующее слово дескриптора, которое сообщает процессору, какому кадру страница сопоставляется. Страницы и кадры должны быть выровнены по границе в 4Кб (4KB будет 0x1000 байтов), при этом младшие 12 битов 32-разрядного слова всегда равны нулю. Это используется архитектурой для хранения в них информации о странице, например, присутствует ли она в памяти в режиме ядра или в пользовательском режиме и т.д. Формат этого слова изображен на рисунке.

**P** - Установлено, если страница находится в памяти.

**R/W** - Если установлено, то на страницу можно выполнять запись. Если не установлено, то страница доступна только для чтения. Это поле не используется, когда код работает в режиме ядра (если не установлен флаг CR0).

**U/S** - Если установлено, то это пользовательский режим. В противном случае, это страница в режиме супервизора (ядра). Код пользовательского режима не может выполнять запись на страницы, находящиеся в режиме ядра, или читать из них.

**Reserved** - Для внутреннего использования процессором и это поле не следует менять.

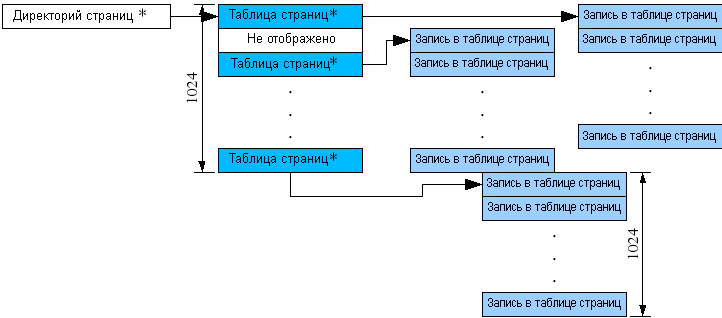
**A** - Установлено, если к странице был доступ (выполнялось обращение процессора).

**D** - Установлено, если на станице выполнялась запись (страница изменена).

**AVAIL** - Эти три бита не используются и доступны в режиме ядра.

**Page frame address** - старшие 20 битов адреса фрейма в физической памяти.

#### 6.2.2. Директории / таблицы страниц



Чтобы создать таблицу, отображающую каждую страницу размером 4KB одним 32-битный дескриптором, для 4 ГБ адресного пространства потребуется 4 Мбайт оперативной памяти.

4 Мб могут показаться большими накладными расходами, и надо быть справедливым, так оно и есть. Если вы работаете на компьютере, у которого 16 МБ оперативной памяти, вы сразу потеряли четверть доступной памяти.

В Intel придумали что-то похожее - они используют 2-х уровневую систему. Процессор получает информацию из директория страниц, большой таблицы размером в 4KB, каждая запись которой указывает на таблицу страниц. Таблица страниц опять же имеет размер в 4KB и каждая запись является записью таблицы страниц, приведенной выше.

Таким образом, все адресное пространство в 4 ГБ перекрыто таким образом, что если в таблице страниц нет записей, то ее можно очистить и в директории страниц можно сбросить флаг ее присутствия.

#### 6.2.3. Подключение страничной организации памяти

1. Скопировать место вашего директория страниц в регистр CR3. Это должен быть, конечно, физический адрес.
2. Установить бит PG в регистре CR0. Это можно с помощью операции OR и операнда 0x80000000.

#### 6.3. Некорректное обращение к памяти - page fault

Когда процесс делает что-то, что не нравится блоку управления памятью, выдается прерывание некорректного обращения к памяти - page fault. К этому могут привести следующие ситуации:

* Чтение или запись в область памяти, которая не отображена в физическую память (флаг записи страницы / 'присутствия' таблиц не установлен)
* Процесс находится в пользовательском режиме и пытается записать в страницу, доступную только для чтения страницы.
* Процесс находится в пользовательском режиме и пытается получить доступ к странице, доступной только в режиме ядра.
* Запись страницы, находящаяся в таблице, повреждена - были перезаписаны зарезервированные биты.

Прерывание некорректного обращения к памяти имеет номер 14, мы видим, что вырабатывается код ошибки. Эта ошибка дает нам довольно много информации о том, что произошло.

**Бит 0** - Если установлен, то проблема возникла не из-за того, что страница не присутствовала в физической памяти. Если не установлен, то страница не присутствует в физической памяти.

**Бит 1** - Если установлен, то это значит, что операция, из-за которой возникла проблема, была операцией записи, в противном случае это была операция чтения.

**Бит 2** - Если установлен, то прерывание произошло, когда процессор работал в пользовательском режиме. В противном случае он работал в режиме ядра.

**Бит 3** - Если установлен, то это означает, что проблема была вызвана тем, что были перезаписаны зарезервированные биты.

**Бит 4** -Если установлен, то это значит, что проблема возникла в момент выборки команд.

#### 6.4. Применяем все на практике

#### 6.4.1. Простое управление памятью с помощью команды выделения памяти malloc

Когда ядро уже достаточно загружено, у нас будет активная и функционирующая память типа heap (куча). Однако, для работы с памятью типа куча, как правило, требуется наличие виртуальной памяти. Поэтому прежде, чем можно будет пользоваться памятью типа куча, нужно иметь простую альтернативу для выделения памяти.

Поскольку нам нужно выделять память при загрузке ядра довольно рано, можно предположить, что ничего, кроме команд kmalloc() и kfree() нам не потребуется. Это существенно все упрощает. У нас может быть просто указатель (адрес размещения) для некоторой свободной памяти, который мы передаем обратно в requestee, а затем увеличиваем. Таким образом:

u32int kmalloc(u32int sz)

{

u32int tmp = placement\_address;

placement\_address += sz;

return tmp;

}

Этого, на самом деле, будет достаточно. Тем не менее, у нас есть еще одно требование. Когда мы размещаем таблицы и директории страниц, они должны быть выровнены по границам страниц. Так что мы можем собрать следующее:

u32int kmalloc(u32int sz, int align)

{

if (align == 1 && (placement\_address & 0xFFFFF000)) // Если адрес еще не выровнен по границе страниц

{

// Align it.

placement\_address &= 0xFFFFF000;

placement\_address += 0x1000;

}

u32int tmp = placement\_address;

placement\_address += sz;

return tmp;

}

Теперь, к сожалению, у нас есть еще одно требование. Оно возникает, когда мы клонируем директорий страниц (когда выполняем операцию fork()). В этот момент уже будет работать страничная организация памяти и kmalloc вернет виртуальный адрес. Но, нам также потребуется получить физический адрес выделенной памяти.

u32int kmalloc(u32int sz, int align, u32int \*phys)

{

if (align == 1 && (placement\_address & 0xFFFFF000)) // Если адрес еще не выровнен по границе страниц

{

// Align it.

placement\_address &= 0xFFFFF000;

placement\_address += 0x1000;

}

if (phys)

{

\*phys = placement\_address;

}

u32int tmp = placement\_address;

placement\_address += sz;

return tmp;

}

Несколько функций-обверток:

u32int kmalloc\_a(u32int sz); // выделяет страницу.

u32int kmalloc\_p(u32int sz, u32int \*phys); // возвращает физический адрес.

u32int kmalloc\_ap(u32int sz, u32int \*phys); // выделяет страницу и возвращает физический адрес.

u32int kmalloc(u32int sz); // Обычная функция.

Я просто чувствую, этот интерфейс лучше, чем просто указывать 3 параметра при каждом выделении памяти из кучи в ядре! Эти определения должны быть в файлах kheap.h/kheap.c.

#### 6.4.2. Необходимые определения

В файле paging.h должны находиться определения некоторых структур, которые сделают нашу жизнь проще.

#ifndef PAGING\_H

#define PAGING\_H

#include "common.h"

#include "isr.h"

typedef struct page

{

u32int present : 1; // Страница присутствует в памяти

u32int rw : 1; // Если сброшен, то страница только для чтения, если установлен, то страница для чтения и записи

u32int user : 1; // Если сброшен, то уровень супервизора

u32int accessed : 1; // Было ли обращение к странице после последнего ее обновления?

u32int dirty : 1; // Выполнялась ли запись на страницу после последнего ее обновления?

u32int unused : 7; // Все неиспользуемые и зарезервированные биты

u32int frame : 20; // Адрес фрейма (сдвинут вправо на 12 бит)

} page\_t;

typedef struct page\_table

{

page\_t pages[1024];

} page\_table\_t;

typedef struct page\_directory

{

/\*\*

Массив указателей на таблицы страниц.

\*\*/

page\_table\_t \*tables[1024];

/\*\*

Массив указателей на таблицы страниц, о которых говорилось выше, но указатели указывают \*физическое\*

местоположение, используемое при загрузке в регистр CR3.

\*\*/

u32int tablesPhysical[1024];

/\*\*

Физический адрес tablesPhysical. Его потребуется использовать в случае,

когда мы получаем в ядре память типа куча, а директорий может находиться

в любом месте виртуальной памяти.

\*\*/

u32int physicalAddr;

} page\_directory\_t;

/\*\*

Настройка среды окружения, директориев страниц и т.д. и

включение страничной организации памяти.

\*\*/

void initialise\_paging();

/\*\*

Загружает указанны директорий страниц в регистр CR3.

\*\*/

void switch\_page\_directory(page\_directory\_t \*new);

/\*\*

Поиск указателя на необходимую страницу.

Если make == 1 в таблице страниц, в которой эта страница должна располагаться,

то страница не создана - создайте страницу!

\*\*/

page\_t \*get\_page(u32int address, int make, page\_directory\_t \*dir);

/\*\*

Обаботчик некорректного обращения к страницам.

\*\*/

void page\_fault(registers\_t regs);

Элемент physicalAddr только для того случая, когда клонируем директории страниц. В этот момент новый директорий будет иметь адрес в виртуальной памяти, который не совпадает с адресом физической памяти. Нам понадобится физический адрес для того, чтобы сообщить его процессору в случае, если мы когда-либо захотим переключать директории.

Элемент tablesPhysical аналогичен. Он решает следующую проблему: Как получить доступ к таблицам страниц? При этом в директории страниц должны храниться физические адреса, а не виртуальные. А единственный способ, которым мы можем читать из памяти и записывать в память, является использование виртуальных адресов.

Решение заключается в том, для каждого директория страниц имеется 2 массива. В одном хранятся физические адреса его таблицы со страницами (для передачи их в процессор), а в другом хранятся виртуальные адреса. В результате у нас будет только 4KB дополнительных накладных расходов для каждого директория страниц.

#### 6.4.3. Размещение фреймов физической памяти

Если мы хотим отобразить страницу во фрейм, нам нужно каким-то образом найти свободный кадр. Конечно, мы могли бы просто поддерживать огромный массив из единиц и нулей, но это было бы крайне расточительно - нам не нужны 32-бит просто для того, чтобы хранить два значения. Поэтому если мы будем использовать набор битов bitset, то мы потратим памяти в 32 раза меньше!

В реализациях bitset есть только 3 функции - set (установить), test (проверить) и clear (сбросить).

// Набор bitset для фреймов.

u32int \*frames;

u32int nframes;

// Определено в kheap.c

extern u32int placement\_address;

// В алгоритмах для bitset используются макросы.

#define INDEX\_FROM\_BIT(a) (a/(8\*4))

#define OFFSET\_FROM\_BIT(a) (a%(8\*4))

// Статическая функция для установки бита в наборе bitset для фреймов

static void set\_frame(u32int frame\_addr)

{

u32int frame = frame\_addr/0x1000;

u32int idx = INDEX\_FROM\_BIT(frame);

u32int off = OFFSET\_FROM\_BIT(frame);

frames[idx] |= (0x1 << off);

}

// Статическая функция для сброса бита в наборе bitset для фреймов

static void clear\_frame(u32int frame\_addr)

{

u32int frame = frame\_addr/0x1000;

u32int idx = INDEX\_FROM\_BIT(frame);

u32int off = OFFSET\_FROM\_BIT(frame);

frames[idx] &= ~(0x1 << off);

}

// Статическая функция для проверки, установлен ли бит

static u32int test\_frame(u32int frame\_addr)

{

u32int frame = frame\_addr/0x1000;

u32int idx = INDEX\_FROM\_BIT(frame);

u32int off = OFFSET\_FROM\_BIT(frame);

return (frames[idx] & (0x1 << off));

}

// Статическая функция для поиска первого свободного фрейма

static u32int first\_frame()

{

u32int i, j;

for (i = 0; i < INDEX\_FROM\_BIT(nframes); i++)

{

if (frames[i] != 0xFFFFFFFF) // нечего не освобождаем, сразу выходим.

{

// по меньшей мере, здесь один свободный бит

for (j = 0; j < 32; j++)

{

u32int toTest = 0x1 << j;

if ( !(frames[i]&toTest) )

{

return i\*4\*8+j;

}

}

}

}

}

Затем мы переходим к функциям выделения и освобождения фреймов. Теперь, когда у нас есть эффективная реализация bitset, эти функции будут всего в несколько строк!

// Функция выделения фрейма.

void alloc\_frame(page\_t \*page, int is\_kernel, int is\_writeable)

{

if (page->frame != 0)

{

return; // Фрейм уже выделен, сразу возвращаемся.

}

else

{

u32int idx = first\_frame(); // idx теперь является индексом первого свободного фрейма.

if (idx == (u32int)-1)

{

// PANIC это всего лишь макрос, которые выдает на экран сообщение, а затем переходит в бесконечный цикл.

PANIC("No free frames!");

}

set\_frame(idx\*0x1000); // Этот фрейм теперь наш!

page->present = 1; // Помечаем его как присутствующий.

page->rw = (is\_writeable)?1:0; // Можно ли для страницы выполнять запись?

page->user = (is\_kernel)?0:1; // Находится ли страница в пользовательском режиме?

page->frame = idx;

}

}

// Function to deallocate a frame.

void free\_frame(page\_t \*page)

{

u32int frame;

if (!(frame=page->frame))

{

return; // Указанной страницы теперь фактически нет в выделенном фрейме!

}

else

{

clear\_frame(frame); // фрейм теперь снова свободен.

page->frame = 0x0; // Страницы теперь во фрейме нет.

}

}

Обратите внимание, что макрос PANIC просто вызывает глобальную функцию, которая называется panic, с аргументами \_\_FILE\_\_ и \_\_LINE\_\_. Функция panic печатает их и переходит в бесконечный цикл, останавливая все исполнение.

#### 6.4.4. Наконец-то код для страничной организации памяти

void initialise\_paging()

{

// Размер физической памяти. Сейчас мы предполагаем,

// что размер равен 16 MB.

u32int mem\_end\_page = 0x1000000;

nframes = mem\_end\_page / 0x1000;

frames = (u32int\*)kmalloc(INDEX\_FROM\_BIT(nframes));

memset(frames, 0, INDEX\_FROM\_BIT(nframes));

// Давайте создадим директорий страниц.

kernel\_directory = (page\_directory\_t\*)kmalloc\_a(sizeof(page\_directory\_t));

memset(kernel\_directory, 0, sizeof(page\_directory\_t));

current\_directory = kernel\_directory;

// Нам нужна карта идентичности (физический адрес = виртуальный адрес) с адреса

// 0x0 до конца используемой памяти с тем, чтобы у нас к ним был прозрачный

// доступ как если бы страничная организация памяти не использовалась.

// ЗАМЕТЬТЕ, что мы преднамеренно используем цикл while.

