| | |
|---|---|
| 1. | Принципы фон-Неймана |
| 3. | ~~Дан следующий фрагмент кода на Си:~~<br>~~int func(void) {~~<br>~~  int i, s = 0;~~<br>~~  int A[5] = {1, 2, 3};~~<br>~~  for (i = 0; i < 5; i++)~~<br>~~    s = s + A[i];~~<br>~~  return s;~~<br>~~}~~<br>~~Какое значение вернёт эта функция? Почему?~~<br>~~*Что изменится, если массив A сделать глобальным/статическим? Почему?*~~<br>~~—"—~~<br>~~Для массива A инициализируются только первые 3 элемента. Оставшиеся два рассматриваются как неинициализированные переменные.~~<br>По одной из версий стандарта, остальные инициализируются нулями, даже если массив в стеке. То же самое с полями структур/объектов. Другое дело, что не все компиляторы следуют стандарту, в данном случае. (Контрпримеры???) |
| 4. | Сколько параметров имеет функция printf?<br>Каков принцип её работы с параметрами?<br>Каков принцип её работы с числами и символами?<br>Переменное число параметров. Это оказалось возможным благодаря соглашению **cdecl**: параметры функции передаются через стек (в обратном порядке), очистку стека производит **вызывающая** функция, которая уж наверняка знает, сколько конкретно параметров было передано вызываемой. (Возвращает результат функция через регистр ax/eax/rax.) Первым параметром функции является ***шаблон*** строки, в стек указатель на него заносится самым последним, т.е. находится на вершине стека. Таким образом, функция printf знает, что в вершине стека лежит указатель на строчку, которая определяет, сколько ещё параметров находится в стеке помимо него (по числу спецификаторов, например, %i). Анализируя шаблон, printf последовательно для каждого спецификатора читает из стека соответствующий параметр, поэтому если спецификаторов окажется больше, printf возьмёт из стека слово, которое уже не является параметром функции printf…<br>Поскольку терминал (окно консоли) отображает только символы из таблицы ASCII, то вывод переменных типа char и char* (строчек) происходит относительно тривиально (функции передаётся число, она выводит на экран символ с соответствующим кодом ASCII). А вот переменные целых типов выводятся/вводятся поразрядно, с использованием алгоритмов по переводу чисел из одной системы счисления в другую, которые мы разбирали на 1-м семинаре. |
| 9. | Каков порядок вычисления операндов в операции &<br>(то есть, какой операнд вычисляется первым: левый или правый)?<br>Покажите это на практике (с использованием Си или с использованием Ассемблера).<br>**Не специфицирован.**<br>Зависит не только от реализации используемого компилятора, от целевой архитектуры, но может зависеть и от конфигурации (Debug/Release, например, в M$ VC++).<br>Проверить это можно, либо установив breakpoint на интересующее выражение и открыв окошко дизассемблера, либо, например, таким кодом:<br>`int func1(int i) {`<br>`  printf("func1: %i\n", i);`<br>`  return i;`<br>`}`<br>`int func2(int i) {`<br>`  printf("func2: %i\n", i);`<br>`  return i;`<br>`}`<br>`…`<br>`  if (func1(1) & func2(2))` |

| | |
|---|---|
| | printf("!=\n"); |
| 14. | Написать простейшую программу, которая зацикливается.<br>(На Си и/или на Ассемблере.)<br>void main() {<br>while (1);<br>}<br><br>.model small<br>.code<br>Entry: jmp Entry<br>end Entry |
| 25. | Что такое бит? |
| 26. | Что такое байт? |
| 27. | Что такое переменная? |
| 28. | Что такое массив? |
| 29. | Что такое стек и для чего он нужен? |
| 30. | Что такое прерывание? |
| 31. | Что такое процедура/функция? |
| 32. | Что такое рекурсия? |
| 33. | Что такое макрос? |
| 34. | В чём принципиальное (фундаментальное) отличие директив от инструкций?<br>(Как в языке Си, так и в Ассемблере.) |
| 35. | Можно ли в макросах на Ассемблере использовать циклы, условные переходы?<br>Почему?<br>~~Нет, циклы и условные переходы нельзя. Так как и то и другое реализовано с помощью меток перехода, то если один и тот же макрос вызвать в программе хотя бы два раза, в текст программы попадёт две метки перехода с одним именем, что вызовет ошибку компиляции.~~<br>*Поправка:* *если использовать относительный адрес в командах перехода, а не метки, то можно.*<br>*Ещё поправка:* *ещё существуют локальные метки. Например:*<br>*local Lbl*<br>*je Lbl*<br>*Lbl:*<br>*Или в MASM'е:*<br>*@@Lbl:*<br>*je @@Lbl* |
