Методические указания к выполнению лабораторных работ по дисциплине «Операционные системы»

Инструментарий, необходимый для выполнения лабораторных работ

1. ОС GNU/Linux
2. GCC (http://mirror.tochlab.net/pub/gnu/gcc/gcc-4.9.0/gcc-4.9.0.tar.gz)
3. BinUtils (http://mirror.tochlab.net/pub/gnu/binutils/binutils-2.9.1.tar.gz)
4. GDB (http://mirror.tochlab.net/pub/gnu/gdb/gdb-7.7.tar.gz)
5. QEMU (http://wiki.qemu-project.org/download/qemu-2.1.1.tar.bz2)

Лабораторная работа №2

Теоретический материал

Основной целью выполнения лабораторных работ является знакомство с устройством ядра операционной системы (ОС) на примере учебной ОС – JOS. Исследование и разработка JOS будут осуществляться с применением эмулятора аппаратного обеспечения различных платформ QEMU. QEMU может работать в двух режимах:

- полнофункциональная эмуляция. В этом режиме QEMU эмулируют систему в целом, включая один или несколько процессоров и различные периферийные устройства. Может быть использован для загрузки различных операционных систем без или для отладки системного кода;
- эмуляция режима пользователя. В этом режиме QEMU может загружать процессы, представленные программами, откомпилированными на одной платформе для другой платформы. Может использоваться для загрузки эмулятора Wine Windows API (http://www.winehq.org) или для облегчения кросс-компиляции и кросс-отладки.

Среди поддерживаемых QEMU аппаратных платформ[1]: PC (x86 или x86_64), ISA PC (PC без шины PCI), PREP (PowerPC processor), Sun4m/Sun4c/Sun4d (32-разрядный процессор Sparc), различные ARM платформы (Luminary Micro, MusicPal) и т.д.

Использование QEMU облегчает отладку ядра операционной системы, например, установку точек останова. Встроенный монитор QEMU обеспечивает лишь ограниченную поддержку при отладке, но QEMU может выступать в роли объекта для удаленной отладки с помощью GDB.

Для выполнения лабораторных работ установите QEMU (см. Приложение «Установка QEMU») и GDB на своей машине. Скачайте с сайта учебного курса архив JOSlabs.tar.gz и разархивируйте его в свой домашний каталог. Перейдите в каталог JOS/conf и в файле env.mk укажите путь, по которому установлен QEMU, например:

QEMU=/usr/local/bin/qemu-system-i386

Выполните команду make для сборки ядра JOS. В результате в командной оболочке будет выведена следующая информация:

```
[user@localhost JOS]$ make
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
```

Теперь можно запускать QEMU, используя файл `obj/kern/kernel.img` в качестве содержимого виртуального жесткого диска для эмулируемого ПК. Этот образ жесткого диска содержит загрузчик (`obj/boot/boot`) и ядро ОС (`obj/kernel`). Команда `make qemu`

[1] Полный список поддерживаемых QEMU аппаратных платформ можно найти в пользовательской документации: http://qemu.weilnetz.de/qemu-doc.html.