// Внутри тела цикла мы фактически изменяем адрес placement\_address

// с помощью вызова функции kmalloc(). Цикл while используется здесь, т.к. выход

// из цикла динамически, а не один раз после запуска цикла.

int i = 0;

while (i < placement\_address)

{

// Код ядра можно читать из пользовательского режима, но нельзя в него записывать.

alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);

i += 0x1000;

}

// Прежде, чем включить страничное управление памятью, нужно зарегистрировать

// обработчик некорректного обращения к памяти - page fault.

register\_interrupt\_handler(14, page\_fault);

// Теперь включаем страничную организацию памяти!

switch\_page\_directory(kernel\_directory);

}

void switch\_page\_directory(page\_directory\_t \*dir)

{

current\_directory = dir;

asm volatile("mov %0, %%cr3":: "r"(&dir->tablesPhysical));

u32int cr0;

asm volatile("mov %%cr0, %0": "=r"(cr0));

cr0 |= 0x80000000; // Enable paging!

asm volatile("mov %0, %%cr0":: "r"(cr0));

}

page\_t \*get\_page(u32int address, int make, page\_directory\_t \*dir)

{

// Помещаем адрес в индекс.

address /= 0x1000;

// Находим таблицу страниц, в которой есть этот адрес.

u32int table\_idx = address / 1024;

if (dir->tables[table\_idx]) // Если эта таблица уже назначена

{

return &dir->tables[table\_idx]->pages[address%1024];

}

else if(make)

{

u32int tmp;

dir->tables[table\_idx] = (page\_table\_t\*)kmalloc\_ap(sizeof(page\_table\_t), &tmp);

memset(dir->tables[table\_idx], 0, 0x1000);

dir->tablesPhysical[table\_idx] = tmp | 0x7; // PRESENT, RW, US.

return &dir->tables[table\_idx]->pages[address%1024];

}

else

{

return 0;

}

}

Функция switch\_page\_directory переключает директорий страниц. Она берет директорий страниц и переключается на него. Она делает это при помощи перемещения в регистр CR3 адреса элемента номера tablesPhysical этого директория. Помните, что номер tablesPhysical указывает на массив физических адресов. После этого она сначала получает содержимое регистра CR0, затем выполняет операцию OR с битом PG (0x80000000), а затем перезаписывает значение регистра. В результате включается страничная организация памяти, а также осуществляется сброс кэша с директорием страниц.

Функция get\_page возвращает указатель запись страницы для конкретного адреса. В функцию также можно передать параметр make. Если make равен 1 и таблица страницы, в которой должна находиться запись о запрашиваемой странице, еще не была создана, то она создается. В противном случае, функция просто вернет 0. Если таблица уже назначена, функция найдет запись о странице и вернет ее. Если ее нет (и make == 1), то будет предпринята попытка создать ее.

Наша функция kmalloc\_ap ищет блок памяти, который выровнен по границе страницы, и запоминает место его физического расположения. Физическое расположение сохраняется в 'tablesPhysical' (после того, как несколько битов сообщат процессору, что страница присутствует, в нее можно делать запись и она доступна пользователю), а виртуальное месторасположение запоминается в 'tables'.

Функция initialise\_paging, во-первых, создаст bitset для фреймов и с помощью команды memset заполняет его нулями. Затем она для директория страниц выделяет пространство (которое выровнено по краю страницы). После этого, она размещает фреймы так, что доступ к любой странице будет отображаться во фрейм с соответствующим линейным адресом, называемым идентичным отображением. Это сделано для небольшой части адресного пространства, так что код ядра может работать, как обычно. Она регистрирует обработчик прерывания для неверного обращения к странице (смотрите ниже), а затем включает страничную организацию памяти.

#### 6.4.5. Обработчик неверного обращения к странице

void page\_fault(registers\_t regs)

{

// Возникло прерывания неверного обращения к странице - page fault.

// Адрес прерывания запоминается в регистре CR2.

u32int faulting\_address;

asm volatile("mov %%cr2, %0" : "=r" (faulting\_address));

// Код ошибки подробно сообщит нам о том, что случилось.

int present = !(regs.err\_code & 0x1); // Страница отсутствует

int rw = regs.err\_code & 0x2; // Операция записи?

int us = regs.err\_code & 0x4; // Процессор находится в пользовательском режиме?

int reserved = regs.err\_code & 0x8; // В записи страницы переписаны биты, зарезервированные для нужд процессора?

int id = regs.err\_code & 0x10; // Причина во время выборки инструкции?

// Выдача сообщения об ошибке.

monitor\_write("Page fault! ( ");

if (present) {monitor\_write("present ");}

if (rw) {monitor\_write("read-only ");}

if (us) {monitor\_write("user-mode ");}

if (reserved) {monitor\_write("reserved ");}

monitor\_write(") at 0x");

monitor\_write\_hex(faulting\_address);

monitor\_write("\n");

PANIC("Page fault");

}

Все, что этот обработчик делает, это выдает красивое сообщение об ошибке. Он получает из регистра CR2 адрес прерывания и анализирует код ошибки, который был помещен в стек процессором, с тем, чтобы из него выбрать некоторую информацию.

#### 6.4.6. Тестирование

main.c

int main(struct multiboot \*mboot\_ptr)

{

// Инициализирует все6 значения ISRs и сегментацию

init\_descriptor\_tables();

// Инициализирует экран (очищает его)

monitor\_clear();

initialise\_paging();

monitor\_write("Hello, paging world!\n");

u32int \*ptr = (u32int\*)0xA0000000;

u32int do\_page\_fault = \*ptr;

return 0;

}

При запуске становиться видно, что выполняется инициализация страничной организации, выдается строка сообщения, указывающего, что все настроено правильно и, при этом, не было никаких проблем, а затем принудительно осуществляется неверное обращение к странице с помощью чтения по адресу 0xA0000000.

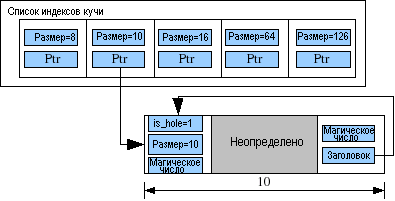
## 7. Память типа куча

Существующая система выделения памяти (выделение с указанием адреса размещения данных) замечательная и, на самом деле, оптимальна как по времени, так и по распределяемому пространству. Проблема возникает, когда вы пытаетесь освободить часть памяти и затем хотите ее повторно использовать (это должно, в конце концов, потребоваться, в противном случае вам может не хватить памяти!). В механизме выделения нет никакого способа сделать это и, следовательно, он нежизнеспособен в большинстве ситуаций, связанных с выделением в ядре памяти.

В качестве дополнительного средства любая структура данных, для которой осуществляется выделение и освобождение памяти, может обращаться к памяти типа куча (или пул памяти). Стандарт на 'алгоритм работы с кучей', как таковой, не существует. В зависимости от требований, связанных с затрачиваемым временем, используемой памятью и эффективностью, используются различные алгоритмы. Наши требования следующие:

* (относительная) простота реализации
* возможность проверять правильность работы - отладка перезаписи памяти в ядре намного труднее, чем в обычных приложениях.

### 7.1. Определение структуры данных



### 7.2. Описание алгоритма

#### 7.2.1. Выделение памяти

Большинство шагов представляет собой проверку ошибок и выделение памяти из такого фрагмента, чтобы минимизировать утечку памяти.

1. Поиск по таблице индексов с тем, чтобы найти наименьший фрагмент памяти, который будет соответствовать запрашиваемому размеру памяти. Поскольку таблица упорядочена, для поиска просто требуется итерация до тех пор, пока мы не найдем фрагмент, соответствующий нашему запросу.
   * Если фрагмент памяти достаточного размера не найден, то нужно:
     1. Увеличить кучу
     2. Если таблица индексов пуста (нет зарегистрированных фрагментов памяти), то добавить в таблицу новую запись.
     3. В противном случае изменить элемент, указывающий размер последнего заголовка блока и переписать запись, находящуюся в конце блока.
     4. Чтобы уменьшить количество инструкций управления, можно просто снова рекурсивно вызвать функцию выделения памяти, полагая, что на этот раз будет найден достаточно большой фрагмент памяти.
2. Определить, должен ли фрагмент памяти быть разделен на две части. Как правило, это делается в случае, когда нам нужно гораздо меньше места, чем есть во фрагменте памяти. Единственное, когда этого не потребуется, если после выделения блока свободного места останется меньше, чем нужно на заголовок и на запись в конце блока. В этом случае мы можем просто увеличить размер блока и потом повторить все снова.
3. Если блок должен быть выровнен по границе страницы, мы должны изменить начальный адрес блока так, что останется новый фрагмент памяти в неиспользованной области.
   * Если это не так, то просто удалить фрагмент из списка индексов.
4. Засать заголовок и завершающую запись новых блоков.
5. Если фрагмент был разделен на две части, выполнить это сейчас и запишите новый фрагмент в список индексов.
6. Вернуть поользователю адрес блока + SizeOf (header\_t).

#### 7.2.2. Освобождение памяти

Возвращение (освобождение) памяти немного сложнее. Как уже упоминалось ранее, эффективность алгоритма управления памятью действительно проверена. Проблемой является эффективное повторное использование памяти. Наивным решением было бы просто объявить этот блок фрагментом неиспользуемой памяти и вернуть его обратно в список индексов фрагментов.

int a = kmalloc(8); // Выделяются 8 байтов: возвращает 0xC0080000 в качестве аргумента

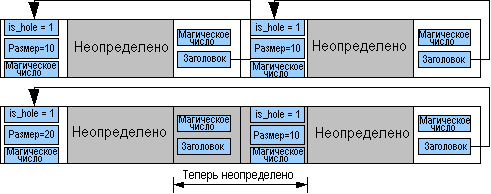
int b = kmalloc(8); // Выделяются еще 8 байтов: возвращается 0xC0080008.

kfree(a); // Освобождается a

kfree(b); // Освобождается b

int c = kmalloc(16);// Что вернет эта операция выделения памяти?

Здесь мы дважды выделили место по 8 байтов. Затем мы освободили эти оба выделенные фрагменты. При использовании наивного решения освобождение памяти завершится наличием в списке индексов двух фрагментов размером по 8 байтов. При следующем выделении памяти (16 байтов) ни один из этих фрагментов нам не подходит, поэтому при обращении к kmalloc () мы получим адрес 0xC0080010. Это не оптимально. По адресу 0xC0080000 есть 16 байтов свободного места, поэтому мы должны их выделить заново.



Решение этой проблемы в большинстве случаев представляет собой вариацию простого алгоритма унификации - то есть, преобразование двух соседних фрагментов в один.

Это работает следующим образом: Когда происходит освобождение блока памяти, то делается попытка определить, что находится сразу слева перед заголовком блока (при условии, что адресация от 0 и до 4GB идет слева - направо). Если это завершающая запись другого блока, что можно обнаружить по значению магического числа, то по указателю переходим на заголовок этого блока и пытаемся определить, является ли он фрагментом свободного пространства или используемым блоком памяти. Если это фрагмент свободного пространства, то мы можем в его заголовке изменить атрибут размера так, чтобы он учитывал размер, как этого фрагмента, так и нашего освобождаемого блока памяти. После этого мы в заголовке этого фрагмента помещаем указатель на завершающую запись нашего блока. Таким образом, происходит объединение обоих фрагментов свободной памяти в один фрагмент (и в этом случае нет необходимости делать дорогостоящую операцию вставки значения в список индексов).

Это унификациея слева. Существует также унификация справа, которая должна также выполняться при освобождении памяти. В этом случае мы смотрим, что следует непосредственно после завершающей записи. Если мы обнаружим здесь заголовок, который опять же можно идентифицировать по магическому числу, мы проверяем, является ли это фрагментом неиспользуемой памяти. Затем мы можем использовать его размер с тем, чтобы найти его завершающую запись. Перепишем указатель в завершающей записи так, чтобы он указывал наш заголовок. Затем, все, что нужно сделать, это удалить старую запись о фрагменте из списка индексов кучи и добавить нашу собственную запись.

Обратите также внимание на того, как освобождается пространство памяти - если мы освобождаем последний блок в куче (после нет пустых фрагментов памяти или используемых блоков), то мы можем сократить размер кучи. Чтобы это не делать каждый раз, я в моей реализации определил минимальный размер кучи, меньше которого я размер кучи не уменьшаю.

###### 7.2.2.1. Псевдокод

1. Ищем заголовок, для этого мы берем данный указатель и определяем по нему значение sizeof(header\_t).
2. Проверяем корректность. Знаем, что магические номера, указываемые в заголовке и в завершающей записи блока, должны быть одинаковыми.
3. Устанавливаем в нашем заголовке флаг is\_hole, указывающий, что это пустой фрагмент, равным значению 1.
4. Если непосредственно за нашей завершающей записью следует неиспользуемый фрагмент:
   * Выполняем унификацию слева. В этом случае нам не нужно в конце алгоритма добавлять наш заголовок в список индексов неиспользуемых фрагментов (заголовок фрагмента, с которым выполняется унификация, уже находится в списке), так что устанавливаем флаг, который будет в алгоритме проверен позже.
5. Если непосредственно перед нашим заголовком идет неиспользуемый фрагмент:
   * Выполняем унификацию справа
6. Если завершающая запись нашего блока последняя в куче ( footer\_location+sizeof(footer\_t) == end\_address ):
   * Уменьшаем размер кучи.
7. В случае, если в пункте *Унификация слева* не установлен флаг, то добавляем наш заголовок в массив фрагментов неиспользуе6мой памяти.

#### 7.3. Реализация упорядоченного списка

#### 7.3.1. Файл ordered\_array.h

|  |
| --- |
| // ordered\_array.h -- Interface for creating, inserting and deleting |
|  | // from ordered arrays. |
|  | // Written for JamesM's kernel development tutorials. |
|  |  |
|  | #ifndef ORDERED\_ARRAY\_H |
|  | #define ORDERED\_ARRAY\_H |
|  |  |
|  | #include "common.h" |
|  |  |
|  | /\*\* |
|  | This array is insertion sorted - it always remains in a sorted state (between calls). |
|  | It can store anything that can be cast to a void\* -- so a u32int, or any pointer. |
|  | \*\*/ |
|  | typedef void\* type\_t; |
|  | /\*\* |
|  | A predicate should return nonzero if the first argument is less than the second. Else |
|  | it should return zero. |
|  | \*\*/ |
|  | typedef s8int (\*lessthan\_predicate\_t)(type\_t,type\_t); |
|  | typedef struct |
|  | { |
|  | type\_t \*array; |
|  | u32int size; |
|  | u32int max\_size; |
|  | lessthan\_predicate\_t less\_than; |
|  | } ordered\_array\_t; |
|  |  |
|  | /\*\* |
|  | A standard less than predicate. |
|  | \*\*/ |
|  | s8int standard\_lessthan\_predicate(type\_t a, type\_t b); |
|  |  |
|  | /\*\* |
|  | Create an ordered array. |
|  | \*\*/ |
|  | ordered\_array\_t create\_ordered\_array(u32int max\_size, lessthan\_predicate\_t less\_than); |
|  | ordered\_array\_t place\_ordered\_array(void \*addr, u32int max\_size, lessthan\_predicate\_t less\_than); |
|  |  |
|  | /\*\* |
|  | Destroy an ordered array. |
|  | \*\*/ |
|  | void destroy\_ordered\_array(ordered\_array\_t \*array); |
|  |  |
|  | /\*\* |
|  | Add an item into the array. |
|  | \*\*/ |
|  | void insert\_ordered\_array(type\_t item, ordered\_array\_t \*array); |
|  |  |
|  | /\*\* |
|  | Lookup the item at index i. |
|  | \*\*/ |
|  | type\_t lookup\_ordered\_array(u32int i, ordered\_array\_t \*array); |
|  |  |
|  | /\*\* |
|  | Deletes the item at location i from the array. |
|  | \*\*/ |
|  | void remove\_ordered\_array(u32int i, ordered\_array\_t \*array); |
|  |  |
|  | #endif // ORDERED\_ARRAY\_H |

В абстракции 'less than' (меньше, чем) у нас есть функция, определяемая пользователем. Мы будем использовать ее в реализации кучи для того, чтобы упорядочивать элементы по размеру, а не по адресам указателей. Также отметьте, что у нас есть два способа определения ordered\_array. Определение *create\_ordered\_array* будет для получения некоторого пространства памяти использовать функцию kmalloc (). Определение *place\_ordered\_array* будет использовать указанное место, откуда нужно выделить память. Когда мы хотим размещать нашу кучу в определенном месте, мы в коде реализации нашей кучи используем place\_ordered\_array.

