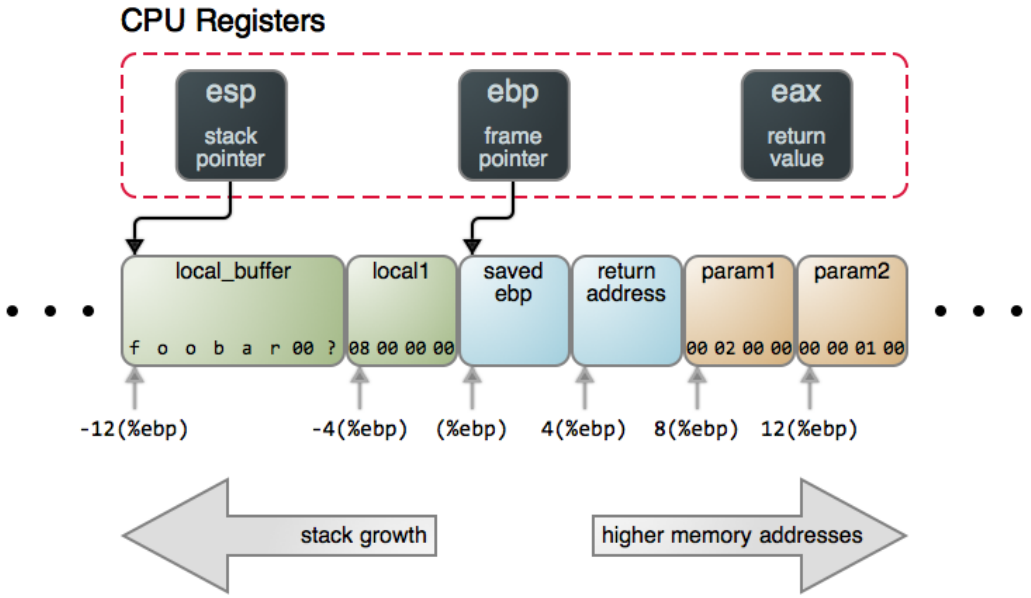
25 августа 2014 в 10:41

Путешествие по Стеку. Часть 1

перевод recovery mode

Блог компании Smart-Soft, C*, Программирование*



В предыдущих материалах мы рассмотрели размещение программы в памяти – одну из центральных концепций, касающихся выполнения программ на компьютерах. Теперь обратимся к *стеку вызовов* – рабочей лошадке большинства языков программирования и виртуальных машин. Нас ожидает знакомство с удивительными вещами вроде функций-замыканий, переполнений буфера и рекурсии. Однако всему свое время – в начале нужно составить базовое представление о том, как работает стек.

Стек имеет такое важное значение, потому что благодаря ему *любая функция* «знает» куда возвращать управление после завершения; функция же, в свою очередь — это базовый строительный блок программы. Вообще, программы внутренне устроены довольно просто. Программа состоит из функций, функции могут вызывать другие функции, в процессе своей работы любая функция помещает данные в стек и снимает их оттуда. Если нужно, чтобы данные продолжили существовать после завершения функции, то место под них выделяется не в стеке, а в куче. Вышесказанное в равной степени относится как к программам, написанным на относительно низкоуровневом C, так и к интерпретируемым языкам вроде JavaScript и C#. Знание данных вещей обязательно пригодится — и если придется отлаживать программу, и если доведется заниматься тонкой подстройкой производительности, да и просто для того, чтобы понимать, что же там, все-таки творится внутри программы.

Итак, начнем. Как только мы вызываем функцию, в стеке для нее создается **стековый кадр**. Стековый кадр содержит *локальные переменные*, а также *аргументы*, которые были переданы вызывающей функцией. Помимо этого кадр содержит служебную информацию, которая используется вызванной функцией, чтобы в нужный момент возвратить управление вызвавшей функции. Точное содержание стека и схема его размещения в памяти могут быть разными в зависимости от процессорной архитектуры и используемой конвенции вызова. В данной статье мы рассматриваем стек на архитектуре x86 с конвенцией вызова, принятой в языке C (**cdecl**). На рисунке вверху изображен стековый кадр, разместившийся у верхушки стека.