выполнит запуск с опциями, необходимыми для установки жесткого диска и прямого вывода содержимого последовательного порта на терминал. В результате выполнения команды появится следующий текст в терминале:

```
[user@localhost JOS]$ make qemu
/usr/local/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial
mon:stdio -gdb tcp::26000 -D qemu.log
VNC server running on `127.0.0.1:5900'
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Все после «`Booting from Hard Disk...`» выведено ядром JOS. `K>` – приглашение для ввода команд, которое выводится небольшой программой-монитором (`monitor.c`, `monitor.h`), включенной в ядро ОС. Ядро JOS выводит сообщения не только на виртуальный VGA дисплей, но также и в виртуальный последовательный порт, поэтому команды можно вводить и в окне терминала, в котором запущен QEMU.

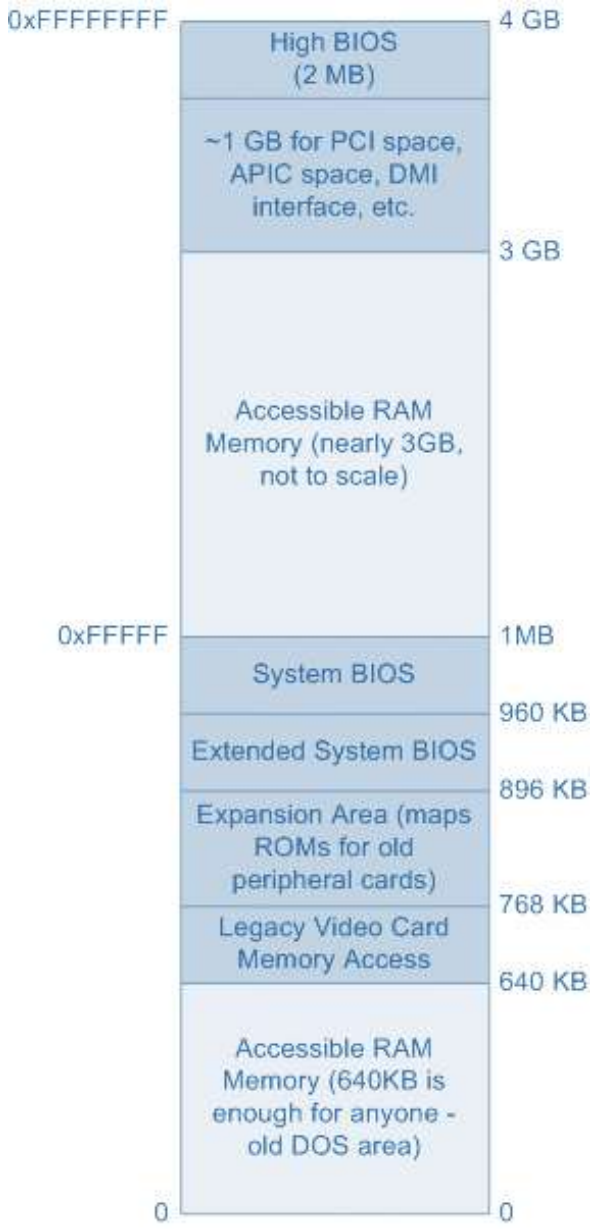
В настоящий момент поддерживаются только две команды: `help` и `kerninfo`.