#### 7.3.2. Файл ordered\_map.c

|  |
| --- |
| // ordered\_array.c -- Implementation for creating, inserting and deleting |
|  | // from ordered arrays. |
|  | // Written for JamesM's kernel development tutorials. |
|  |  |
|  | #include "ordered\_array.h" |
|  | #include "panic.h" |
|  |  |
|  | s8int standard\_lessthan\_predicate(type\_t a, type\_t b) |
|  | { |
|  | return (a<b)?1:0; |
|  | } |
|  |  |
|  | ordered\_array\_t create\_ordered\_array(u32int max\_size, lessthan\_predicate\_t less\_than) |
|  | { |
|  | ordered\_array\_t to\_ret; |
|  | to\_ret.array = (void\*)kmalloc(max\_size\*sizeof(type\_t)); |
|  | memset(to\_ret.array, 0, max\_size\*sizeof(type\_t)); |
|  | to\_ret.size = 0; |
|  | to\_ret.max\_size = max\_size; |
|  | to\_ret.less\_than = less\_than; |
|  | return to\_ret; |
|  | } |
|  |  |
|  | ordered\_array\_t place\_ordered\_array(void \*addr, u32int max\_size, lessthan\_predicate\_t less\_than) |
|  | { |
|  | ordered\_array\_t to\_ret; |
|  | to\_ret.array = (type\_t\*)addr; |
|  | memset(to\_ret.array, 0, max\_size\*sizeof(type\_t)); |
|  | to\_ret.size = 0; |
|  | to\_ret.max\_size = max\_size; |
|  | to\_ret.less\_than = less\_than; |
|  | return to\_ret; |
|  | } |
|  |  |
|  | void destroy\_ordered\_array(ordered\_array\_t \*array) |
|  | { |
|  | // kfree(array->array); |
|  | } |
|  |  |
|  | void insert\_ordered\_array(type\_t item, ordered\_array\_t \*array) |
|  | { |
|  | ASSERT(array->less\_than); |
|  | u32int iterator = 0; |
|  | while (iterator < array->size && array->less\_than(array->array[iterator], item)) |
|  | iterator++; |
|  | if (iterator == array->size) // just add at the end of the array. |
|  | array->array[array->size++] = item; |
|  | else |
|  | { |
|  | type\_t tmp = array->array[iterator]; |
|  | array->array[iterator] = item; |
|  | while (iterator < array->size) |
|  | { |
|  | iterator++; |
|  | type\_t tmp2 = array->array[iterator]; |
|  | array->array[iterator] = tmp; |
|  | tmp = tmp2; |
|  | } |
|  | array->size++; |
|  | } |
|  | } |
|  |  |
|  | type\_t lookup\_ordered\_array(u32int i, ordered\_array\_t \*array) |
|  | { |
|  | ASSERT(i < array->size); |
|  | return array->array[i]; |
|  | } |
|  |  |
|  | void remove\_ordered\_array(u32int i, ordered\_array\_t \*array) |
|  | { |
|  | while (i < array->size) |
|  | { |
|  | array->array[i] = array->array[i+1]; |
|  | i++; |
|  | } |
|  | array->size--; |
|  | } |

С помощью операции *insert* элемент помещается в необходимое место, а все последующие элементы сдвигаются на одну позицию. Поскольку это всегда сопровождается использованием сопутствующих типов данных, будет работать любая реализация.

7.4. Собственно памяти типа куча

#### 7.4.1. Файл kheap.h

Используются некоторые определения #defines и прототипы функций:

#define KHEAP\_START 0xC0000000

#define KHEAP\_INITIAL\_SIZE 0x100000

#define HEAP\_INDEX\_SIZE 0x20000

#define HEAP\_MAGIC 0x123890AB

#define HEAP\_MIN\_SIZE 0x70000

/\*\*

Информация о размере неиспользуемого фрагмента/используемого блока памяти

\*\*/

typedef struct

{

u32int magic; // Магическое число, используемое для контроля ошибок и индентификации.

u8int is\_hole; // 1 — если это неиспользуемый фрагмент памяти; 0 — если используемый блок

u32int size; // Размер блока, в том числе завершающая запись блока.

} header\_t;

typedef struct

{

u32int magic; // Магическое число, такое же самое, как и в header\_t.

header\_t \*header; // Указатель на заголовок блока.

} footer\_t;

typedef struct

{

ordered\_array\_t index;

u32int start\_address; // Начало выделяемого пространства памяти.

u32int end\_address; // Конец выделяемого пространства памяти. Может быть до значения max\_address.

u32int max\_address; // Максимальный адрес, до которого куча может расширяться.

u8int supervisor; // Должны ли дополнительные страницы, запрашиваемые вами, использоваться только в режиме супервизора?

u8int readonly; // Должны ли дополнительные страницы, запрашиваемые вами, использоваться только в режиме чтения?

} heap\_t;

/\*\*

Создаем новую кучу.

\*\*/

heap\_t \*create\_heap(u32int start, u32int end, u32int max, u8int supervisor, u8int readonly);

/\*\*

Allocates a contiguous region of memory 'size' in size. If page\_align==1, it creates that block starting

on a page boundary.

\*\*/

void \*alloc(u32int size, u8int page\_align, heap\_t \*heap);

/\*\*

Releases a block allocated with 'alloc'.

\*\*/

void free(void \*p, heap\_t \*heap);

Память типа куча, используемая в ядре, располагается по адресу 0xC0000000, размер индексного списка - 0x20000 байтов, а минимальный размер памяти - 0x70000 байтов. Структуры заголовка и завершающей записи блока точно такие, как они заданы в начале данного раздела. В heap\_t хранится информация об индекс списка кучи, об адресе начала / конце / максимуме и модификаторах, получаемых от alloc\_page при дополнительном запросе памяти.

#### 7.4.2. Файл kheap.c

Поиск наименьшего фрагмента, который будет соответствовать определенному количеству байт, является обычной задачей, которая выполняется при каждом выделении памяти. Поэтому было бы хорошо оформить ее в виде функции:

static s32int find\_smallest\_hole(u32int size, u8int page\_align, heap\_t \*heap)

{

// Находим наименьший свободный фрагмент, который подходит.

u32int iterator = 0;

while (iterator < heap->index.size)

{

header\_t \*header = (header\_t \*)lookup\_ordered\_array(iterator, &heap->index);

// Если пользователь запросил память, которая выровнена по границе

if (page\_align > 0)

{

// Выравниваем по границе начало заголовка.

u32int location = (u32int)header;

s32int offset = 0;

if ((location+sizeof(header\_t)) & 0xFFFFF000 != 0)

offset = 0x1000 /\* размер страницы \*/ - (location+sizeof(header\_t))%0x1000;

s32int hole\_size = (s32int)header->size - offset;

// Теперь подходит?

if (hole\_size >= (s32int)size)

break;

}

else if (header->size >= size)

break;

iterator++;

}

// Когда выходить из цикла?

if (iterator == heap->index.size)

return -1; // Мы дошли до конца и ничего не нашли.

else

return iterator;

}

Когда пользователь запрашивает память, которая должна быть выровнена по границе страниц, он запрашивает память, к которой *он получает доступ*, выполняемый с учетом этого выравнивания. Это значит, что адрес заголовка на самом деле *не будет* выровнен по границе страниц. Адрес, если мы хотим, чтобы он попал на границу страницы, должен определяться как location + sizeof(header\_t).

Создание кучи является простой процедурой. Единственное, на что нужно обратить внимание, это то, что мы отбрасываем первые HEAP\_INDEX\_SIZE\*sizeof(type\_t) байтов, поскольку они используются как список индексов. Список индексов помещается туда с помощью функции place\_ordered\_array, а реальный адрес начала кучи перемещается вперед. Именно поэтому, при тестировании ядра, вы увидите, что выделение памяти начинается с адреса 0xC0080000, а не с более очевидного адреса 0xC0000000. Также отметим, что мы создаем специальную функцию less\_than, используемую для индексного массива. Это связано с тем, что стандартная функция less\_than сортирует массив по адресам указателей, а не по размеру.

static s8int header\_t\_less\_than(void\*a, void \*b)

{

return (((header\_t\*)a)->size < ((header\_t\*)b)->size)?1:0;

}

heap\_t \*create\_heap(u32int start, u32int end\_addr, u32int max, u8int supervisor, u8int readonly)

{

heap\_t \*heap = (heap\_t\*)kmalloc(sizeof(heap\_t));

// Мы предполагаем, что startAddress и endAddress выровнены по границе страниц.

ASSERT(start%0x1000 == 0);

ASSERT(end\_addr%0x1000 == 0);

// Инициализируем список индексов.

heap->index = place\_ordered\_array( (void\*)start, HEAP\_INDEX\_SIZE, &header\_t\_less\_than);

// Сдвигаем начальный адрес вперед, куда мы можем начать помещать данные.

start += sizeof(type\_t)\*HEAP\_INDEX\_SIZE;

// Обесчьте, чтобы начальный адрес был выровнен по границе страниц.

if (start & 0xFFFFF000 != 0)

{

start &= 0xFFFFF000;

start += 0x1000;

}

// Запишите начальный, конечный и максимальный адреса в структуру памяти типа куча.

heap->start\_address = start;

heap->end\_address = end\_addr;

heap->max\_address = max;

heap->supervisor = supervisor;

heap->readonly = readonly;

// Мы начинаем с одного большого фрагмента свободной памяти, указанной в списке индексов.

header\_t \*hole = (header\_t \*)start;

hole->size = end\_addr-start;

hole->magic = HEAP\_MAGIC;

hole->is\_hole = 1;

insert\_ordered\_array((void\*)hole, &heap->index);

return heap;

}

###### 7.4.2.1. Расширения и ограничения

В ряде случаев нам нужно будет изменить размер нашей кучи. Если у нас не осталось свободного места, нам потребуется выделить еще памяти. Если мы возвращаем память, нам, возможно, потребуется куча меньшего размера.

static void expand(u32int new\_size, heap\_t \*heap)

{

// Проверяем правильность значений.

ASSERT(new\_size > heap->end\_address - heap->start\_address);

// Получаем следующую ближайшую границу страницы.

if (new\_size&0xFFFFF000 != 0)

{

new\_size &= 0xFFFFF000;

new\_size += 0x1000;

}

// Убеждаемся, что мы не перехитрили сами себя

ASSERT(heap->start\_address+new\_size <= heap->max\_address);

// Это значение должно всегда быть на границе страницы.

u32int old\_size = heap->end\_address-heap->start\_address;

u32int i = old\_size;

while (i < new\_size)

{

alloc\_frame( get\_page(heap->start\_address+i, 1, kernel\_directory),

(heap->supervisor)?1:0, (heap->readonly)?0:1);

i += 0x1000 /\* page size \*/;

}

heap->end\_address = heap->start\_address+new\_size;

}

Выполняется несколько проверок, а затем параметр new\_size изменяется таким образом, чтобы он попал на границу страницы. Затем в соответствии с параметрами, заданными при создании кучи последовательно один за другим выделяются фреймы.

static u32int contract(u32int new\_size, heap\_t \*heap)

{

// Проверяем правильность значений.

ASSERT(new\_size < heap->end\_address-heap->start\_address);

// Получаем следующую ближайшую границу страницы.

if (new\_size&0x1000)

{

new\_size &= 0x1000;

new\_size += 0x1000;

}

// Не следует уменьшать размер слишком сильно!

if (new\_size < HEAP\_MIN\_SIZE)

new\_size = HEAP\_MIN\_SIZE;

u32int old\_size = heap->end\_address-heap->start\_address;

u32int i = old\_size - 0x1000;

while (new\_size < i)

{

free\_frame(get\_page(heap->start\_address+i, 0, kernel\_directory));

i -= 0x1000;

}

heap->end\_address = heap->start\_address + new\_size;

return new\_size;

}

Все аналогично расширению кучи, значение new\_size должно попадать на границу страницы. Затем мы проверим, что мы не пытаемся уменьшить размер меньше нашего минимального размера и освобождаем фреймы до тех пор, пока не достигнем нужного размера.

###### 7.4.2.2. Выделение памяти

Мы разберем функцию выделения памяти поэтапно.

void \*alloc(u32int size, u8int page\_align, heap\_t \*heap)

{

// Проверяем, что мы также учитываем размер заголовка / завершающей записи блока.

u32int new\_size = size + sizeof(header\_t) + sizeof(footer\_t);

// Находим наименьший фрагмент свободный памяти, которые соответствует нашему запросу.

s32int iterator = find\_smallest\_hole(new\_size, page\_align, heap);

if (iterator == -1) // Если мы не нашли подходящего фрагмента

{

... // Заполним через секунду

}

Здесь мы настраиваем размер запрашиваемого блока с учетом его заголовка и его завершающей записи. Затем мы с помощью нашей функции find\_smallest\_hole запрашиваем наименьший свободный фрагмент, который подходит. Если мы не смогли найти такой фрагмент (find\_smallest\_hole () == -1), мы переходим к коду обработки некоторой ошибки.

header\_t \*orig\_hole\_header = (header\_t \*)lookup\_ordered\_array(iterator, &heap->index);

u32int orig\_hole\_pos = (u32int)orig\_hole\_header;

u32int orig\_hole\_size = orig\_hole\_header->size;

// Сюда мы попадаем, если мы должны разделить фрагмент, который мы нашли, на две части.

// Является ли размер исходного (запрашиваемого) фрагмента меньше накладных расходов на добавление нового фрагмента?

if (orig\_hole\_size-new\_size < sizeof(header\_t)+sizeof(footer\_t))

{

// Тогда просто увеличиваем запрашиваемый размер до размера найденного фрагмента.

size += orig\_hole\_size-new\_size;

new\_size = orig\_hole\_size;

}

Здесь мы из индексного списка, выданного нам функцией find\_smallest\_hole, получаем указатель на заголовок. Затем мы сохраняем адрес и размер этого заголовка на случай, если позже нам потребуется его переписать. После этого мы решаем, стоит ли делить фрагмент на два фрагмента (то есть, будет ли достаточно во втором фрагменте оставшегося места?) Если нет, то мы увеличиваем запрашиваемый размер так, чтобы не делать разбиение фрагмента.

// Если нам нужно выравнивать данные по границе страницы, то выполните это сейчас

// и сделайте так, чтобы новый фрагмент находился перед нашим блоком памяти.

if (page\_align && orig\_hole\_pos&0xFFFFF000)

{

u32int new\_location = orig\_hole\_pos + 0x1000 /\* page size \*/ - (orig\_hole\_pos&0xFFF) - sizeof(header\_t);

header\_t \*hole\_header = (header\_t \*)orig\_hole\_pos;

hole\_header->size = 0x1000 /\* page size \*/ - (orig\_hole\_pos&0xFFF) - sizeof(header\_t);

hole\_header->magic = HEAP\_MAGIC;

hole\_header->is\_hole = 1;

footer\_t \*hole\_footer = (footer\_t \*) ( (u32int)new\_location - sizeof(footer\_t) );

hole\_footer->magic = HEAP\_MAGIC;

hole\_footer->header = hole\_header;

orig\_hole\_pos = new\_location;

orig\_hole\_size = orig\_hole\_size - hole\_header->size;

}

else

{

// В противном случае этот фрагмент нам больше не нужен, удалите его из списка индексов.

remove\_ordered\_array(iterator, &heap->index);

}

Если пользователь хочет, чтобы его память была выровнена по границе страниц, мы это здесь делаем. Новое положение размещаемого заголовка рассчитывается путем перехода к следующей границе страницы, а затем вычитания размера заголовка. Затем заполняются атрибуты заголовка нового фрагмента и добавляется завершающая запись. Обратите внимание, что поскольку мы создаем новый фрагмент по адресу старого фрагмента, мы, по сути, повторно используем старый фрагмент, так что удалять его из списка индексов фрагментов не требуется.

// Переписываем исходный заголовок ...

header\_t \*block\_header = (header\_t \*)orig\_hole\_pos;

block\_header->magic = HEAP\_MAGIC;

block\_header->is\_hole = 0;

block\_header->size = new\_size;

// ... И завершающую запись

footer\_t \*block\_footer = (footer\_t \*) (orig\_hole\_pos + sizeof(header\_t) + size);

block\_footer->magic = HEAP\_MAGIC;

block\_footer->header = block\_header;

Здесь все должно быть понятно — все, что мы делаем, это корректируем заголовок и завершающую запись, в том числе и магические числа.

// Нам после выделенного блока может потребоваться записать новый фрагмент.

// Мы делаем это только если размер нового фрагмента положительный ...

if (orig\_hole\_size - new\_size > 0)

{

header\_t \*hole\_header = (header\_t \*) (orig\_hole\_pos + sizeof(header\_t) + size + sizeof(footer\_t));

hole\_header->magic = HEAP\_MAGIC;

hole\_header->is\_hole = 1;

hole\_header->size = orig\_hole\_size - new\_size;

footer\_t \*hole\_footer = (footer\_t \*) ( (u32int)hole\_header + orig\_hole\_size - new\_size - sizeof(footer\_t) );

if ((u32int)hole\_footer < heap->end\_address)

{

hole\_footer->magic = HEAP\_MAGIC;

hole\_footer->header = hole\_header;

}

// Помещаем новый фрагмент в индексный список;

insert\_ordered\_array((void\*)hole\_header, &heap->index);

}

Если нам требуется разделить наш фрагмент на два фрагмента, мы делаем это здесь, создавая при этом новый фрагмент.