| 36. | Можно ли в макросах на Ассемблере использовать вызовы процедур?<br>Можно, функции для того и придуманы, чтобы их можно было многократно вызывать командой call.<br>См. пример вызов функции putnum (определённой в модуле ioproc.asm) из макроса putn из io.asm. |
| 38. | Какие Вам известны типы памяти? |
| 40. | Массив и списки: сходства, отличия, достоинства, недостатки.<br>И массивы, и списки используются для хранения данных, набора элементов одного размера. Для массива память выделяется сразу для всех элементов, а для списка – по кусочкам для каждого элемента, по мере необходимости.<br>***Массивы:***<br>**Достоинства:**<br>1) легкость вычисления адреса элемента по его индексу (поскольку элементы массива располагаются один за другим), легко определить количество элементов, т.к. оно жёстко задано или хранится отдельно;<br>2) одинаковое время доступа ко всем элементам;<br>3) малый размер элементов: они состоят только из информационного поля;<br>4) легко определять массив и работать с ним.<br>**Недостатки:** |

| | |
|---|---|
| | 1) для статического массива – отсутствие динамики, невозможность удаления или добавления элемента без сдвига других (при добавлении может потребоваться перераспределение памяти для **всего** массива);<br>*2) для динамического и/или гетерогенного массива – более низкое (по сравнению с обычным статическим) быстродействие и дополнительные накладные расходы на поддержку динамических свойств и/или гетерогенности;*<br>3) при работе с массивом в стиле C (с указателями) и при отсутствии дополнительных средств контроля – угроза выхода за границы массива и повреждения данных.<br>***Списки:***<br>**Достоинства (по сравнению с массивами):**<br>1) возможность добавлять, удалять и перемещать элементы, не трогая при этом другие (только перенаправив ссылки).<br>**Недостатки (по сравнению с массивами):**<br>1) ниже быстродействие;<br>2) накладные расходы на связь элементов (ссылки);<br>3) сложнее в реализации.<br>…Что-то ещё?.. |
| 41. | В чём сходство и различие между одномерными статическим и динамическим массивами? Могут ли они использоваться друг вместо друга?<br>В чём сходство и различие между двумерными статическим и динамическим массивами? Могут ли они использоваться друг вместо друга?<br>Память под статический массив выделяется на этапе компиляции, а под динамический – на этапе выполнения, специальными вызовами. Имя массива является указателем на его 0-й элемент, для статического массива этот указатель константный. Везде, где можно использовать одномерный статический массив, можно использовать и динамический. Обратное не всегда верно: нельзя выделить/освободить память под статический массив по желанию, нельзя переменной-имени массива присвоить другое значение.<br>N-мерный статический массив – это массив (N-1)-мерных массивов. Однако элемент двумерного статического массива является массивом (и имеет соответствующий размер), а элемент динамического массива – указатель на одномерный массив (размер 4 байта; в 64-битных ОС 8 байт). Поэтому для двумерного статического массива верны равенства:<br>&A == A == &A[0] == A[0] == &A[0][0],<br>а для двумерного динамического:<br>&B != B == &B[0] != B[0] == &B[0][0]<br>Поэтому двумерный статический массив ещё нельзя, например, передавать в функцию, которая в качестве параметра принимает двумерный динамический массив. |
| 42. | * С каким типом памяти *(оперативной, виртуальной, внешней)* работают инструкции Ассемблера в загрузочных секторах (MBR, VBR)? Можно ли в том коде использовать прерывание 21h?<br>Как только процессор включается, он начинает работать в **реальном режиме**, что означает, что никакой виртуальной памяти ещё нет, и обращение идёт непосредственно к оперативной памяти.<br>Прерывание 21h использовать нельзя, т.к. это прерывание, определяемое DOS, а на этапе выполнения MBR и VBR никакой DOS (ещё) не загружен. Но можно использовать, например, прерывания BIOS (10h, 13h). |
| 48. | Взаимооднозначное ли ассемблирование и дизассемблирование? Можно ли, написав программу на ассемблере, скомпилировав, а потом дизассемблировав исполняемый .EXE-файл, получить исходный код программы, как в .ASM-программе? |
| 50. | Где в памяти хранятся данные разных типов (целые, вещественные, символьные, строковые, сложные типы данных)?<br>Чаще всего:<br>Целочисленные и символьные (тоже, на самом деле, целочисленные) константы – в секции кода, подставляются прямо в команды процессора;<br>Строковые, вещественные константы и константы сложных структур – в глобальной секции |