```
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
K> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101a47 (virt)  00101a47 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112944 (virt)  00112944 (phys)
Kernel executable memory footprint: 75KB
K>
```

Команда `help` выводит справочную информацию о существующих командах. Команда `kerninfo` выводит виртуальные и соответствующие им физические адреса

Физическое адресное пространство

Физическое адресное пространство памяти для 32-разрядных ПК распределено следующим образом:



Первые ПК, которые базировались на 16-разрядных процессорах, могли адресовать только 1 Мб физической памяти. Физическое адресное пространство таких ПК начиналось с адреса 0x00000000, а заканчивалось адресом 0x000FFFFF. Область памяти в 640 Кб, расположенная в нижних адресах, была единственным оперативным запоминающим устройством (ОЗУ), которое могли использовать 16-разрядные ПК.

При этом область в 384 Кб, начиная с адреса 0x000A0000 и до 0x000FFFFF была зарезервирована аппаратным обеспечением для специальных целей, например, для буферов видеоизображения и микропрограмм периферийных устройств. Наиболее важная часть этой зарезервированной области памяти – Basic Input/Output System (BIOS), занимающая область памяти размером 64 Кб с адреса 0x000F0000 до адреса 0x000FFFFF. В первых ПК микропрограмма BIOS хранилась в постоянном запоминающем устройстве (ПЗУ), в настоящее время микропрограмма BIOS записана в перезаписываемую флэш-память. BIOS отвечает за выполнение инициализации системы: активация видеокарты, проверка установленной памяти и определение ее размера и т.д. После выполнения этих функций BIOS загружает ОС с загрузочного диска, тем самым передавая управление ПК операционной системе.

Когда исторический барьер в 1 Мб был преодолен, появилась возможность адресации до 4 Гб памяти. Однако распределение младшего Мб этого адресного пространства осталось прежним, для того чтобы сохранить совместимость новой архитектуры со старым программным обеспечением. В современных ПК существует «дыра» в физической памяти, начиная с адреса 0x000A0000 и до 0x00100000, которая разделяет ОЗУ на «нижнюю» или «обычную» память (первые 640 Кб) и «расширенную» память (остальное адресное пространство). Кроме этого, часть адресного пространства, находящаяся над доступным ОЗУ для программ, резервируется BIOS для 32-разрядных PCI устройств.

Современные 32-разрядные процессоры могут работать с физическим адресным пространством, превышающим 4 Гб, т.е. ОЗУ может быть расширено за пределы адреса 0xFFFFFFFF. В этом случае образуется вторая «дыра» в адресном пространстве для отображения 32-разрядных устройств.

Учебная операционная система JOS, применяемая для выполнения лабораторных работ, будет использовать первые 256 Мб физической памяти, т.о. предполагается работа с 32-разрядным физическим адресным пространством.

Задание 1

В этой части лабораторной работы будут использоваться отладочные возможности QEMU для понимания того, как происходит загрузка 32-разрядного ПК.

Для выполнения задания откройте два терминальных окна. В одном введите команду `make qemu-gdb`. Эта команда запустит QEMU, но QEMU остановится сразу же, как только процессор выполнит первую команду и будет ожидать запуска GDB. Во втором терминале в том же каталоге, где Вы запустили команду `make`, выполните команду `gdb`. Вы увидите следующую информацию в окне терминала:

```
[user@localhost JOS]$ gdb
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:1234
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

В каталоге JOS присутствует файл `.gdbinit`, который служит для настройки GDB по выполнению отладки 16-разрядного программного кода, используемого на ранней стадии загрузки ПК. Этот файл нужно скопировать в домашний каталог пользователя перед

запуском GDB и вставить в конец этого файла строку: «add-auto-load-safe-path /home/username[2]/JOS/.gdbinit».

Следующая строка:

```
[f000:fff0] 0xffff0:     ljmp   $0xf000,$0xe05b
```

является результатом дизассемблирования отладчиком GDB первой выполняемой команды. Из этой строки можно сделать вывод, что:

- загрузка начинается с физического адреса `0xffff0`, который соответствует вершине 64-Кбайтной области, зарезервированной для BIOS;
- при этом значение сегментного регистра `CS = 0xf000`, а регистра счетчика команд `IP = 0xfff0`;
- первая выполняемая команда – `jmp`, которая выполняет переход к адресу `IP = 0xe05b` сегмента `CS = 0xf000`.

Почему QEMU начинает свою работу именно так? Потому что это соответствует режиму работы IBM-совместимых ПК, начиная с процессора Intel 8088. BIOS всегда отображается на физическое адресное пространство в диапазоне адресов `0x000f0000-0x000fffff`, что обеспечивает возможность захвата им контроля над системой после нажатия кнопки включения ПК или после его перезагрузки. Эмулятор QEMU имеет свой собственный BIOS, который отображается на данную область эмулируемого физического адресного пространства. После выполнения сброса процессора он входит в реальный режим работы, устанавливает значение сегментного регистра `CS` в `0xf000`, а значение счетчика команд `IP` в `0xfff0`, т.е. загрузка начинается с адреса, формируемого как `CS:IP`. Как происходит преобразование сегментного адреса `0xf000:fff0` в физический?