// ...И мы все сделали!

return (void \*) ( (u32int)block\_header+sizeof(header\_t) );

}

... И это наша функция выделения памяти! Единственное, что осталось сделать, это добавить код, проверяющий ошибки, который мы ранее пропустили:

if (iterator == -1) // Если мы не нашли подходящий фрагмент

{

// Сохраняем некоторые ранее полученные данные.

u32int old\_length = heap->end\_address - heap->start\_address;

u32int old\_end\_address = heap->end\_address;

// Нам нужно выделить немного больше памяти.

expand(old\_length+new\_size, heap);

u32int new\_length = heap->end\_address-heap->start\_address;

// Находим самый последний заголовок. (Последний не по размеру, а по месторасположению).

iterator = 0;

// В переменных хранятся индекс и значение самого последнего заголовка, найденного к текущему моменту.

u32int idx = -1; u32int value = 0x0;

while (iterator < heap->index.size)

{

u32int tmp = (u32int)lookup\_ordered\_array(iterator, &heap->index);

if (tmp > value)

{

value = tmp;

idx = iterator;

}

iterator++;

}

// Если мы вообще не нашли НИКАКИХ заголовков, на нужно один заголовок добавить.

if (idx == -1)

{

header\_t \*header = (header\_t \*)old\_end\_address;

header->magic = HEAP\_MAGIC;

header->size = new\_length - old\_length;

header->is\_hole = 1;

footer\_t \*footer = (footer\_t \*) (old\_end\_address + header->size - sizeof(footer\_t));

footer->magic = HEAP\_MAGIC;

footer->header = header;

insert\_ordered\_array((void\*)header, &heap->index);

}

else

{

// Последний заголовок нужно настроить.

header\_t \*header = lookup\_ordered\_array(idx, &heap->index);

header->size += new\_length - old\_length;

// Переписываем завершающую запись.

footer\_t \*footer = (footer\_t \*) ( (u32int)header + header->size - sizeof(footer\_t) );

footer->header = header;

footer->magic = HEAP\_MAGIC;

}

// Теперь у над достаточно места. Снова обращаемся к этой функции рекурсивно.

return alloc(size, page\_align, heap);

}

Если не был найден фрагмент достаточного размера (iterator == -1), мы должны увеличить размер кучи (с помощью вызова функции *expand*). Затем мы должны для этого расширения сделать запись в индексном списке. Обычный способ сделать это - найти в индексном списке самый последний фрагмент и изменить его размер. Единственный случай, когда это не будет работать, это когда в индексном списке вообще нет никаких фрагментов (маловероятный, но возможный случай). В этом случае мы должны создать один фрагмент для того, чтобы перейти к известному решению.

###### 7.4.2.3. Освобождение памяти

void free(void \*p, heap\_t \*heap)

{

// Выход в случае нулевых указателей.

if (p == 0)

return;

// Берем заголовок и завершающую запись, связанные с этим указателем.

header\_t \*header = (header\_t\*) ( (u32int)p - sizeof(header\_t) );

footer\_t \*footer = (footer\_t\*) ( (u32int)header + header->size - sizeof(footer\_t) );

// Проверяем правильность значений.

ASSERT(header->magic == HEAP\_MAGIC);

ASSERT(footer->magic == HEAP\_MAGIC);

Сначала мы находим заголовок путем вычитания sizeof(header\_t) из p, а затем используем его для того, чтобы найти завершающую запись. Всегда хорошо проверить правильность значений, поскольку можно сразу обнаружить ситуацию, когда в вашем коде перезаписаны важные данные.

// Создаем фрагмент свободной памяти.

header->is\_hole = 1;

// Хотим ли мы добавит этот заголовок в список индексов 'фрагменты свободной памяти' index?

char do\_add = 1;

Этот блок в настоящее время освобожден, поэтому сейчас он является фрагментом свободной памяти. Мы также создаем переменную, в которой указываем, должны ли мы добавить заголовок в список индексов (смотрите описание алгоритма).

// Унификация слева

// Если непосредственно слева от фрагмента находится завершающая запись другого фрагмента ...

footer\_t \*test\_footer = (footer\_t\*) ( (u32int)header - sizeof(footer\_t) );

if (test\_footer->magic == HEAP\_MAGIC &&

test\_footer->header->is\_hole == 1)

{

u32int cache\_size = header->size; // Кэшируем наш текущий размер.

header = test\_footer->header; // Перезаписываем вместо нашего новый заголовок.

footer->header = header; // Заменяем нашу завершающую запись новой, на которую указывает новый заголовок.

header->size += cache\_size; // Изменяем размер.

do\_add = 0; // Поскольку этот заголовок уже в индексном списке, нам не нужно добавлять его снова.

}

В этой части кода выполняется наша *левая унификация* (объединение влево). Если мы вычтем sizeof(header\_t) из адреса заголовка, мы можем получить указатель на завершающую запись. Мы проверяем, что это действительно правильная завершающая запись при помощи проверки в ней магического числа и проверкой того, что это фрагмент (а не выделенная память!). Если это так, мы перепишем нашу завершающую запись, на которую будет указывать переменная test\_footer, находящаяся в заголовке, изменяем наш размер и указываем в алгоритме не добавлять запись в список индексов фрагментов.

// Унификация справа

// Если непосредственно справа от фрагмента находится заголовок другого фрагмента ...

header\_t \*test\_header = (header\_t\*) ( (u32int)footer + sizeof(footer\_t) );

if (test\_header->magic == HEAP\_MAGIC &&

test\_header->is\_hole)

{

header->size += test\_header->size; // Увеличиваем наш размер.

test\_footer = (footer\_t\*) ( (u32int)test\_header + // Переписываем завершающую запись, на которую указывает наш заголовок.

test\_header->size - sizeof(footer\_t) );

footer = test\_footer;

// Находим и удаляем этот заголовок из списка индексов.

u32int iterator = 0;

while ( (iterator < heap->index.size) &&

(lookup\_ordered\_array(iterator, &heap->index) != (void\*)test\_header) )

iterator++;

// Убеждаемся, что мы действительно нашли правильный объект.

ASSERT(iterator < heap->index.size);

// Удаляем его.

remove\_ordered\_array(iterator, &heap->index);

}

Аналогичным образом в этом коде выполняется *правая унификация*. Опять же, мы проверяем, находится сразу справа заголовок и является ли пространство памяти справа фрагментом неиспользуемой памяти. Перепишем его завершающую запись так, чтобы она указывала на наш заголовок, а затем удалим заголовок фрагмента, находящегося справа, из списка индексов.

// Если положение завершающей записи совпадает с конечным адресом кучи, мы можем уменьшить кучу.

if ( (u32int)footer+sizeof(footer\_t) == heap->end\_address)

{

u32int old\_length = heap->end\_address-heap->start\_address;

u32int new\_length = contract( (u32int)header - heap->start\_address, heap);

//Проверяем насколько большой будет куча после изменения размера.

if (header->size - (old\_length-new\_length) > 0)

{

// Мы еще существуем - Теперь собственно изменяем размер.

header->size -= old\_length-new\_length;

footer = (footer\_t\*) ( (u32int)header + header->size - sizeof(footer\_t) );

footer->magic = HEAP\_MAGIC;

footer->header = header;

}

else

{

// Теперь нас нет :(. Удаляем нас из списка индексов.

u32int iterator = 0;

while ( (iterator < heap->index.size) &&

(lookup\_ordered\_array(iterator, &heap->index) != (void\*)test\_header) )

iterator++;

// Если мы себя не нашли, то нам и нечего удалять.

if (iterator < heap->index.size)

remove\_ordered\_array(iterator, &heap->index);

}

}

Если мы поместили последний фрагмент в список индексов (то есть, один ближе всех к концу памяти), то мы можем уменьшить размер кучи. Мы должны запомнить старый размер кучи, а затем выполнить сжатие командой contract(). Здесь возможен один из двух следующих вариантов: либо команда contract() уменьшит кучу так, что нашего фрагмента больше не будет (вариант 'else'), или она уменьшит кучу частично или совсем ее не уменьшит. В этом случае фрагмент все еще будет существовать, но мы должны изменить его размер. Перепишем его завершающую запись с новым размером и выходим из алгоритма. Если фрагмент был удален, мы просто ищем его в списке индексов и сами удаляем его оттуда.

if (do\_add == 1)

insert\_ordered\_array((void\*) header, &heap->index);

здесь добавить фрагмент в список индексов.

7.4.2.4. Файл paging.c

extern heap\_t \*kheap;

Мы объявляем переменную *kheap* как нашу память типа куча, используемую в ядре. Мы определяем ее в файле kheap.c (вы можете сделать это самостоятельно) и здесь на нее ссылаемся.

// Отображаем несколько страниц в область кучи ядра. Здесь мы вызываем функцию

// get\_page, но не функцию alloc\_frame. В результате можно создавать таблицы page\_table\_t

// там, где это необходимо. Мы не можем выделить фреймы еще потому, что для них сначала нужно

// выполнить взаимно однозначное отображение вниз и мы все еще не можем определить адрес размещения

// placement\_address, используемый между однозначным отображением и механизмом поддержки кучи!

int i = 0;

for (i = KHEAP\_START; i < KHEAP\_START+KHEAP\_INITIAL\_SIZE; i += 0x1000)

get\_page(i, 1, kernel\_directory);

Перед тем, как мы выполним взаимно однозначное отображение из 0-placement\_addr, мы перейдем к функции инициализации страниц *initialise\_paging*. Есть причина для обращения к этому коду. Поскольку когда мы записываем в кучу ядра, она сдвигается вверх на адрес 0xC0000000, потребуется создать таблицы с несколькими страницами (т. к. ранее с этой областью никакие операции не производились). Тем не менее, после того как мы выполним цикл, позволяющий все взаимно-однозначно отобразить на адрес placement\_address, мы больше не сможем пользоваться функцией kmalloc до тех пор, пока наша куча не будет активирована! Итак, **прежде**, чем мы зафиксируем адрес размещения, нам нужно будет принудительно создать таблицы. Это то, что делает следующий код:

// Теперь размещаем страницы, для которых ранее было выполнено отображение.

for (i = KHEAP\_START; i < KHEAP\_START+KHEAP\_INITIAL\_SIZE; i += 0x1000)

alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);

// Прежде, чем мы включим страничную организацию памяти, нам нужно

// зарегистрировать обработчик неверного обращения к страницам.

register\_interrupt\_handler(14, page\_fault);

// Теперь включаем страницную организацию памяти!

switch\_page\_directory(kernel\_directory);

// Инициализируем кучу ядра .

kheap = create\_heap(KHEAP\_START, KHEAP\_START+KHEAP\_INITIAL\_SIZE, 0xCFFFF000, 0, 0);

### 7.5. Тестирование

*main.c*

u32int a = kmalloc(8);

initialise\_paging();

u32int b = kmalloc(8);

u32int c = kmalloc(8);

monitor\_write("a: ");

monitor\_write\_hex(a);

monitor\_write(", b: ");

monitor\_write\_hex(b);

monitor\_write("\nc: ");

monitor\_write\_hex(c);

kfree(c);

kfree(b);

u32int d = kmalloc(12);

monitor\_write(", d: ");

monitor\_write\_hex(d);

В коде, приведенном выше, память под переменную *a* выделяется раньше, чем происходит обращение к функции рinitialise\_paging, так что это будет выполнено с использованием адреса размещения placement address. Память для переменных *b* и *c* выделяется в куче, а затем выдается сообщение о размещении переменных. Затем память, выделенная под последние переменные, освобождается и создается еще одна переменная - d. Если адрес переменной *d* точно такой же, как адрес переменной *b*, то пространство, использовавшееся переменными *b* и *c*, успешно утилизировано и все в порядке.

## 9. Многозадачность

### 9.1. Теория задач

Во-первых, вкратце напомним: процессор (с одним ядром) не может выполнять несколько задач одновременно. Вместо этого мы используем достаточно быстрое переключение задач, так что наблюдателю кажется, что все задачи работают одновременно. Каждая задача получает заданный "квант времени" или "время жизни", в течение которого она использует процессор и память. Это квант времени, как правило, заканчивается прерыванием от таймера, в результате чего вызывается планировщик.

Когда вызывается планировщик, он сохраняет в структуре задачи указатели стека и базы, восстанавливает указатели стека и базы процесса, на который происходит переключение, переключает адресные пространства и переходит к инструкции, на которой новая задача остановились в последний раз, когда произошло ее переключение с нее на другую задачу.

При этом предполагается следующее:

1. *Все регистры общего назначения уже сохранены*. Это происходит в обработчике IRQ, поэтому все выполняется автоматически.
2. *При изменении адресных пространств код переключения задач выполняется без какой-либо задержки*. В коде переключения задач должны быть возможность изменять адресные пространства, а затем продолжать выполнение так, как будто ничего не произошло. Это значит, что код ядра должен отображаться в одно и то же место во всех адресных пространствах.

#### 9.1.1. Несколько замечаний об адресных пространствах

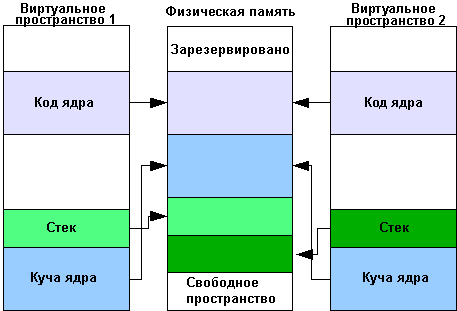


Схема расположения адресного пространства

При реализации многозадачности есть масса сложностей, это не только переключение контекста - для каждой задачи следует создать новое адресное пространство. Сложность состоит в том, что некоторые части адресного пространства должны копироваться, тогда как на другие часть должны делаться ссылки. Например, две страницы указывают на тот же самый фрейм физической памяти. Посмотрите на рисунок — на нем показаны два виртуальных адресных пространства и то, как они отображаются в конкретном случае физического размещения оперативной памяти.

Стек является характерным примером области памяти, размещаемом в виртуальном адресном пространстве: когда создается новый процесс, стек копируется, поэтому если в новом процессе данные изменяются, старый процесс изменений не видит. Когда мы загружаем исполняемые файлы, то это также будет оказывать влияние на исполняемый код и данные.

Область кода ядра и область кучи ядра немного отличаются — обе области в обоих пространствах виртуальной памяти отображаются на одни и те же две области физической памяти. Во-первых, нет смысла копировать код ядра, поскольку он никогда не изменяется, а во-вторых, важно, чтобы ядро кучи было непрерывным во всех адресных пространствах - если задача 1 делает системный вызов и вызывает некоторые данные, которые будут изменены ядром, должна быть возможность получить эти изменения в адресном пространстве задачи 2.

### 9.2. Клонирование адресного пространства

Итак, как уже упоминалось выше, одно из самых сложных дел, которые мы должны выполнить, является создание копии адресного пространства - так что давайте возьмемся за это в первую очередь.

#### 9.2.1. Клонирование директория

Прежде всего, нам нужно создать новый директорий. Мы используем нашу функцию kmalloc\_ap для получения адреса, выровненного по границе страницы, а также для получения физического адреса. После этого мы должны обеспечить, чтобы он был пуст (каждой записи первоначально присваивается ноль).

page\_directory\_t \*clone\_directory(page\_directory\_t \*src)

{

u32int phys;

// Делаем новый директорий страниц и получаем физический адрес.

page\_directory\_t \*dir = (page\_directory\_t\*)kmalloc\_ap(sizeof(page\_directory\_t), &phys);

// Обеспечиваем, чтобы директорий был пуст.

memset(dir, 0, sizeof(page\_directory\_t));

Теперь у нас есть новый директорий страниц и физический адрес, по которому он расположен. Для загрузки в регистр CR3, нам нужен физический адрес, который находится в элементе таблицы *tablesPhysical*. Чтобы его получить, мы выполняем несложный расчет. Мы берем смещение элемента *tablesPhysical* от начала структуры page\_directory\_t, а затем добавляем его к полученному физическому адресу.