Сразу бросаются в глаза три процессорных регистра. *Указатель стека*, **esp**, предназначается для того, чтобы указывать на верхушку стека. Вплотную к верхуше всегда находится объект, который *был добавлен в стек, но еще оттуда не снят*. Точно также в реальной жизни обстоят дела со стопкой тарелок или пачкой 100-долларовых банкнот.

Хранимый в регистре **esp** адрес изменяется по мере того, как объекты добавляются и снимаются со стека, однако он всегда указывает на последний добавленный и еще не снятый со стека объект. Многие процессорные инструкции изменяют значение регистра **esp** как побочный результат своего выполнения. Реализовать работу со стеком без регистра **esp** было бы проблематично.

В случае с процессорами Intel, ровно как и со многими другими архитетурами, стек растет в направлении *меньших*

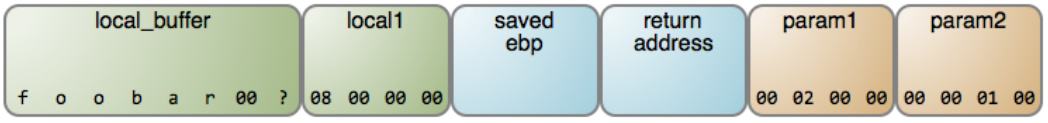
адресов памяти. Поэтому верхушка, в данном случае, соответствует наименьшему адресу в стеке, по которому хранятся валидные используемые данные: в нашем случае это переменная **local_buffer**. Думаю, должно быть понятно, что означает стрелка от **esp** к **local_buffer**. Здесь все, как говорится, по делу – стрелка указывает *точно* на *первый байт*, занимаемый **local_buffer**, и это соответствует тому адресу, который хранится в регистре **esp**.

Далее на очереди еще один регистр, используемый для отслеживания позиций в стеке – регистр **ebp** – *базовый указатель* или *указатель базы стекового кадра*. Данный регистр предназначен для того, чтобы указывать на позицию в стековом кадре. Благодаря регистру **ebp** *текущая функция* всегда имеет своего рода точку отсчёта для доступа к аргументам и локальным переменным. Хранимый в регистре адрес изменяется, когда функция начинает или прекращает выполнение. Мы можем довольно просто адресовать любой объект в стековом кадре как смещение относительно **ebp**, что и показано на рисунке.

В отличии от **esp**, манипуляции с регистром **ebp** осуществляется в основном самой программой, а не процессором. Иногда можно добиться выигрыша в производительности просто отказавшись от страндартного использования регистра **ebp** – за это отвечают некоторые флаги компилятора. Ядро Linux – пример того, где используется такой прием.

Наконец, регистр **eax** традиционно используется для хранения данных, возвращаемых вызвавшей функции — это высказывание справедливо для большинства поддерживаемых в языке C типов.

Теперь давайте разберем данные, содержащиеся в стековом кадре. Рисунок показывает точное побайтовое содержимое кадра, с направлением роста адресов слево-направо – это то, что мы обычно видим в отладчике. А вот и сам рисунок:



Локальная переменная **local_buffer** – это массив байт, представляющий собой нуль-терминированную ASCII-строку; такие строки — неизменный атрибут всех программ на C. Размер строки — 7 байт, и, скорее всего, она была получена в результате клавиатурного ввода или чтения из файла. В нашем массиве может храниться 8 байт и, следовательно, один байт остается неиспользуемым. *Значение этого байта неизвестно*. Дело в том, что, данные то и дело добавляются и снимаются со стека, и в этом «бесконечном танце операции добавления и снятия» никогда нельзя знать заранее, что содержит память, *пока не осуществишь в нее запись*. Компилятор языка C не обременяет себя тем, чтобы инициализировать стековый кадр нулями. Поэтому содержащиеся там данные заранее неизвестны и являются в некоторой степени случайными. Уж сколько крови попило такое поведение компилятора у программистов!

Идем далее. **local1** – 4-байтовое целое число, и на рисунке видно содержимое каждого байта. Кажется, что это большое число – только взгляните на все эти нули после восьмерки, однако здесь наша интуиция сослужила нам дурную службу.