В реальном режиме преобразование адреса происходит в соответствии со следующей формулой: *physical address* = 16 * *segment* + *offset*. Т. е. для нашего случая преобразование выглядит так:

```
16 * 0xf000 + 0xfff0 = 0xf0000 + 0xfff0 = 0xffff0
```

Задание 2

Используя команду GDB для пошагового выполнения – `si` – выполните еще несколько команд BIOS и объясните их назначение[3].

BIOS настраивает таблицу дескрипторов прерываний и инициализирует различные устройства, такие как, например, дисплей VGA. После этого становится возможным вывод различных диагностических сообщений на экран ПК.

После инициализации шины PCI и всех важных устройств, о которых «знает» BIOS, начинается поиск загрузочного диска. Если такой диск найден, BIOS находит программу-загрузчик на этом диске и передает управление ей.

---

[2] Вместо “username” нужно указать имя своей учетной записи в GNU/Linux.

[3] Справочную информацию о прерываниях можно найти в Phil Storrs PC Hardware book и BIOS Services and Software Interrupts.

Загрузчик

Жесткие диски разделены на области, называемые *секторами* (обычно размером в 512 байт). Сектор является минимальной областью диска, используемой при выполнении операций ввода-вывода на него. Первый сектор загрузочного диска называется *загрузочным сектором*. Здесь расположен код программы-загрузчика ОС. Когда BIOS обнаруживает загрузочный диск, он загружает содержимое загрузочного сектора в память, начиная с физического адреса `0x7c00` и до адреса `0x7dff`, а затем использует команду `jmp` для установки регистров `CS:IP` to `0000:7c00`, тем самым передавая управление загрузчику.

Загрузчик учебной ОС JOS состоит из одного исходного файла на языке Ассемблер `boot/boot.S` и одного исходного файла на языке программирования Си `boot/main.c`. Загрузчик выполняет две главные функции:

1. переключение процессора из реального режима работы в 32-разрядный защищенный режим, потому что только в этом режиме возможна адресация более 1 Мб памяти;
2. чтение кода ядра ОС.

Файл `obj/boot/boot.asm` является результатом дизассемблирования загрузчика. С помощью этого файла можно увидеть, где именно в физической памяти находится код загрузчика, а также проследить, что происходит, когда GDB пошагово выполняет загрузчик. Файл `obj/kern/kernel.asm` содержит результат дизассемблирования ядра JOS, который может понадобиться при отладке.

С помощью команды `b` отладчика GDB можно устанавливать точки останова. Например, `b *0x7c00` устанавливает точку останова по адресу `0x7C00`. Для того чтобы продолжить выполнение, используйте команды `c` и `si`. Команда `c` заставляет QEMU продолжить выполнение до следующей точки останова (или до тех пор, пока в GDB не будет нажато сочетание клавиш `Ctrl-C`).

Используйте команду `x/i` для того, чтобы вывести несколько следующих команд. Синтаксис этой команды – `x/Ni ADDR`, где `N` – количество команд, которые должны быть дизассемблированы, а `ADDR` – адрес в памяти, с которого следует начать дизассемблирование.

Эти и другие команды отладчика GDB приведены в Приложении 2.

Задание 3

Установите точку останова по адресу `0x7c00`, куда загружается содержимое загрузочного сектора диска. Продолжите выполнение команд до этой точки останова. Используя команду `x/i` в GDB дизассемблируйте несколько команд загрузчика, сравните исходный код загрузчика в файле `boot/boot.S` с результатом дизассемблирования в `obj/boot/boot.asm` и GDB.

Зайдите в функцию `bootmain()` в файле `boot/main.c`, а затем в `readsect()`. Определите команды ассемблера, которые соответствуют каждой инструкции в `readsect()`. Отследите функцию `readsect()` до конца и вернитесь в `bootmain()`,

определите начало и конец цикла `for`, считывающего оставшиеся секторы ядра с диска. Определите, какой код запустится, когда цикл завершится, установите там точку останова и продолжите выполнение до этой точки. Затем выполните по шагам оставшуюся часть загрузчика.