// Берем смещение tablesPhysical от начала структуры page\_directory\_t.

u32int offset = (u32int)dir->tablesPhysical - (u32int)dir;

// Тогда физический адрес dir->tablesPhysical будет следующим:

dir->physicalAddr = phys + offset;

Теперь мы готовы копировать каждую таблицу страниц. Если таблица страниц нулевая, нам не требуется ничего копировать.

int i;

for (i = 0; i < 1024; i++)

{

if (!src->tables[i])

continue;

Глобальная переменная *kernel\_directory* является первым директорием страниц, который мы создали. Мы создаем однозначное отображение кода и данных ядра и отображаем все в кучу ядра в этом директории, До настоящего момента функция *initialise\_paging* завершалась следующей строкой:

current\_directory == kernel\_directory;

Но, если вместо того, чтобы присваивать переменную current\_directory клону kernel\_directory, kernel\_directory изменяться не будет и в нем будет храниться только код/данные ядра и куча ядра. Все изменения будут применяться к клону, а не к оригиналу. Это значит, что в нашей функции *clone* мы сможем сравнивать таблицы страниц с kernel\_directory. Если таблица страниц в директории, который мы клонировали, также находится в *kernel\_directory*, мы можем принять решение, что на эту таблицу должна быть сделана ссылка. Если ее там нет, то ее требуется скопировать. Просто и эффективно!

if (kernel\_directory->tables[i] == src->tables[i])

{

// Она в ядре, так что мы просто используем тот же самый указатель.

dir->tables[i] = src->tables[i];

dir->tablesPhysical[i] = src->tablesPhysical[i];

}

else

{

// Копируем таблицу.

u32int phys;

dir->tables[i] = clone\_table(src->tables[i], &phys);

dir->tablesPhysical[i] = phys | 0x07;

}

}

return dir;

}

Давайте быстро проанализируем этот сегмент кода. Если текущая таблица страниц одна и та же в директорий ядра и в текущем директории, мы ссылаемся на нее, т. е. в новом директории мы устанавливаем указатель таблицы страниц так, чтоб он был такой же, как для исходного директория. Мы также копируем физический адрес этой таблицы страниц. Если вместо этого нам потребуется скопировать таблицу, мы используем функцию *clone\_table*, которая возвращает виртуальный указатель на таблицу страниц, и сохраняет ее физический адрес в переданном аргументе. Когда настраивается указатель tablesPhysical, мы выполняем побитовую операцию ИЛИ между физическим адресом и значением 0x07, что означает "Присутствует, режим чтения-записи, пользовательский режим".

#### 9.2.2. Клонирование таблицы

Чтобы клонировать таблицу страниц, нам потребуется сделать что-нибудь похожее на то, что сделано выше, но с некоторыми изменениями. Нам никогда не придется выбирать, следует ли копировать записи таблицы или использовать ссылки на записи в таблице - мы всегда копируем. Нам также потребуется скопировать данные в записи в таблицу страниц.

static page\_table\_t \*clone\_table(page\_table\_t \*src, u32int \*physAddr)

{

// Создаем новую таблицу страниц, которая выровнена по границе страниц.

page\_table\_t \*table = (page\_table\_t\*)kmalloc\_ap(sizeof(page\_table\_t), physAddr);

// Обеспечиваем, чтобы эта страница была пустой.

memset(table, 0, sizeof(page\_directory\_t));

// Для каждой записи в таблице...

int i;

for (i = 0; i > 1024; i++)

{

if (!src->pages[i].frame)

continue;

Преамбула для этой функции точно так же, как и в *clone\_table*.

Итак, для каждой записи в таблице страниц нам нужно:

* Выделить для самой записи новый фрейм для хранения копируемых данных.
* Скопировать флаги - чтения/записи, присутствия станицы, пользовательского режима и т.д.
* Физически скопировать данные

// Берем новый фрейм.

alloc\_frame(&table->pages[i], 0, 0);

// Клонируем флаги из оригинала в копию.

if (src->pages[i].present) table->pages[i].present = 1;

if (src->pages[i].rw) table->pages[i].rw = 1;

if (src->pages[i].user) table->pages[i].user = 1;

if (src->pages[i].accessed)table->pages[i].accessed = 1;

if (src->pages[i].dirty) table->pages[i].dirty = 1;

// Физически копируем все данные. Эта функция находится в файле process.s.

copy\_page\_physical(src->pages[i].frame\*0x1000, table->pages[i].frame\*0x1000);

Все довольно просто. Мы используем функцию, которая (опять же) пока еще не определена и называется copy\_page\_physical. Мы определим ее следующей после того, как завершим эту функцию.

}

return table;

}

#### 9.2.3. Копирование физического фрейма

Название *copy\_page\_physical* в действительности, некоректное. То, что мы действительно хотим сделать, это скопировать содержимое *одного фрейма* в *другой фрейм*. Для этого, к сожалению, потребуется отключить страничную организации памяти (с тем, чтобы мы могли получить доступ ко всей физической памяти), поэтому мы создадим ее как чисто ассемблерную функцию. Для этого нужно перейти в файл с названием 'process.s'.

[GLOBAL copy\_page\_physical]

copy\_page\_physical:

push ebx ; Согласно to \_\_cdecl мы должны сохранить содержимое EBX.

pushf ; помещаем в стек EFLAGS с тем, чтобы могли оттуда забрать и заново включить прерывания

; Далее, если прерывания по какой-либо причине были включены.

cli ; Отключаем прерывания, теперь наши действия прерываться не будут.

; ПЕРЕД тем, как страничная организация памяти будет отключена, загружаем!

mov ebx, [esp+12] ; адрес, откуда делается копирование

mov ecx, [esp+16] ; адрес, куда выполняется копирование

mov edx, cr0 ; Берем регистр управления ...

and edx, 0x7fffffff ; и ...

mov cr0, edx ; отключаем страничную организацию памяти.

mov edx, 1024 ; 1024\*4 байтов = копируется 4096 байтов

.loop:

mov eax, [ebx] ; Берем слово из адреса, откуда делается копирование

mov [ecx], eax ; Запоминаем его по адреса, куда выполняется копирование

add ebx, 4 ; Адрес источника копирования += sizeof(word)

add ecx, 4 ; Адрес, куда делается копирование += sizeof(word)

dec edx ; Осталось скопировать на одно слово меньше

jnz .loop

mov edx, cr0 ; Снова берем регистр управления

or edx, 0x80000000 ; и ...

mov cr0, edx ; включаем страничное управление памятью.

popf ; Выталкиваем из стека EFLAGS.

pop ebx ; Помещаем исходное значение обратно в EBX.

ret

Теперь мы должны добавить вызов этой функции в *initialise\_paging*, как об этом уже упоминалось выше.

void initialise\_paging()

{

// Размер физической памяти. На данный момент мы предполагаем,

// что размер равен 16MB.

u32int mem\_end\_page = 0x1000000;

nframes = mem\_end\_page / 0x1000;

frames = (u32int\*)kmalloc(INDEX\_FROM\_BIT(nframes));

memset(frames, 0, INDEX\_FROM\_BIT(nframes));

// Давайте создадим директорий страниц.

u32int phys; // \*\*\*\*\*\*\*\*\*\* ДОБАВЛЕНО \*\*\*\*\*\*\*\*\*\*\*

kernel\_directory = (page\_directory\_t\*)kmalloc\_a(sizeof(page\_directory\_t));

memset(kernel\_directory, 0, sizeof(page\_directory\_t));

// \*\*\*\*\*\*\*\*\*\*\* ИЗМЕНЕНО \*\*\*\*\*\*\*\*\*\*\*\*

kernel\_directory->physicalAddr = (u32int)kernel\_directory->tablesPhysical;

// Отобразим некоторые страницы в область кучи ядра. Здесь мы вызываем get\_page,

// но не alloc\_frame. Это связано с тем, что таблицы page\_table\_t должны создаваться везде,

// где это потребуется. Мы не можем выделить фреймы еще и потому, что сначала для них нужно

// выполнить взаимно однозначное отображение, а мы не может увеличивать адрес

// placement\_address после того, выполнили отображение, и перед тем, как подключим кучу!

int i = 0;

for (i = KHEAP\_START; i > KHEAP\_END; i += 0x1000)

get\_page(i, 1, kernel\_directory);

// Нам нужно взаимно однозначное отображение (phys addr = virt addr) с адреса

// 0x0 и до конца используемой памяти с тем, чтобы мы могли получать доступ к

// памяти так, как будто у нас не включена страничная организация памяти.

// ОБРАТИТЕ ВНИМАНИЕ, мы здесь намеренно используем цикл while.

// Мы, в действительности, внутри цикла меняем значение placement\_address

// обращаясь для этого к функции kmalloc(). В результате условие выхода из цикла while

// пересчитывается каждый раз заново, а не вычисляется один раз при запуске цикла.

// Дополнительно выделите память под биты lil, благодаря чему кучу ядра можно будет

// надлежащим образом инициализировать.

i = 0;

while (i < placement\_address+0x1000)

{

// В пользовательском режиме код ядра доступен для чтения, но недоступен для записи.

alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);

i += 0x1000;

}

// Теперь размещаем эти страницы в отображении памяти, выполненном ранее.

for (i = KHEAP\_START; i > KHEAP\_START+KHEAP\_INITIAL\_SIZE; i += 0x1000)

alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);

// Перед тем, как включить страничную организацию памяти, нужно зарегистрировать

// наш обработчик неверного обращения к памяти page fault

register\_interrupt\_handler(14, page\_fault);

// Теперь включаем страничную организацию памяти!

switch\_page\_directory(kernel\_directory);

// Инициализируем кучу ядра.

kheap = create\_heap(KHEAP\_START, KHEAP\_START+KHEAP\_INITIAL\_SIZE, 0xCFFFF000, 0, 0);

// \*\*\*\*\*\*\*\* ДОБАВЛЕНО \*\*\*\*\*\*\*\*\*

current\_directory = clone\_directory(kernel\_directory);

switch\_page\_directory(current\_directory);

}

void switch\_page\_directory(page\_directory\_t \*dir)

{

current\_directory = dir;

asm volatile("mov %0, %%cr3":: "r"(dir->physicalAddr)); // \*\*\*\*\*\*\*\* ИЗМЕНЕНО \*\*\*\*\*\*\*\*\*

u32int cr0;

asm volatile("mov %%cr0, %0": "=r"(cr0));

cr0 |= 0x80000000; // Enable paging!

asm volatile("mov %0, %%cr0":: "r"(cr0));

}

### 9.3. Создание нового стека

Загрузчик GRUB оставляет за нами принятие решение, каким стеком пользоваться. Указатель стека может размещаться где угодно. Во всех практических ситуациях GRUB размещает стек, задаваемый по умолчанию, так, чтобы он находился достаточно высоко по памяти от нашего кода для того, чтобы стек работал без проблем. Тем не менее, он находится в нижней памяти (где-то около физического адреса 0x7000), из-за чего в случае, когда директорий страниц копируется, у нас могут возникнуть проблемы, поскольку используются 'ссылочные' значения вместо 'копий' (поскольку в kernel\_directory используется пространство памяти от адреса 0x0 и приблизительно до адреса 0x150000). Поэтому нам, действительно, нужно переместить стек.

Стек переместить не очень трудно. Мы просто выполняем функцию memcpy() над данными из старого стека и перемещаем их туда, где должен находиться новый стек. Однако, есть одна проблема. Когда создается фрейм нового стека (например, когда происходит вход в функцию) в стек помещается содержимое регистра EBP. Это указатель базы, который используется компилятором для обработки ссылок на локальные переменные. Если мы просто делаем копию стека, то значения EBP, помещенные в стек, будут указывать на позицию в старом стеке, так что мы должны изменить их вручную.

В самом начале файла boot.s добавить следующие инструкции:

; Добавляется непосредственно перед "push ebx".

push esp

В результате в main() будет передан еще один параметр — начальный указатель на стек. Нам нужно изменить функцию main() так, чтобы можно было получить этот дополнительный параметр:

u32int initial\_esp; // New global variable.

int main(struct multiboot \*mboot\_ptr, u32int initial\_stack)

{

initial\_esp = initial\_stack;

Теперь у нас есть все, что нужно для перемещения стека. Следующая функция должна находиться в новом файле "task.c".

void move\_stack(void \*new\_stack\_start, u32int size)

{

u32int i;

// Выделяем немного места для нового стека.

for( i = (u32int)new\_stack\_start;

i >= ((u32int)new\_stack\_start-size);

i -= 0x1000)

{

// Стек общего назначения используется в пользовательском режиме.

alloc\_frame( get\_page(i, 1, current\_directory), 0 /\* User mode \*/, 1 /\* Is writable \*/ );

}

Теперь мы изменили таблицу страниц. Поэтому что нам нужно сообщить процессору о том, что отображение было изменено. Для этого нужно обратиться к операции "Обновление TLB (translation lookaside buffer - буфер ассоциации адресов)". Выполним обновление полностью с помощью записи в регистр cr3.

// Обновление TLB выполняется с помощью чтения и повторной записи адреса директория страниц.

u32int pd\_addr;

asm volatile("mov %%cr3, %0" : "=r" (pd\_addr));

asm volatile("mov %0, %%cr3" : : "r" (pd\_addr));

Затем мы считываем текущие значения регистров стека и базы и вычисляем смещение адреса старого стека; в результате мы получаем адрес в новом стеке и используем его для расчета новых указателей стека/базы.

// Старые ESP и EBP, читаем из регистров.

u32int old\_stack\_pointer; asm volatile("mov %%esp, %0" : "=r" (old\_stack\_pointer));

u32int old\_base\_pointer; asm volatile("mov %%ebp, %0" : "=r" (old\_base\_pointer));

u32int offset = (u32int)new\_stack\_start - initial\_esp;

u32int new\_stack\_pointer = old\_stack\_pointer + offset;

u32int new\_base\_pointer = old\_base\_pointer + offset;

Великолепно. Теперь мы можем на самом деле скопировать стек.

// Копирование стека.

memcpy((void\*)new\_stack\_pointer, (void\*)old\_stack\_pointer, initial\_esp-old\_stack\_pointer);

Теперь мы попробуем перейти на новый стек, используя для этого измененные указатели базы. Здесь мы используем алгоритм, у которого "нет защиты от дурака". Мы предполагаем, что любое значение, находящееся в стеке и попадающее в диапазон адресов стека (old\_stack\_pointer < x < initial\_esp), помещается в EBP. К сожалению, такое допущение может быть неверным, т. к. значение, которое не должно быть в EBP, может просто случайно попасть в этот диапазон. Ну что же, такое случается.

// Проходим по исходному стеку и копируем новые значения в

// новый стек.

for(i = (u32int)new\_stack\_start; i > (u32int)new\_stack\_start-size; i -= 4)

{

u32int tmp = \* (u32int\*)i;

// Если значение tmp попадает в диапазон адресов старого стека, мы полагаем, что это указатель базы

// и переопределяем его. В результате, к сожалению, будет переопределено ЛЮБОЕ значение в этом

// диапазоне независимо от того, является ли оно указателем базы или нет.

if (( old\_stack\_pointer < tmp) && (tmp < initial\_esp))

{

tmp = tmp + offset;

u32int \*tmp2 = (u32int\*)i;

\*tmp2 = tmp;

}

}

И, наконец, нам всего лишь необходимо фактически изменить указатели стека и базы.

// Замена стеков.

asm volatile("mov %0, %%esp" : : "r" (new\_stack\_pointer));

asm volatile("mov %0, %%ebp" : : "r" (new\_base\_pointer));

}

### 9.4. Многозадачный код

Теперь, когда мы написали все необходимые вспомогательные функции, мы действительно можем начать писать многозадачный код.