| | |
|---|---|
| | данных (иногда защищённой от записи);<br>локальные автоматические переменные – в стеке;<br>инициализированные глобальные и статические переменные – в секции данных;<br>неинициализированные глобальные и статические переменные – в секции BSS (которая отсутствует в образе исполняемого модуля на диске, но выделяется в памяти в процессе загрузки программы и инициализируется нулями);<br>динамические переменные (память для которых выделяется вызовами malloc() и new) – в **куче**. |
| 54. | Можно ли в программе операцию "or" заменить на операцию "add"?<br>*При каких условиях можно?*<br>Только если соответствующие биты операндов, выставленные в 1, имеют пустое пересечение. Если после<br>`and ax, bx`<br>значение ax равно 0. |
| 56. | Перевести десятичную дробь 10.25 в двоичную.<br>Перевести в нормализованное представление.<br>**$1010.01_2 = 1.01001E11_2$** |
| 57. | Перевести десятичную дробь 10.1 в двоичную.<br>Перевести в нормализованное представление.<br>**$1010.0(0011)_2 = 1.0100(0011)E11_2$** |
| 63. | Что такое прямой, обратный, дополнительный код?<br>В каком из них записываются отрицательные числа в архитектуре x86? Почему? |
| 64. | Можно ли на Си или Ассемблере написать универсальную функцию, считывающую с экрана как знаковые (int), так и беззнаковые (unsigned int) числа?<br>*А выводящую на экран?*<br>Можно. См. функцию getnum из модуля ioproc.asm. |
| 65. | Можно ли на Си или Ассемблере написать универсальную функцию, выводящую на экран как знаковые (int), так и беззнаковые (unsigned int) числа?<br>*А считывающую с экрана?*<br>Нельзя. В каком виде выводить, например, число FF FD, как -3 или как 65533? |
| 68. | Для чего нужны разделяемые библиотеки?<br>Динамические подключаемые (разделяемые) библиотеки – Dynamic-Link Library (.dll-файлы) в Windows и Shared Object (.so-файлы) в Unix/Linux – представляют собой набор скомпилированных функций, некоторые из которых экспортируются наружу и вызываются обычно из исполняемых файлов (.exe) или других библиотек. Т.е., по сути, являются частью программ, исполняемыми файлами, только без функции main(). DLL (аналогично .so) выполняют следующие функции:<br>1) Как и указывает их название, **разделение кода** между различными процессами. Если .dll/.so будет использоваться одновременно несколькими программами, то в память она загрузится только один раз и будет спроецирована во все использующие её процессы. (Однако не полностью: к примеру, секция данных (глобальных переменных) будет у каждого процесса своя.) Т.е. избежание дублирования кода в оперативной памяти. Кроме того, экономия места и на диске тоже.<br>Например, функция *CreateWindow*, которая используется во всех оконных приложениях, реализована в библиотеке *user32.dll*, и программистам не надо вручную рисовать окна, а достаточно вызвать эту функцию с нужными параметрами.<br>2) Опять же, как отражено в названии, возможность **динамически** (во время выполнения программы) загрузить ту или иную библиотеку по выбору (***явное*** связывание – функции *LoadLibrary* / *dlopen*), найти в ней адрес функции по её имени (*GetProcAddress* / *dlsym*) и вызвать её. Иначе, в случае ***неявного*** связывания, библиотеки загружаются автоматически при инициализации процесса на основе таблицы импорта .exe-файла (или кто там эту библиотеку вызывает).<br>3) Реализация **модульности** программы на бинарном уровне. Если будет найдена ошибка в .dll/.so, то достаточно обновить лишь эту библиотеку, а все приложения, её использующие, даже не придётся перекомпилировать (если, конечно, не изменился интерфейс функций библиотеки). |

| | |
|---|---|
| | 4) На технологии разделяемых библиотек практически полностью построены современные ОС Windows и Linux (**API режима пользователя**). Большая часть сервисов, предоставляемых ОС, реализованы как функции, находящиеся в системных библиотеках, например, *CreateWindow* (*user32.dll*), *WriteFile*, *ExitProcess* (*kernel32.dll*) и *read*, *fork*, *printf* (*libc.so*).<br>5) Возможность подмены стандартной функции своей реализацией (**hack**).<br>6) Особые случаи, когда программа не может быть реализована в виде исполняемого файла, например, ловушки оконных сообщений.<br>P.S. Для сравнения, код статических библиотек (.lib в Windows и .a в Linux) включается прямо в файл исполняемой программы во время компиляции (на этапе линковки), но не целиком, а только те функции, которые реально используются. |