Ответьте на следующие вопросы:

- В какой точке процессор начинает выполнение 32-разрядного кода? Что именно вызывает переключение из 16-разрядного в 32-разрядный режим?
- Какова *последняя* выполняемая команда загрузчика и какова *первая* команда ядра?
- *Где* размещается первая команда ядра?
- Как загрузчик решает, сколько секторов он должен прочитать, для того чтобы получить все ядро с диска? Где он находит эту информацию?

Загрузка ядра

Теперь подробнее рассмотрим код загрузчика, написанный на языке программирования Си (файл `boot/main.c`).

При компиляции и компоновке программы, написанной на Си, компилятор преобразует файл-исходник (`*.c`) в объектный файл (`*.o`), содержащий ассемблерные команды, закодированные в двоичном формате. Компоновщик затем соединяет все скомпилированные объектные файлы в один *двоичный образ* такой, как, например, файл `obj/kern/kernel` учебной ОС JOS. Формат этого файла – Executable and Linkable Format (ELF)[4]. ELF файл содержит заголовок с загрузочной информацией и программные разделы, каждый из которых представляет собой непрерывный участок кода или данных, загружаемый в память, начиная с определенного адреса. Загрузчик не изменяет код или данные, он только загружает их в память и начинает выполнение.

Заголовок ELF файла имеет фиксированную длину, за ним следует *программный заголовок* переменной длины. Определения на языке Си для этих заголовков находятся в файле `inc/elf.h`. Нас будут интересовать следующие программные разделы:

- `.text`: исполняемые команды программы;
- `.rodata`: неизменяемые данные, такие как строковые константы;
- `.data`: раздел данных, содержащий инициализированные переменные, такие как глобальные переменные, объявленные с инициализацией, например, `int x = 5;`.

Когда компоновщик определяет распределение памяти программы, он резервирует место для неинициализированных глобальных переменных, например, `int x;` в разделе `.bss`, который следует в памяти непосредственно за разделом `.data`.

Изучите полный список имен, размеров и адресов всех разделов ядра с помощью команды:

```
[user@localhost JOS]$ objdump -h obj/kern/kernel
```

[4] Подробно о формате исполняемого файла ELF можно прочитать в http://www.uclibc.org/docs/elf.pdf и в http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf.

Обратите внимание на "VMA" (или связующий адрес) и "LMA" (или загрузочный адрес) раздела `.text`. Загрузочный адрес раздела является тем адресом в памяти, начиная с которого раздел будет размещен в памяти. Этот адрес хранится в поле `ph->p_pa`.

Связующий адрес – это адрес, с которого начинается исполнение раздела. Часто связующий и загрузочный адреса совпадают. Например, в разделе `.text` загрузчика:

```
[user@localhost JOS]$ objdump -h obj/boot/boot.out
```

BIOS загружает загрузчик в память, начиная с адреса `0x7c00`, это и есть загрузочный адрес загрузочного сектора. В то же время выполнение загрузчика тоже начинается с этого адреса, так что это и связующий адрес. Для установки связующего адреса нужно использовать опцию `-Ttext 0x7C00` для компоновщика в файле `boot/Makefrag`, тогда компоновщик будет правильно определять адреса памяти в генерируемом коде.

Задание 4

Измените связующий адрес в файле `boot/Makefrag` на неправильное значение, выполните команду `make clean`, перекомпилируйте код JOS с помощью `make` и выполните загрузчик пошагово, чтобы увидеть, что произойдет в этом случае. Не забудьте после этого изменить связующий адрес на правильное значение и снова выполните `make clean`.

В отличие от загрузчика загрузочный и связующий адреса ядра ОС не совпадают.