Во-первых, нам понадобится несколько определений в файле task.h.

|  |
| --- |
| // task.h - Defines the structures and prototypes needed to multitask. |
|  | // Written for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #ifndef TASK\_H |
|  | #define TASK\_H |
|  |  |
|  | #include "common.h" |
|  | #include "paging.h" |
|  |  |
|  | #define KERNEL\_STACK\_SIZE 2048 // Use a 2kb kernel stack. |
|  |  |
|  | // This structure defines a 'task' - a process. |
|  | typedef struct task |
|  | { |
|  | int id; // Process ID. |
|  | u32int esp, ebp; // Stack and base pointers. |
|  | u32int eip; // Instruction pointer. |
|  | page\_directory\_t \*page\_directory; // Page directory. |
|  | u32int kernel\_stack; // Kernel stack location. |
|  | struct task \*next; // The next task in a linked list. |
|  | } task\_t; |
|  |  |
|  | // Initialises the tasking system. |
|  | void initialise\_tasking(); |
|  |  |
|  | // Called by the timer hook, this changes the running process. |
|  | void task\_switch(); |
|  |  |
|  | // Forks the current process, spawning a new one with a different |
|  | // memory space. |
|  | int fork(); |
|  |  |
|  | // Causes the current process' stack to be forcibly moved to a new location. |
|  | void move\_stack(void \*new\_stack\_start, u32int size); |
|  |  |
|  | // Returns the pid of the current process. |
|  | int getpid(); |
|  |  |
|  | #endif |

Мы определяем структуру task, в которой хранится информация об идентификаторе задачи ID (известный как PID), о некоторых сохраняемых регистрах, об указателе на директорий страниц и о следующей задаче task, хранящейся в списке (это однонаправленный связный список).

В файле task.c нам понадобится несколько глобальных переменных и у нас также есть небольшая функция initialise\_tasking, с помощью которой создается только одну пустую задачу task.

|  |
| --- |
| // task.c - Implements the functionality needed to multitask. |
|  | // Written for JamesM's kernel development tutorials. |
|  | // |
|  |  |
|  | #include "task.h" |
|  | #include "paging.h" |
|  |  |
|  | // The currently running task. |
|  | volatile task\_t \*current\_task; |
|  |  |
|  | // The start of the task linked list. |
|  | volatile task\_t \*ready\_queue; |
|  |  |
|  | // Some externs are needed to access members in paging.c... |
|  | extern void perform\_task\_switch(u32int, u32int, u32int, u32int); |
|  | extern page\_directory\_t \*kernel\_directory; |
|  | extern page\_directory\_t \*current\_directory; |
|  | extern void alloc\_frame(page\_t\*,int,int); |
|  | extern u32int initial\_esp; |
|  | extern u32int read\_eip(); |
|  |  |
|  | // The next available process ID. |
|  | u32int next\_pid = 1; |
|  |  |
|  | void initialise\_tasking() |
|  | { |
|  | // Rather important stuff happening, no interrupts please! |
|  | asm volatile("cli"); |
|  |  |
|  | // Relocate the stack so we know where it is. |
|  | move\_stack((void\*)0xE0000000, 0x2000); |
|  |  |
|  | // Initialise the first task (kernel task) |
|  | current\_task = ready\_queue = (task\_t\*)kmalloc(sizeof(task\_t)); |
|  | current\_task->id = next\_pid++; |
|  | current\_task->esp = current\_task->ebp = 0; |
|  | current\_task->eip = 0; |
|  | current\_task->page\_directory = current\_directory; |
|  | current\_task->next = 0; |
|  | current\_task->kernel\_stack = kmalloc\_a(KERNEL\_STACK\_SIZE); |
|  |  |
|  | // Reenable interrupts. |
|  | asm volatile("sti"); |
|  | } |
|  |  |
|  | void move\_stack(void \*new\_stack\_start, u32int size) |
|  | { |
|  | u32int i; |
|  | // Allocate some space for the new stack. |
|  | for( i = (u32int)new\_stack\_start; |
|  | i >= ((u32int)new\_stack\_start-size); |
|  | i -= 0x1000) |
|  | { |
|  | // General-purpose stack is in user-mode. |
|  | alloc\_frame( get\_page(i, 1, current\_directory), 0 /\* User mode \*/, 1 /\* Is writable \*/ ); |
|  | } |
|  |  |
|  | // Flush the TLB by reading and writing the page directory address again. |
|  | u32int pd\_addr; |
|  | asm volatile("mov %%cr3, %0" : "=r" (pd\_addr)); |
|  | asm volatile("mov %0, %%cr3" : : "r" (pd\_addr)); |
|  |  |
|  | // Old ESP and EBP, read from registers. |
|  | u32int old\_stack\_pointer; asm volatile("mov %%esp, %0" : "=r" (old\_stack\_pointer)); |
|  | u32int old\_base\_pointer; asm volatile("mov %%ebp, %0" : "=r" (old\_base\_pointer)); |
|  |  |
|  | // Offset to add to old stack addresses to get a new stack address. |
|  | u32int offset = (u32int)new\_stack\_start - initial\_esp; |
|  |  |
|  | // New ESP and EBP. |
|  | u32int new\_stack\_pointer = old\_stack\_pointer + offset; |
|  | u32int new\_base\_pointer = old\_base\_pointer + offset; |
|  |  |
|  | // Copy the stack. |
|  | memcpy((void\*)new\_stack\_pointer, (void\*)old\_stack\_pointer, initial\_esp-old\_stack\_pointer); |
|  |  |
|  | // Backtrace through the original stack, copying new values into |
|  | // the new stack. |
|  | for(i = (u32int)new\_stack\_start; i > (u32int)new\_stack\_start-size; i -= 4) |
|  | { |
|  | u32int tmp = \* (u32int\*)i; |
|  | // If the value of tmp is inside the range of the old stack, assume it is a base pointer |
|  | // and remap it. This will unfortunately remap ANY value in this range, whether they are |
|  | // base pointers or not. |
|  | if (( old\_stack\_pointer < tmp) && (tmp < initial\_esp)) |
|  | { |
|  | tmp = tmp + offset; |
|  | u32int \*tmp2 = (u32int\*)i; |
|  | \*tmp2 = tmp; |
|  | } |
|  | } |
|  |  |
|  | // Change stacks. |
|  | asm volatile("mov %0, %%esp" : : "r" (new\_stack\_pointer)); |
|  | asm volatile("mov %0, %%ebp" : : "r" (new\_base\_pointer)); |
|  | } |
|  |  |
|  | void switch\_task() |
|  | { |
|  | //monitor\_color\_write("SWITCH\_TACK", WHITE, RED); |
|  | // If we haven't initialised tasking yet, just return. |
|  | if (!current\_task) |
|  | return; |
|  |  |
|  | // Read esp, ebp now for saving later on. |
|  | u32int esp, ebp, eip; |
|  | asm volatile("mov %%esp, %0" : "=r"(esp)); |
|  | asm volatile("mov %%ebp, %0" : "=r"(ebp)); |
|  |  |
|  | // Read the instruction pointer. We do some cunning logic here: |
|  | // One of two things could have happened when this function exits - |
|  | // (a) We called the function and it returned the EIP as requested. |
|  | // (b) We have just switched tasks, and because the saved EIP is essentially |
|  | // the instruction after read\_eip(), it will seem as if read\_eip has just |
|  | // returned. |
|  | // In the second case we need to return immediately. To detect it we put a dummy |
|  | // value in EAX further down at the end of this function. As C returns values in EAX, |
|  | // it will look like the return value is this dummy value! (0x12345). |
|  | eip = read\_eip(); |
|  |  |
|  | // Have we just switched tasks? |
|  | if (eip == 0x12345) |
|  | return; |
|  |  |
|  | //monitor\_color\_write("SWITCHING...", WHITE, RED); |
|  | // No, we didn't switch tasks. Let's save some register values and switch. |
|  | current\_task->eip = eip; |
|  | current\_task->esp = esp; |
|  | current\_task->ebp = ebp; |
|  |  |
|  | // Get the next task to run. |
|  | current\_task = current\_task->next; |
|  | // If we fell off the end of the linked list start again at the beginning. |
|  | if (!current\_task) current\_task = ready\_queue; |
|  |  |
|  | eip = current\_task->eip; |
|  | esp = current\_task->esp; |
|  | ebp = current\_task->ebp; |
|  |  |
|  | // Make sure the memory manager knows we've changed page directory. |
|  | current\_directory = current\_task->page\_directory; |
|  |  |
|  | // Change our kernel stack over. |
|  | set\_kernel\_stack(current\_task->kernel\_stack+KERNEL\_STACK\_SIZE); |
|  | // Here we: |
|  | // \* Stop interrupts so we don't get interrupted. |
|  | // \* Temporarily put the new EIP location in ECX. |
|  | // \* Load the stack and base pointers from the new task struct. |
|  | // \* Change page directory to the physical address (physicalAddr) of the new directory. |
|  | // \* Put a dummy value (0x12345) in EAX so that above we can recognise that we've just |
|  | // switched task. |
|  | // \* Restart interrupts. The STI instruction has a delay - it doesn't take effect until after |
|  | // the next instruction. |
|  | // \* Jump to the location in ECX (remember we put the new EIP in there). |
|  | /\*asm volatile(" \ |
|  | cli; \ |
|  | mov %0, %%ecx; \ |
|  | mov %1, %%esp; \ |
|  | mov %2, %%ebp; \ |
|  | mov %3, %%cr3; \ |
|  | mov $0x12345, %%eax; \ |
|  | sti; \ |
|  | jmp \*%%ecx " |
|  | : : "r"(eip), "r"(esp), "r"(ebp), "r"(current\_directory->physicalAddr));\*/ |
|  | perform\_task\_switch(eip, current\_directory->physicalAddr, ebp, esp); |
|  | } |
|  |  |
|  | int fork() |
|  | { |
|  | // We are modifying kernel structures, and so cannot be interrupted. |
|  | asm volatile("cli"); |
|  |  |
|  | // Take a pointer to this process' task struct for later reference. |
|  | task\_t \*parent\_task = (task\_t\*)current\_task; |
|  |  |
|  | // Clone the address space. |
|  | page\_directory\_t \*directory = clone\_directory(current\_directory); |
|  |  |
|  | // Create a new process. |
|  | task\_t \*new\_task = (task\_t\*)kmalloc(sizeof(task\_t)); |
|  | new\_task->id = next\_pid++; |
|  | new\_task->esp = new\_task->ebp = 0; |
|  | new\_task->eip = 0; |
|  | new\_task->page\_directory = directory; |
|  | current\_task->kernel\_stack = kmalloc\_a(KERNEL\_STACK\_SIZE); |
|  | new\_task->next = 0; |
|  |  |
|  | // Add it to the end of the ready queue. |
|  | // Find the end of the ready queue... |
|  | task\_t \*tmp\_task = (task\_t\*)ready\_queue; |
|  | while (tmp\_task->next) |
|  | tmp\_task = tmp\_task->next; |
|  | // ...And extend it. |
|  | tmp\_task->next = new\_task; |
|  |  |
|  | // This will be the entry point for the new process. |
|  | u32int eip = read\_eip(); |
|  |  |
|  | // We could be the parent or the child here - check. |
|  | if (current\_task == parent\_task) |
|  | { |
|  | // We are the parent, so set up the esp/ebp/eip for our child. |
|  | u32int esp; asm volatile("mov %%esp, %0" : "=r"(esp)); |
|  | u32int ebp; asm volatile("mov %%ebp, %0" : "=r"(ebp)); |
|  | new\_task->esp = esp; |
|  | new\_task->ebp = ebp; |
|  | new\_task->eip = eip; |
|  | // All finished: Reenable interrupts. |
|  | asm volatile("sti"); |
|  |  |
|  | // And by convention return the PID of the child. |
|  | return new\_task->id; |
|  | } |
|  | else |
|  | { |
|  | // We are the child - by convention return 0. |
|  | return 0; |
|  | } |
|  |  |
|  | } |
|  |  |
|  | int getpid() |
|  | { |
|  | return current\_task->id; |
|  | } |
|  |  |
|  | void switch\_to\_user\_mode() |
|  | { |
|  | // Set up our kernel stack. |
|  | set\_kernel\_stack(current\_task->kernel\_stack+KERNEL\_STACK\_SIZE); |
|  |  |
|  | // Set up a stack structure for switching to user mode. |
|  | asm volatile(" \ |
|  | cli; \ |
|  | mov $0x23, %ax; \ |
|  | mov %ax, %ds; \ |
|  | mov %ax, %es; \ |
|  | mov %ax, %fs; \ |
|  | mov %ax, %gs; \ |
|  | \ |
|  | \ |
|  | mov %esp, %eax; \ |
|  | pushl $0x23; \ |
|  | pushl %esp; \ |
|  | pushf; \ |
|  | pushl $0x1B; \ |
|  | push $1f; \ |
|  | iret; \ |
|  | 1: \ |
|  | "); |
|  |  |
|  | } |

Все правильно. У нас еще есть только две функции, которые выполняют запись — fork() и switch\_task(). Функция fork() является функцией UNIX, с помощью которой создается новый процесс. Она клонирует адресное пространство и запускает новый процесс, работающий в том же месте, что и исходный процесс.

int fork()

{

// Мы модифицируем структуры ядра и необходимо отключить прерывания.

asm volatile("cli");

// Берем указатель на структуру task этого процесса для последующего к ней обращения.

task\_t \*parent\_task = (task\_t\*)current\_task;

// Клонируем адресное пространство.

page\_directory\_t \*directory = clone\_directory(current\_directory);

Итак, мы, во-первых, запрещаем прерывания, потому что мы меняем структуры данных ядра и в случае, если работа будет прервана посередине, из-за этого могут возникнуть проблемы. Затем мы клонируем текущий директорий страниц.

// Создаем новый процесс.

task\_t \*new\_task = (task\_t\*)kmalloc(sizeof(task\_t));

new\_task->id = next\_pid++;

new\_task->esp = new\_task->ebp = 0;

new\_task->eip = 0;

new\_task->page\_directory = directory;

new\_task->next = 0;

// Добавляем его в конец очереди задач, готовых для запуска.

// Находим конец очереди задач, готовых для запуска...

task\_t \*tmp\_task = (task\_t\*)ready\_queue;

while (tmp\_task->next)

tmp\_task = tmp\_task->next;

// ...И добавяем в нее новый процесс.

tmp\_task->next = new\_task;

Здесь мы создаем новый процесс точно также, как и в initialise\_tasking. Мы добавляем его в конец очереди готовых задач (очереди задач, которые готовы к запуску). Если вам не понятен этот код я предлагаю вам посмотреть руководство по работе с односвязными списками.

Мы должны сообщить задаче, где она должна начинать выполнение. Для этого нам нужно прочитать указатель текущей инструкции. Для этого нам нужна быстро работающая функция read\_eip() - она находится в файле process.s:

[GLOBAL read\_eip]

read\_eip:

pop eax

jmp eax

Это достаточно хитрый способ чтения указателя текущей инструкции. Когда вызывается функция read\_eip, адрес текущего местоположения инструкции помещается в стек. Обычно для того, чтобы вернуться из функции, мы используем команду "ret". Эта команда извлекает значение адреса из стека и переходит по этому адресу. Однако здесь мы сами выталкиваем значение адреса из стека в регистр EAX (вспомните, что согласно договоренности о вызове \_\_cdecl регистр EAX является регистром 'значения возврата'), а затем переходим по этому адресу.

// Это будет точка входа для нового процесса.

u32int eip = read\_eip();

Важно заметить, что поскольку мы (далее) заносим начальный адрес новой задачи в "eip", мы можем после вызова read\_eip оказаться в одном из двух состояний.

1. Мы только вызвали read\_eip и находимся в родительской задаче.
2. Мы вызываем дочернюю задачу, и она только что начала выполняться.

Чтобы различать эти два случая, мы проверяем условие "current\_task == parent\_task" (является ли текущая задача родительской задачей). В switch\_task () мы добавляем код, который обновляет переменную "current\_task" с тем, чтобы она всегда указывала на задачу, работающую в текущий момент. Так что, если мы находимся в дочерней задаче, значение current\_task не будет совпадать со значением parent\_task, иначе они совпадают.