| 69. | * Почему в одном процессе не могут соседствовать 16-битные и 32-битные модули (или 32-битные и 64-битные)? Например, 32-битный .exe не может использовать 16-битную .dll. |
| 70. | Можно ли файл так открыть, чтобы читать из него из одной позиции, а писать в другую? |
| 73. | Какие поразрядные операции Вы знаете? |
| 75. | * Можно ли выполнить какой-либо код до main'a?<br>В C++ можно: например, при инициализации глобальных объектов вызываются их конструкторы, а также можно инициализировать глобальные переменные результатом вызова функции.<br>В чистом Си в gcc можно определить функцию с атрибутом constructor:<br>`void __attribute__ ((constructor)) func(void) {`<br>`  puts("premain");`<br>`}`<br>Но это нестандартное GNU-расширение, только для GCC.<br>Вообще, до main'а выполняется много кода, так называемый ***startup code***, который начинается с EntryPoint процесса (функция *_start* в GCC, *mainCRTStartup* (или *WinMainCRTStartup*) в Visual C++ и *__acrtused* в C++ Builder), инициализирует кучу, получает и обрабатывает аргументы командной строки и переменные окружения, настраивает сопроцессор и режим I/O по умолчанию (текстовый) и т.д., вызывает main и затем exit, чаще всего следующим образом:<br>`exit(main(argc, argv, envp));`<br>И, наконец, в Windows можно выполнить свой код даже до EntryPoint своего процесса. Все .dll, от которых программа зависит, загружаются в память процесса, так же, как и сам .exe-модуль, и для каждой .dll выполняется её функция *DllMain*, которая её инициализирует, **до** того, как управление передастся на EntryPoint .exe-модуля. Поэтому можно просто написать свою .dll-библиотеку и подключить её к своему .exe. |
| 76. | В какой памяти находится программа, в то время, когда её исполняет процессор? |
| 78. | Что такое исходные и бинарные (исполняемые) модули, в чём их разница?<br>Может ли модуль-библиотека DLL быть собрана из нескольких исходных модулей? |
| 79. | Можно ли, имея только исполняемый файл, определить, из скольких исходных модулей он был собран? |
| 81. | Почему при передаче через параметры функции<br>массив передаётся по указателю, а структура – по значению? |
| 91. | Дана последовательность байт:<br>00 00 A0 40<br>Какое вещественное число лежит в этих ячейках памяти?<br>В архитектуре x86 используется порядок байт little-endian, т.е. обратный порядок, поэтому после чтения данных из памяти в регистр процессора получится:<br>$40A00000_{16}$<br>Т.к. данные занимают 4 байта и сказано, что это вещественное число, то это тип float, т.е. короткое представление. Для него один бит (старший) отводится на знак числа, 8 бит – на порядок (который кодируется в смещённом виде, где 01…1 = 0; 10…0 = 1; 10…1 = 2 и т.д.), остальные 23 бита на мантиссу. Запишем данное число в двоичной системе счисления:<br>$0\|100\ 0000\ 1\|010\ 0000\ 0000\ 0000\ 0000\ 0000_2$<br>знак порядок мантисса<br>Получаем $+1.01E10_2 = 101_2 = \mathbf{5}_{10}$ |
| 94. | Записать число -3.5 в коротком, длинном и расширенном представлениях |

| | |
|---|---|
| | (в 16-чном виде).<br>Сначала переведём в двоичную систему счисления:<br>$3.5_{10} = 11.1_2$<br>Затем в нормализованное представление (сдвинем точку к первой 1):<br>$11.1_2 = 1.11E1_2$<br>Вспомним, что знак всегда определяется старшим битом, порядок кодируется в смещённом виде, где 01…1 = 0; 10…0 = 1; 10…1 = 2 и т.д., а мантисса записывается без старшего разряда, т.е. без “1.” (кроме расширенного представления).<br>• Короткое (float, 4 байта): порядок 8 бит, мантисса 23 бита:<br>$1\|100\ 0000\ 0\|110\ 0000\ 0000\ 0000\ 0000\ 0000_2 = C0600000_{16}$<br>знак порядок мантисса<br>• Длинное (double, 8 байт): порядок 11 бит, мантисса 52 бита:<br>$1\|100\ 0000\ 0000\|\ 1100\ 0000\ \ldots\ 0000_2 = C00C000000000000_{16}$<br>знак порядок мантисса<br>• Расширенное (long double, 10 байт): порядок 15 бит, мантисса 64 бита:<br>$1\|100\ 0000\ 0000\ 0000\|\ \mathbf{1}110\ 0000\ \ldots\ 0000_2 = C000E000000000000000_{16}$ |
| | Что такое указатель? |
| | Что такое файл? |
| | Для чего нужны сегментные регистры? |
| | Чем различаются объявления int func(void) и int func()? |