Еще одно поле присутствует в заголовке ELF файла – поле `e_entry`. Это поле содержит связующий адрес *точки входа* в программу: адрес в разделе кода программы, с которого начинается выполнение программы. Точку входа можно увидеть, запустив команду:

```
[user@localhost JOS]$ objdump -f obj/kern/kernel
```

Загрузчик в файле `boot/main.c` считывает каждый раздел ядра с диска в память, а затем переходит к точке входа.

Ядро

Теперь рассмотрим ядро JOS более подробно.

Использование виртуальной памяти для решения проблемы позиционной зависимости

Ядра ОС обычно связываются и запускаются в старших виртуальных адресах, таких как `0xf0100000`, чтобы младшая часть виртуального адресного пространства могла быть использована пользовательскими приложениями.

В случае JOS мы будем использовать процессорное устройство управления памятью для отображения виртуального адреса `0xf0100000` в физический адрес `0x00100000`. Т.е. мы просто отображаем первые 4 Мб физической памяти, что достаточно для запуска ОС. Это отображение выполняется в файле `kern/entrypgdir.c` с помощью статически инициализированной таблицы страниц. До тех пор пока флаг `CR0_PG` в `kern/entry.S` не установлен, все обращения к памяти обрабатываются как физические адреса. Как только

CR0_PG устанавливается, все обращения к памяти воспринимаются как виртуальные адреса и транслируются в физические. entry_pgdir транслирует виртуальные адреса в диапазоне от 0xf0000000 до 0xf0400000 в физические в диапазоне от 0x00000000 до 0x00400000, а также виртуальные адреса от 0x00000000 до 0x00400000 в физические от 0x00000000 до 0x00400000. Любой виртуальный адрес не из этих диапазонов вызовет аппаратное исключение, поскольку обработка прерываний пока еще не реализована. QEMU при этом сформирует дамп состояния ПК и закончит свою работу.

Форматированный вывод на консоль

Задание 5

Возможность вывода в консоль реализована в файлах kern/printf.c, lib/printfmt.c и kern/console.c. В этих файлах отсутствует возможность вывода восьмеричных чисел в форме "%o". Вставьте необходимый код для вывода таких чисел.

Ответьте на следующие вопросы:

1. Объясните взаимосвязь printf.c и console.c. Какая функция экспортируется в console.c? Как эта функция используется в printf.c?
2. Объясните следующий фрагмент кода из файла console.c:

```
1       if (crt_pos >= CRT_SIZE) {
2         int i;
3         memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
                                                 sizeof(uint16_t));
4         for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5           crt_buf[i] = 0x0700 | ' ';
6           crt_pos -= CRT_COLS;
7       }
```

3. Что будет выведено на консоль после 'y=' при выполнении следующего кода?

```
cprintf("x=%d y=%d", 3);
```

Стек

Задание 6

Определите, где в ядре происходит инициализация стека, а также, где стек располагается в памяти. Как ядро резервирует место для стека?

32-разрядный регистр esp (указатель стека) содержит самый младший (нижний) адрес стека, указывающий на последние добавленные в стек данные. Вся остальная область, зарезервированная для стека ниже этого адреса, свободна. При добавлении значения в стек esp уменьшается, а затем происходит запись значения по адресу, на который теперь указывает esp. При извлечении значения из стека происходит чтение данных по адресу, на который указывает esp, а потом значение указателя стека

увеличивается. В 32-разрядном режиме стек может содержать только 32-разрядные значения, и значение `esp` всегда кратно 4.

Регистр `ebp` (базовый указатель) используется для сохранения значения регистра `esp` перед выполнением функции программы на Си. При этом старое значение `ebp` помещается в стек. Если все функции в программе придерживаются этого соглашения, то в любой момент выполнения программы можно пройти назад по стеку, следуя по цепочке сохраненных указателей `ebp`, и определить, какая вложенная последовательность вызовов функций стала причиной достижения в программе данной конкретной точки.