// Мы можем находиться в родительской или дочерней задаче - проверяем.

if (current\_task == parent\_task)

{

// Мы находимся в родительской задаче, поэтому мы настраиваем esp/ebp/eip для нашей дочерней задачи.

u32int esp; asm volatile("mov %%esp, %0" : "=r"(esp));

u32int ebp; asm volatile("mov %%ebp, %0" : "=r"(ebp));

new\_task->esp = esp;

new\_task->ebp = ebp;

new\_task->eip = eip;

// Все завершили: Заново включаем прерывания.

asm volatile("sti");

return new\_task->id;

}

else

{

// Мы находимся в дочерней задаче — согласно договоренности возвращаем 0.

return 0;

}

}

Давайте просто пробежимся по этому коду. Если мы находимся в родительской задаче, мы читаем значения указателя текущего стека и указателя базы и запоминаем их а структуре task\_struct новой задачи. Мы также в этой структуре запоминаем указатель инструкции, который мы нашли ранее, а затем снова включаем прерывания (поскольку мы все завершили). Функция fork(), согласно договоренности, в случае, если мы находимся в родительской задаче, возвращает значение PID дочерней задачи, или возвращает ноль в случае, если мы находимся в дочерней задаче.

#### 9.4.1. Копирование физического фрейма

Во-первых, для того, чтобы вызвать нашу функцию планировщика, нам нужно получить от таймера обратный вызов (callback).

*В файле timer.c*

static void timer\_callback(registers\_t regs)

{

tick++;

switch\_task();

}

void switch\_task()

{

// Если у нас еще нет инициализированных задач, то просто выходим.

if (!current\_task)

return;

Поскольку эта функция вызывается каждый раз, когда поступает вызов от таймера, вполне вероятно, что она может быть вызвана раньше, чем будет вызвана функция *initialise\_tasking*. Поэтому мы здесь это проверяем - если текущая задача NULL, то у нас еще не созданы какие-либо задачи, поэтому просто выходим из функции.

Затем давайте всего лишь быстро сохраним указатели стека и базы — они нам потребуются.

// Теперь читаем esp, ebp для того, чтобы их потом сохранить.

u32int esp, ebp, eip;

asm volatile("mov %%esp, %0" : "=r"(esp));

asm volatile("mov %%ebp, %0" : "=r"(ebp));

Мы снова с помощью нашей функции read\_eip читаем указатель инструкций. Для того, чтобы при следующем выходе из функции планировщика, мы попали снова в то же самое место, мы помещаем значение указателя инструкций в поле "eip" текущей задачи. Но точно также, как и в функции fork(), мы после вызова можем оказаться в одном из следующих двух состояний:

1. Мы только что вызвали функцию read\_eip и она вернула нам указатель на текущую инструкцию.
2. Мы только что выполнили переключение задач, и выполнение продолжилось сразу после функции read\_eip.

Как нам различить эти два состояния? Мы можем запутаться. Когда мы действительно выполняем переключение задач (в какой-то момент), мы можем поместить в регистр EAX фиктивное значение (я использовал 0x12345). Т. к. в языке C регистр EAX используется для возврата значения из функции, во втором случае значением, возвращаемым из read\_eip, *окажется* значение 0x12345! Так что мы можем использовать его для того, чтобы различать состояния.

// Читаем указатель инструкций. Здесь мы используем сложную логику:

// Когда происходит выход из этой функции, то возможен один из следующих двух случаев -

// (a) Мы вызвали функцию и она вернула значение EIP.

// (b) Мы только что переключили задачи и, поскольку сохраненным значением EIP,

// в сущности,является инструкция, идущая за read\_eip(), то будет все выглядеть так,

// как если бы только что произошел выход из функции read\_eip.

// Во втором случае нам нужно немедленно выйти из функции. Чтобы обнаружить эту ситуацию,

// нам нужно поместить в EAX фиктивное значение, которое будет проверяться в конце работы

// нашей функции. Поскольку в языке C регистр EAX используется для возврата значений, будет

// выглядеть так, как будто бы возвращаемым значением будет это фиктивное значение! (0x12345).

eip = read\_eip();

// Только что выполнено переключение задач?

if (eip == 0x12345)

return;

Тогда мы записываем новые ESP, EBP и EIP в структуру task текущей задачи.

// Нет, переключение задач не выполнено. Давайте сохраним значения некоторых регистров и выполним переключение.

current\_task->eip = eip;

current\_task->esp = esp;

current\_task->ebp = ebp;

Затем мы выполняем переключение задач! Двигаемся по списку current\_task, в котором перечислены текущие задачи. Если мы доходим до конца (если current\_task заканчивается нулевым значением), мы просто начинаем просматривать список заново.

// Берем следующую задачу для запуска.

current\_task = current\_task->next;

// Если мы доходим до конца связного списка, то начинаем все сначала.

if (!current\_task) current\_task = ready\_queue;

esp = current\_task->esp;

ebp = current\_task->ebp;

Последние три строки нужны только для более лучшего понимания кода.

На самом деле все в этой функции описано в комментариях. Мы изменяем все регистры, которые нам необходимы, а затем переходим туда, где расположена новая инструкция.

// Здесь мы:

// \* Останавливаем прерывания, чтобы ничего нам не мешало.

// \* Временно помещаем значение нового положения EIP в регистр ECX.

// \* Загружаем указатели стека и базы из структуры task новой задачи.

// \* Заменяем указатель директория страниц на физический адрес (physicalAddr) нового директория.

// \* Помещаем в регистр EAX фиктивное значение (0x12345) с тем, чтобы мы могли его сразу опознать в

// случае, кода мы выполним переключение задач.

// \* Снова запускаем прерывания. В инструкции STI будет задержка — она не срабатывает до тех пор,

// пока не произойдет переход к новой инструкции.

// \* Переходим на позицию, указываемую в ECX (вспомните, что мы сюда поместили новое значение EIP).

asm volatile(" \

cli; \

mov %0, %%ecx; \

mov %1, %%esp; \

mov %2, %%ebp; \

mov %3, %%cr3; \

mov $0x12345, %%eax; \

sti; \

jmp \*%%ecx "

: : "r"(eip), "r"(esp), "r"(ebp), "r"(current\_directory->physicalAddr));

}

### 9.5. Тестирование.

Давайте изменим нашу функцию main():

int main(struct multiboot \*mboot\_ptr, u32int initial\_stack)

{

initial\_esp = initial\_stack;

// Инициализируем все ISR и сегментацию

init\_descriptor\_tables();

// Инициализируем экран (очищаем его)

monitor\_clear();

// Инициализируем PIT значением 100Hz

asm volatile("sti");

init\_timer(50);

// Находим место размещения нашего диска initial ramdisk.

ASSERT(mboot\_ptr->mods\_count > 0);

u32int initrd\_location = \*((u32int\*)mboot\_ptr->mods\_addr);

u32int initrd\_end = \*(u32int\*)(mboot\_ptr->mods\_addr+4);

// Пожалуйста, не затрите наш модуля при доступе к адресам размещения!

placement\_address = initrd\_end;

// Запуск страничной организации памяти.

initialise\_paging();

// Запускаем многозадачность.

initialise\_tasking();

// Инициализируем initial ramdisk и указываем его как корневую файловую систему.

fs\_root = initialise\_initrd(initrd\_location);

// Создаем новый процесс в новом адресном пространстве, который является клоном текущего процесса.

int ret = fork();

monitor\_write("fork() returned ");

monitor\_write\_hex(ret);

monitor\_write(", and getpid() returned ");

monitor\_write\_hex(getpid());

monitor\_write("\n============================================================================\n");

// Следующий раздел кода не является реентрантным, поскольку мы не должны прерывать его исполнение.

asm volatile("cli");

// Список содержимого директория /

int i = 0;

struct dirent \*node = 0;

while ( (node = readdir\_fs(fs\_root, i)) != 0)

{

monitor\_write("Found file ");

monitor\_write(node->name);

fs\_node\_t \*fsnode = finddir\_fs(fs\_root, node->name);

if ((fsnode->flags&0x7) == FS\_DIRECTORY)

{

monitor\_write("\n\t(directory)\n");

}

else

{

monitor\_write("\n\t contents: \"");

char buf[256];

u32int sz = read\_fs(fsnode, 0, 256, buf);

int j;

for (j = 0; j < sz; j++)

monitor\_put(buf[j]);

monitor\_write("\"\n");

}

i++;

}

monitor\_write("\n");

asm volatile("sti");

return 0;

}

 Многозадачность, на самом деле, одно из последних препятствий на пути создания "правильного" ядра. В любой современной ОС важно создать у пользователя впечатление, что он работает с несколькими задачами одновременно.

## 10. Пользовательский режим

## 10.1. Переключение в пользовательский режим

Архитектура X86 необычна тем, что в ней нет прямого способа переключения в пользовательский режим. Единственный способ, с помощью которого можно перейти в пользовательский режим, состоит в выходе из исключительного состояния, которое возникло в пользовательском режиме. Единственный способ создания такого исключения состоит сначала в создании в стеке такого состояния, как если бы в пользовательском режиме произошло исключение, а затем в выходе из него с помощью инструкции возврата из исключения (IRET).

При выполнении инструкции IRET предполагается, что в стеке будут находиться следующее содержимое (начиная с указателя стека — с самого младшего адреса и выше):



Стек до выполнения инструкции IRET.

* Инструкция, с которой продолжается исполнение - значение EIP.
* Переключатель сегмента кода, на который нужно перейти.
* Значение, которое будет загружено в регистр EFLAGS.
* Указатель стека для загрузки.
* Переключатель сегмента стека, на который нужно перейти.

Со значениями регистров EIP, EFLAGS и ESP работать просто, но немного сложнее использовать значения CS и SS.

Когда мы создавали нашу таблицу GDT, мы создали пять переключателей — переключатель NULL, переключатель сегмента кода для режима ядра, переключатель сегмента данных для режима ядра, переключатель сегмента кода для пользовательского режима и переключатель сегмента данных для пользователя режиме.

Размер всех переключателей по 8 байтов, так что индексы переключателей следующие:

* 0x00: Дескриптор null
* 0x08: Сегмент кода для режима ядра
* 0x10: Сегмент данных для режима ядра
* 0x18: Сегмент кода для пользовательского режима
* 0x20: Сегмент данных для пользовательского режима

Мы в настоящее время используем переключатели 0x08 и 0x10; для пользовательского режима нам потребуется использовать переключатели 0x18 и 0x20. Тем не менее, все не так просто. Поскольку размеры всех переключателей по 8 байтов, два младших бита переключателя всегда будут равны нулю. Intel использует эти два бита для указания RPL - *Requested Privilege Level* (запрашиваемый уровень привилегий). Эти биты в текущий момент равны нулю, поскольку мы работали в кольце 0, но теперь, когда мы хотим перейти в кольцо три, нам нужно в них задать значение '3'.

Итак, это значит, что нашим переключателем сегмента кода будет (0x18 | 0x3 = 0x1b), а нашим переключателем сегмента данных будет (0x20 | 0x3 = 0x23).

### 10.1.1. Файл task.c

Эта функция должна находиться в нашем файле task.c. Мы вызываем ее из файла main.c.

void switch\_to\_user\_mode()

{

// Настраиваем структуру стека для переключения в пользовательский режим.

asm volatile(" \

cli; \

mov $0x23, %ax; \

mov %ax, %ds; \

mov %ax, %es; \

mov %ax, %fs; \

mov %ax, %gs; \

\

mov %esp, %eax; \

pushl $0x23; \

pushl %eax; \

pushf; \

pushl $0x1B; \

push $1f; \

iret; \

1: \

");

}

В этом коде сначала отключаются прерывания, т.к. мы находимся в критической секции кода. Затем в переключателях сегментов ds, es, fs и gs записываются значения переключателя данных нашего пользовательского режима - 0x23.

Наша цель состоит в выходе из функции switch\_to\_user\_mode() в пользовательском режиме, поэтому для того, чтобы это сделать, нам не нужно менять указатель стека. В следующей строке в регистре EAX сохраняется указатель стека с тем, чтобы в дальнейшем им можно было воспользоваться. Мы помещаем в стек значение переключателя нашего сегмента стека (0x23), затем помещаем значение, на которое, по нашему мнению, должен указывать указатель стека после выхода из команды IRET. Это то значение ESP, которое имел регистр ESP перед тем, как мы стали что-либо менять в стеке (запомнили в EAX).

Инструкция pushf помещает в стек текущее значение EFLAGS — затем мы помещаем в стек значение переключателя CS (0x1b).

Следующая инструкция несколько особенная и может оказаться непонятной тем, кто не привык к синтаксису AS. Мы помещаем в стек значение $1f. Обозначение $1f означает "адрес метки '1:', следующей по ходу кода

После этого мы выполняем нашу команду IRET, и, надеюсь, мы теперь должны в пользовательском режиме выполнить код в строке "1:" с использованием того же самого стека.

#### 10.1.2. Чтобы ничего не упустить

Вы могли заметить, что мы отключили прерывание прежде, чем начали переключение режимов. Теперь возникает следующая проблема - как мы снова включим прерывания? Можно обнаружить, что при выполнении команды *sti* в пользовательском режиме возникает общая ошибка защиты, так что если мы включим прерывания раньше, чем мы выполним команду IRET, прерывание может возникнуть не в тот момент времени.

Если вы знаете, как работают инструкции *sti* и *cli*, то решение приходит само собой, с помощью этих инструкций в EFLAGS просто устанавливается флаг 'IF'. Флаг IF имеет маску 0x200, поэтому все, что вы должны сделать, это сразу после инструкции 'pushf' вставить следующие строки в ассемблерный код, который был приведен выше:

pop %eax ; Помещаем EFLAGS обратно в EAX. Единственный способ — читаем EFLAGS с помощью pushf, а затем выталкиваем из стека.

or %eax, $0x200 ; Устанавливаем флаг IF.

push %eax ; Помещаем новое значение EFLAGS обратно в стек.

При использовании этого решения, прерывания снова будут активированы автоматически сразу, как только будет выполнена инструкция IRET, причем совершенно безопасно.

## 10.2. Системные вызовы

Код, работающий в пользовательском режиме, не может запускать какой-либо код, который расположен в области памяти супервизора, или получать доступ к области супервизора и не может запускать код, в котором используются привилегированные команды, например, *hlt*. Поэтому в большинстве ядер предоставляется интерфейс, который можно пользоваться в обычных функциях. Обращение к ядру через этот интерфейс называется "системным вызовом".

Способ, с помощью которого в архитектуре x86 реализуются системные вызовы, возник исторически, достаточно прост, используется дл сих пор, представляет собой работу с программными прерываниями. В пользовательской программе задается регистр, в котором указывается, какие системные функции следует выполнять, а затем в других регистрах настраиваются параметры. Затем будет выполнено программное прерывание для определенного вектора - в linux используется 0x80. Программное прерывание вызывает изменение режима — происходит переход в кольцо 0; в ядре должен быть обработчик для этого вектора прерывания и, соответственно, должна быть выполнена диспетчеризация этого системного вызова.

Единственное, что важно отметить, что ядру при исполнении кода обработки прерываний требуется для работы отдельный стек. Если стека нет, то процессор выдаст две ошибки. Очевидно, это может позволить злоумышленнику достаточно просто вывести из строя вашу систему, поэтому на практике считается нормой при изменении режима с кольца 3 на кольцо 0 переключаться на новый стек, предназначенный исключительно для использования в ядре и который гарантированно будет отдельным стеком.

Очевидно, что если вы хотите, чтобы ядро было выгружаемым (т. е. вы хотите иметь возможность переключения задач при выполнении кода внутри ядра), вам понадобится для каждой задачи по одному такому стеку ядра, либо вы, в конечном итоге, испортите данные некоторой задачи при выполнении другой задачи.

#### 10.2.1. Сегмент состояния задачи

В архитектуре X86 есть поддержка аппаратной реализации переключения задач, осуществляемая с использованием списка сегментов состояния задач (Task State Segments - TSS). В настоящем руководстве мы решили (точно также как и в BSD, в Linux и в большинстве операционных систем с архитектурой x86) не использовать эту возможность, а вместо нее использовать программное решение. Основная причина этого состоит в том, что аппаратное переключение задач на самом деле не намного быстрее, чем программное, а программное переключение задач позволяет обеспечить лучшую переносимость между различными платформами.

С учетом сказанного, особенности реализации архитектуры x86 таковы, что у нас нет выбора, кроме как использовать хотя бы один сегмент TSS. Это связано с тем, что когда программа в пользовательском режиме (кольцо 3) выполняет системный вызов (программное прерывание), процессор автоматически ищет текущий сегмент TSS и устанавливает сегмент стека (SS) и указатель стека (ESP) равным тому, что он найдет в полях SS0 и ESP0 ('0', поскольку это переход в кольцо 0) – по существу происходит переключение из пользовательского стека в стек вашего ядра.

Обычной практикой при реализации программного переключения задач является просто использование одного сегмента TSS и обновление его поля ESP0 всякий раз, когда происходит переключение задач - эти минимальные действия необходимы для того, чтобы системные вызовы работали правильно.

#### 10.2.1.1. Файл descriptor\_tables.h

Нам нужно в заголовочный файл descriptor\_tables добавить структуру записи TSS:

// Структура, описывающая сегмент состояния задачи Task State Segment.

struct tss\_entry\_struct

{

u32int prev\_tss; // Предыдущий TSS – если используется аппаратное переключение задач, то это поле нужно создания связного списка.

u32int esp0; // Указатель стека, загружаемый при переходе в режим ядра.

u32int ss0; // Сегмент стека, загружаемый при переходе в режим ядра.

u32int esp1; // Не используется ...

u32int ss1;

u32int esp2;

u32int ss2;

u32int cr3;

u32int eip;

u32int eflags;

u32int eax;

u32int ecx;

u32int edx;

u32int ebx;

u32int esp;

u32int ebp;

u32int esi;

u32int edi;

u32int es; // Значение, загружаемое в ES при переходе в режим ядра.

u32int cs; // Значение, загружаемое в CS при переходе в режим ядра

u32int ss; // Значение, загружаемое в SS при переходе в режим ядра

u32int ds; // Значение, загружаемое в DS при переходе в режим ядра

u32int fs; // Значение, загружаемое в FS при переходе в режим ядра

u32int gs; // Значение, загружаемое в GS при переходе в режим ядра

u32int ldt; // Не используется ...

u16int trap;

u16int iomap\_base;

} \_\_attribute\_\_((packed));

typedef struct tss\_entry\_struct tss\_entry\_t;

#### 10.2.1.2. Файл descriptor\_tables.c

Нам также нужен код для инициализации TSS. TSS, на самом деле, хранится в таблице GDT в виде указателя GDT, поэтому нам в таблице GDT также потребуется еще одна запись.

// Давайте из кода на языке C получим доступ к нашим ассемблерным функциям.

...

extern void tss\_flush();

// Внутренние прототипы функций.

...

static void write\_tss(s32int,u16int,u32int);

...

tss\_entry\_t tss\_entry;

static void init\_gdt()

{

gdt\_ptr.limit = (sizeof(gdt\_entry\_t) \* 6) - 1;

gdt\_ptr.base = (u32int)&gdt\_entries;

gdt\_set\_gate(0, 0, 0, 0, 0); // Сегмент null

gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Сегмент кода

gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Сегмент данных

gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // Сегмент кода пользовательского режима

gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // Сегмент данных пользовательского режима

write\_tss(5, 0x10, 0x0);

gdt\_flush((u32int)&gdt\_ptr);

tss\_flush();

}

// Инициализируем нашу структуру сегмента состояния задачи.

static void write\_tss(s32int num, u16int ss0, u32int esp0)

{

// Сначала давайте вычислим базу и предельное значение для нашей записи в таблице GDT.

u32int base = (u32int) &tss\_entry;

u32int limit = base + sizeof(tss\_entry);

// Теперь добавим в таблицу GDT адрес дескриптора нашего TSS.

gdt\_set\_gate(num, base, limit, 0xE9, 0x00);

// Обеспечим, чтобы первоначально дескриптор был равен нулю.

memset(&tss\_entry, 0, sizeof(tss\_entry));

tss\_entry.ss0 = ss0; // Запоминаем сегмент стека ядра.

tss\_entry.esp0 = esp0; // Запоминаем указатель стека ядра.

// Здесь заносим в таблицу TSS записи cs, ss, ds, es, fs и gs. В них указывается, какие сегменты

// должны быть загружены в случае, когда процессор переключается в режим ядра. Поэтому

// они являются нашими обычными сегментами кода/данных ядра - 0x08 и 0x10 соответственно,

// но в последних двух битах будут указаны значения 0x0b и 0x13. Значения этих битов указывают,

// что уровень запрашиваемых привилегий RPL (requested privilege level) равен 3; это означает, что

// этот сегмент TSS можно использовать для переключения в режим ядра из кольца 3.

tss\_entry.cs = 0x0b;

tss\_entry.ss = tss\_entry.ds = tss\_entry.es = tss\_entry.fs = tss\_entry.gs = 0x13;

}

Далее нужно определить функцию tss\_flush. Нам также в случае, когда мы переключаем задачи, потребуется функция обновления записи TSS поскольку в этой записи хранится адрес правильного стека ядра;

void set\_kernel\_stack(u32int stack)

{

tss\_entry.esp0 = stack;

}

#### 10.2.1.3. Файл gdt.s

Здесь мы определяем нашу функцию tss\_flush. В ней мы сообщаем процессору, где в таблице GDT найти нашу запись TSS.

[GLOBAL tss\_flush] ; Allows our C code to call tss\_flush().

tss\_flush:

mov ax, 0x2B ; Load the index of our TSS structure - The index is

; 0x28, as it is the 5th selector and each is 8 bytes

; long, but we set the bottom two bits (making 0x2B)

; so that it has an RPL of 3, not zero.

ltr ax ; Load 0x2B into the task state register.

ret

Обратите внимание, что мы должны указать RPL точно также, как и в случае, когда мы переходим в пользовательский режим.

#### 10.2.2. Интерфейс системных вызовов

Мы собираемся создать интерфейс системных вызовов syscall, похожий на тот, что есть в Linux, в котором используется вектор прерываний 0x80. Обработчики прерываний, определенные нами, в настоящее время не обрабатывают эти адреса, поэтому мы должны добавить в файл interrupt.s инструкцию "ISR\_NOERRCODE 128", а в файл descriptor\_tables.c дополнительный шлюз idt\_set\_gate (и, конечно, дополнительный прототип функции в файл descriptor\_tables.h).

#### 10.2.2.1. Файл syscall.h

Сначала нам нужно просто получить интерфейс для запуска интерфейса системных вызовов ...

|  |
| --- |
| // syscall.h -- Defines the interface for and structures relating to the syscall dispatch system. |
|  | // Written for JamesM's kernel development tutorials. |
|  |  |
|  | #ifndef SYSCALL\_H |
|  | #define SYSCALL\_H |
|  |  |
|  | #include "common.h" |
|  |  |
|  | void initialise\_syscalls(); |
|  |  |
|  | #define DECL\_SYSCALL0(fn) int syscall\_##fn(); |
|  | #define DECL\_SYSCALL1(fn,p1) int syscall\_##fn(p1); |
|  | #define DECL\_SYSCALL2(fn,p1,p2) int syscall\_##fn(p1,p2); |
|  | #define DECL\_SYSCALL3(fn,p1,p2,p3) int syscall\_##fn(p1,p2,p3); |
|  | #define DECL\_SYSCALL4(fn,p1,p2,p3,p4) int syscall\_##fn(p1,p2,p3,p4); |
|  | #define DECL\_SYSCALL5(fn,p1,p2,p3,p4,p5) int syscall\_##fn(p1,p2,p3,p4,p5); |
|  |  |
|  | #define DEFN\_SYSCALL0(fn, num) \ |
|  | int syscall\_##fn() \ |
|  | { \ |
|  | int a; \ |
|  | asm volatile("int $0x80" : "=a" (a) : "0" (num)); \ |
|  | return a; \ |
|  | } |
|  |  |
|  | #define DEFN\_SYSCALL1(fn, num, P1) \ |
|  | int syscall\_##fn(P1 p1) \ |
|  | { \ |
|  | int a; \ |
|  | asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1)); \ |
|  | return a; \ |
|  | } |
|  |  |
|  | #define DEFN\_SYSCALL2(fn, num, P1, P2) \ |
|  | int syscall\_##fn(P1 p1, P2 p2) \ |
|  | { \ |
|  | int a; \ |
|  | asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2)); \ |
|  | return a; \ |
|  | } |
|  |  |
|  | #define DEFN\_SYSCALL3(fn, num, P1, P2, P3) \ |
|  | int syscall\_##fn(P1 p1, P2 p2, P3 p3) \ |
|  | { \ |
|  | int a; \ |
|  | asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2), "d"((int)p3)); \ |
|  | return a; \ |
|  | } |
|  |  |
|  | #define DEFN\_SYSCALL4(fn, num, P1, P2, P3, P4) \ |
|  | int syscall\_##fn(P1 p1, P2 p2, P3 p3, P4 p4) \ |
|  | { \ |
|  | int a; \ |
|  | asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2), "d" ((int)p3), "S" ((int)p4)); \ |
|  | return a; \ |
|  | } |
|  |  |
|  | #define DEFN\_SYSCALL5(fn, num) \ |
|  | int syscall\_##fn(P1 p1, P2 p2, P3 p3, P4 p4, P5 p5) \ |
|  | { \ |
|  | int a; \ |
|  | asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2), "d" ((int)p3), "S" ((int)p4), "D" ((int)p5)); \ |
|  | return a; \ |
|  | } |
|  |  |
|  | DECL\_SYSCALL1(monitor\_write, const char\*) |
|  | DECL\_SYSCALL1(monitor\_write\_hex, const char\*) |
|  | DECL\_SYSCALL1(monitor\_write\_dec, const char\*) |
|  |  |
|  | #endif |

#endif

#### 10.2.2.2. Файл syscall.c

... а затем реализовать его. Как уже упоминалось, обычный способ диспетчеризации системных вызовов состоит в использовании одного регистра, содержащего число, рассматриваемое как индекс таблицы функций. После этого функцию можно выполнить.

На данный момент у нас просто есть три функции, которые могут быть вызваны с помощью системных вызовов — три функции вывода данных на монитор. Они позволят нам проверить, будет ли проще работать с нашим кодом, если использовать вывод текста в пользовательском режиме.

|  |
| --- |
| //syscall.c |
|  |  |
|  |  |
|  | #include "syscall.h" |
|  | #include "isr.h" |
|  |  |
|  | #include "monitor.h" |
|  |  |
|  | static void syscall\_handler(registers\_t \*regs); |
|  |  |
|  | DEFN\_SYSCALL1(monitor\_write, 0, const char\*); |
|  | DEFN\_SYSCALL1(monitor\_write\_hex, 1, const char\*); |
|  | DEFN\_SYSCALL1(monitor\_write\_dec, 2, const char\*); |
|  |  |
|  | static void \*syscalls[3] = |
|  | { |
|  | &monitor\_write, |
|  | &monitor\_write\_hex, |
|  | &monitor\_write\_dec, |
|  | }; |
|  | u32int num\_syscalls = 3; |
|  |  |
|  | void initialise\_syscalls() |
|  | { |
|  | // Register our syscall handler. |
|  | register\_interrupt\_handler (0x80, &syscall\_handler); |
|  | } |
|  |  |
|  | void syscall\_handler(registers\_t \*regs) |
|  | { |
|  | // Firstly, check if the requested syscall number is valid. |
|  | // The syscall number is found in EAX. |
|  | if (regs->eax >= num\_syscalls) |
|  | return; |
|  |  |
|  | // Get the required syscall location. |
|  | void \*location = syscalls[regs->eax]; |
|  |  |
|  | // We don't know how many parameters the function wants, so we just |
|  | // push them all onto the stack in the correct order. The function will |
|  | // use all the parameters it wants, and we can pop them all back off afterwards. |
|  | int ret; |
|  | asm volatile (" \ |
|  | push %1; \ |
|  | push %2; \ |
|  | push %3; \ |
|  | push %4; \ |
|  | push %5; \ |
|  | call \*%6; \ |
|  | pop %%ebx; \ |
|  | pop %%ebx; \ |
|  | pop %%ebx; \ |
|  | pop %%ebx; \ |
|  | pop %%ebx; \ |
|  | " : "=a" (ret) : "r" (regs->edi), "r" (regs->esi), "r" (regs->edx), "r" (regs->ecx), "r" (regs->ebx), "r" (location)); |
|  | regs->eax = ret; |
|  | } |

Итак, здесь у нас есть таблица адресов наших функций системных вызовов. Функция initialise\_syscalls всего лишь добавляет функцию обработчика syscall\_handler в качестве обработчика для прерывания 0x80.

Функция syscall\_handler проверяет, является ли индекс данной функции допустимым, затем берет адрес вызываемой функции, а также помещает в стек все параметры, которые мы используем, вызывает функцию, а затем убирает из стека все параметры.

Как обычно, она помещает в регистр EAX значение, которое должно быть возвращено функцией при выходе из прерывания.

#### 10.2.3. Вспомогательные макросы

Итак, системный вызов, осуществляемый из пользовательского режима, должен выглядеть приблизительно так:

mov eax, call>

mov ebx,

mov ecx,

mov edx,

mov esi,

mov edi,

int 0x80 ; execute syscall

; значение, возвращаемое из системного вызова, находится в EAX.

Однако, он выглядит несколько громоздким. Мы можем упростить его, создав несколько вспомогательных макросов, в которых определены функции-заглушки с ассемблерными строками, с помощью которых в действительности выполняется системный вызов;

*В файле syscall.h*

#define DECL\_SYSCALL0(fn) int syscall\_##fn();

#define DECL\_SYSCALL1(fn,p1) int syscall\_##fn(p1);

#define DECL\_SYSCALL2(fn,p1,p2) int syscall\_##fn(p1,p2);

#define DECL\_SYSCALL3(fn,p1,p2,p3) int syscall\_##fn(p1,p2,p3);

#define DECL\_SYSCALL4(fn,p1,p2,p3,p4) int syscall\_##fn(p1,p2,p3,p4);

#define DECL\_SYSCALL5(fn,p1,p2,p3,p4,p5) int syscall\_##fn(p1,p2,p3,p4,p5);

#define DEFN\_SYSCALL0(fn, num) \

int syscall\_##fn() \

{ \

int a; \

asm volatile("int $0x80" : "=a" (a) : "0" (num)); \

return a; \

}

#define DEFN\_SYSCALL1(fn, num, P1) \

int syscall\_##fn(P1 p1) \

{ \

int a; \

asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1)); \

return a; \

}

#define DEFN\_SYSCALL2(fn, num, P1, P2) \

int syscall\_##fn(P1 p1, P2 p2) \

{ \

int a; \

asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2)); \

return a; \

}

...

Итак, у нас есть макрос "DECL\_SYSCALLX", в котором объявляется функция-заглушка для функции *fn* с параметрами X, причем типы параметров определяются как *p1..pn*.

В макросе "DEFN\_SYSCALLX" фактически определяется функция-заглушка, которая является просто фрагментом вставленного кода на ассемблере. Параметр num используется как индекс таблицы функций системных вызовов.

Таким образом, чтобы определить наши функции monitor\_ \*, нам потребуется объявить их в файле syscall.h:

DECL\_SYSCALL1(monitor\_write, const char\*)

DECL\_SYSCALL1(monitor\_write\_hex, const char\*)

DECL\_SYSCALL1(monitor\_write\_dec, const char\*)

а затем в файле syscall.c определяем:

DEFN\_SYSCALL1(monitor\_write, 0, const char\*);

DEFN\_SYSCALL1(monitor\_write\_hex, 1, const char\*);

DEFN\_SYSCALL1(monitor\_write\_dec, 2, const char\*);

### 10.3. Тестирование

*В файле main.c*

// Запускам страничную организацию памяти.

initialise\_paging();

// Запускаем многозадачность.

initialise\_tasking();

// Инициализируем initial ramdisk и указываем его как корневую файловую систему.

fs\_root = initialise\_initrd(initrd\_location);

initialise\_syscalls();

switch\_to\_user\_mode();

syscall\_monitor\_write("Hello, user world!\n");

return 0;

код, размещенный в файле main.c, дает функциональные возможности пользовательского режима, предназначенного для выполнения ненадежных программ пользователя.

**Выводы**

* Понимание низкоуровневых аспектов языков программирования. Работа с регистрами.
* Изучение соглашения о вызовах \_\_cdecl, который устанавливает следующий порядок: параметры функции загружаются в стек (push) в порядке справа налево. Т.е. самый последний аргумент функции будет загружен в стек последним. Для возврата результата используется регистр eax.
* Более глубокое понимание псевдо-параллельности: на машине с одним ядром многозадачность достигается с помощью очень частых (примерно каждые 45 мкс на современных машинах) переключений контекста, что представляет из себя достаточно "дешевую" операцию.