Задание 7

Найдите адрес функции `test_backtrace` в `obj/kern/kernel.asm`, установите там точку останова и проверьте, что происходит каждый раз при вызове этой функции, когда запускается ядро. Сколько 32-разрядных слов записывается в стек каждым вызовом `test_backtrace` и что это за слова?

Реализуйте функцию `mon_backtrace()` – функцию обратной трассировки стека. Прототип этой функции уже есть в файле `kern/monitor.c`. При реализации можете воспользоваться функцией `read_ebp()`, объявленной в `inc/x86.h`. Добавьте команду монитора ядра для вызова этой функции в интерактивном режиме пользователем.

Функция обратной трассировки должна выводить информацию о фрэймах вызовов функций в следующем формате:

```
Stack backtrace:
ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

Первая строка содержит информацию о текущей выполняемой функции, т.е. о самой `mon_backtrace`, вторая строка содержит информацию о функции, которая вызвала `mon_backtrace`, третья строка – информацию о функции, которая вызвала эту функцию и т.д.

Значение `ebp` содержит базовый указатель стека, используемый функцией, т.е. позицию указателя стека сразу после того, как функция была вызвана. Значение `eip` – это адрес возврата из функции: адрес команды, к которой вернется управление после выхода из функции. Обычно это команда после `call`. Наконец, пять шестнадцатеричных значений, записанных после `args` – это первые пять аргументов функции, которые помещаются в стек непосредственно перед вызовом функции. Если функция была вызвана с меньшим количеством аргументов, то естественно не все эти значения используются.

На практике не адреса вызывающих функций, а их имена могут иметь значение. Например, если нужно узнать, какие функции могут содержать ошибки, вызвавшие сбой ядра.

Чтобы реализовать эту возможность, в файле `kern/kdebug.c` определена функция `debuginfo_eip()`, которая осуществляет поиск `eip` в таблице символов и возвращает отладочную информацию для конкретного адреса.

Приложение 1
Установка QEMU

Ссылка для скачивания QEMU:

http://wiki.qemu-project.org/download/qemu-2.1.1.tar.bz2.

Перед установкой QEMU убедитесь, что у Вас установлены следующие пакеты:

- ✓ gcc
- ✓ gcc-c++
- ✓ libtool
- ✓ zlib-devel
- ✓ glib2-devel

Приложение 2
Команды GDB[5]

В данном приложении приведены команды, необходимые для выполнения лабораторных работ.

`Ctrl-C`

Останов виртуальной машины и GDB на текущей команде.

`c (или continue)`

Продолжение выполнения до следующей точки останова или нажатия **`Ctrl-C`**.

`si (или stepi)`

Выполнение одной машинной команды.

`b function или b file:line (или breakpoint)`

Установка точки останова на текущей функции или строке.

`b *addr (или breakpoint)`

Установка точки останова по адресу *addr*.

`set print pretty`

Структурированная распечатка массивов и структур.

`info registers`

Вывод содержимого регистров общего назначения, регистров EIP, EFLAGS и сегментных регистров.

`x/Nx addr`

Вывод шестнадцатеричного дампа *N* слов, начиная с виртуального адреса *addr*. Если *N* не указано, то по умолчанию используется 1.*addr*.

`x/Ni addr`

[5] С полным списком команд GDB можно ознакомиться в справочном руководстве man или в справочнике по отладке с применением GDB: https://sourceware.org/gdb/current/onlinedocs/gdb/.

Вывод *N* ассемблерных команд, начиная с адреса *addr*. При использовании `$eip` в качестве *addr* выводятся команды, расположенные по адресу, хранящемуся в `eip`.

`symbol-file file`

Переключение в бинарный файл *file*.

`thread n`

GDB фокусируется на одном потоке в один момент времени. Эта команда переключает фокус в поток с номером *n* (нумерация начинается с нуля).

`info threads`

Список всех потоков, включая их состояние (активный или приостановленный) и в какой функции они находятся.