Процессоры Intel используют *прямой порядок байтов* (дословно «остроконечный»), и это значит, что числа хранятся в памяти *начиная с младшего байта*. Иными словами, самый младший значащий байт хранится в ячейке памяти с наименьшим адресом. На рисунках и схемах байты многобайтовых чисел традиционно изображаются в порядке слева-направо. В случае с прямым порядком байт, самый младший значащий байт будет изображен в крайней левой позиции, что отличается от привычного нам способа представления и записи чисел.

Неплохо знать о том, что вся эта «остроконечная / тупоконечная» терминология восходит к произведению Джонатана Свифта «Путешествия Гулливера». Подобно тому, как жители Лилипутии чистили яйцо с острого конца, процессоры Intel тоже обрабатывают числа начиная с младшего байта.

Таким образом, переменная **local1** в действительности хранит число 8 (да-да, прям как количество щупалец у осьминога). Что касается **param1**, то там во втором от начала октете изображена двойка, поэтому в результате получаем число 2 * 256 = 512 (мы умножаем на 256, потому что каждый октет – это диапазон от 0 до 255). **param2** хранит число 1 * 256 * 256 = 65536.

Служебная информация стекового кадра включает в себя два компонента: адрес стекового кадра *вызвавшей функции* (на рисунке — saved ebp) и адрес инструкции, куда необходимо передать управление по завершении данной функции (на рисунке – return address). Эта информация делает возможным возвращение управления, и следовательно, дальнейшее выполнение программы как будто никакого вызова и не было.

Теперь давайте рассмотрим процесс «рождения» стекового кадра. Стек растет *не в том направлении, которое обычно ожидают увидеть*, и сначала это может сбивать с толку. Например, чтобы увеличить стек на 8 байт, программист *вычитает* 8 из значения, хранимого в регистре **esp**. Вычитание – странный способ что-либо увеличить. Забавно, не правда ли!

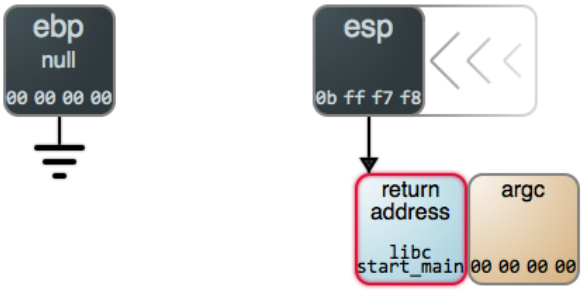
Возьмем для примера простенькую программу на C:

Simple Add Program - add.c

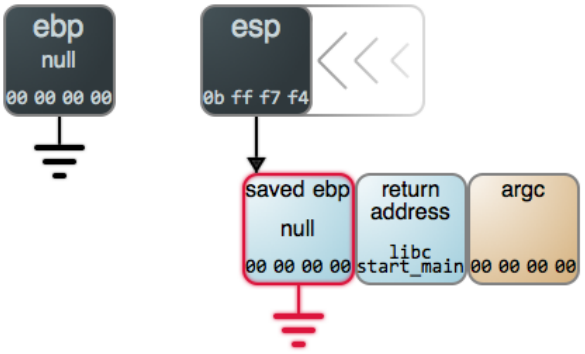
```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Предположим, программу запустили без параметров в командной строке. При выполнении «сишной» программы в Linux, первым делом управление получает код, содержащийся в стандартной библиотеке C. Этот код вызовет функцию **main()** нашей программы, и, в данном случае, переменная **argc** будет равна 0 (*на самом деле, переменная будет равна «1», что соответствует параметру — названию, под которым запущена программа, но давайте для простоты это момент сейчас опустим*). При вызове функции **main()** происходит следующее:

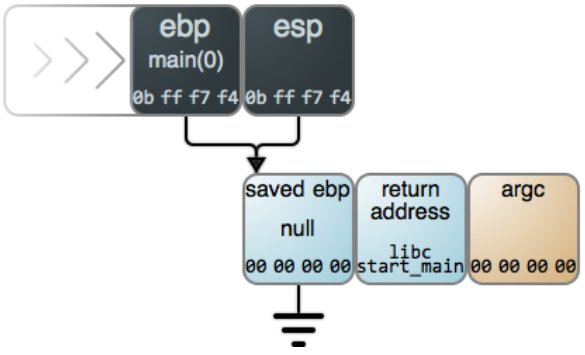
1. `call main # помещаем адрес возврата в стек, передаем управление по метке`



2. `pushl %ebp # сохраняем текущее значение регистра ebp в стек`



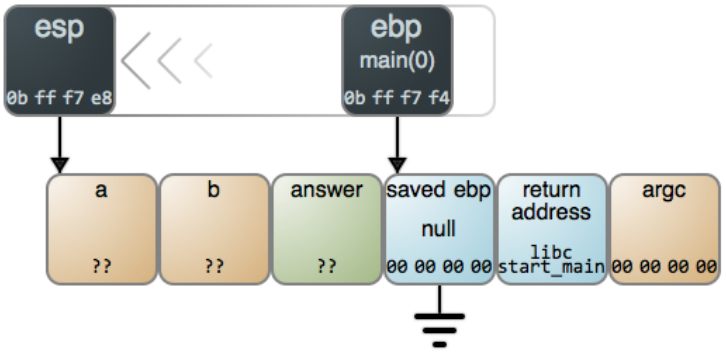
3. `movl %esp, %ebp # копируем значение регистра esp в регистр ebp`



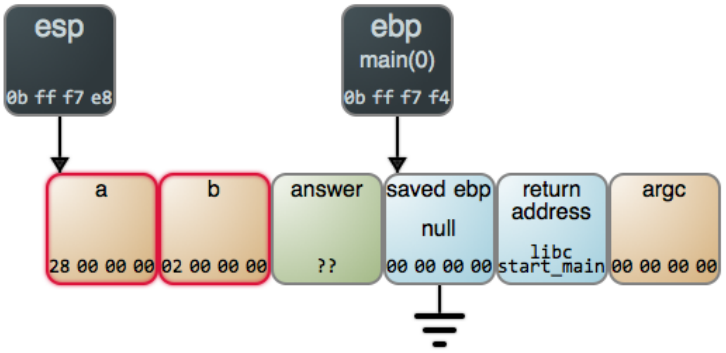
Шаг 2 и 3, а также 4 (описан ниже) соответствуют последовательности инструкций, которая называется **«прологом»** и встречается практически в любой функции: текущее значение регистра **ebp** помещается в стек, затем значение регистра **esp** копируется в регистр **ebp**, что фактически приводит к созданию нового стекового кадра. Пролог функции **main()** такой же, как и других функций, с той лишь разницей, что при начале выполнения программы регистр **ebp** содержит нули.

Если взглянуть на то, что располагается в стеке под **argc**, то будут видны еще некоторые данные – указатель на строку-название, под которым программа была запущена, указатели на строки-параметры, переданные через командную строку (традиционный C-массив **argv**), а также указатели на переменные среды и непосредственно сами эти переменные. Однако, на данном этапе нам это не особо важно, так что продолжаем двигаться по направлению к вызову функции **add()**:

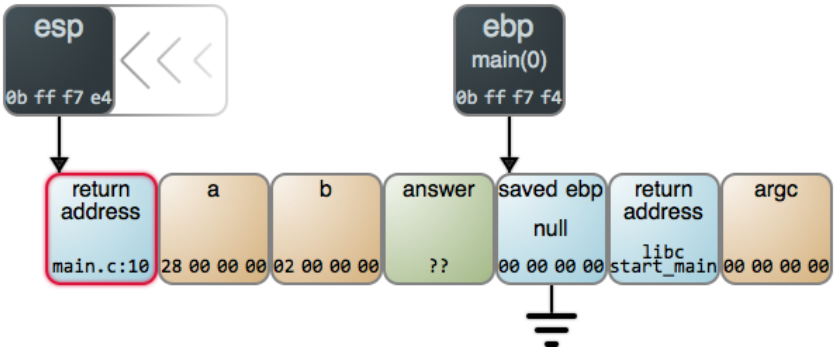
```
4.  subl $12, %esp  # выделяем место под данные
```




```
5.  movl $2, 4(%esp)  # присваиваем b значение 2
    movl $40, (%esp)  # присваиваем a значение 40
```

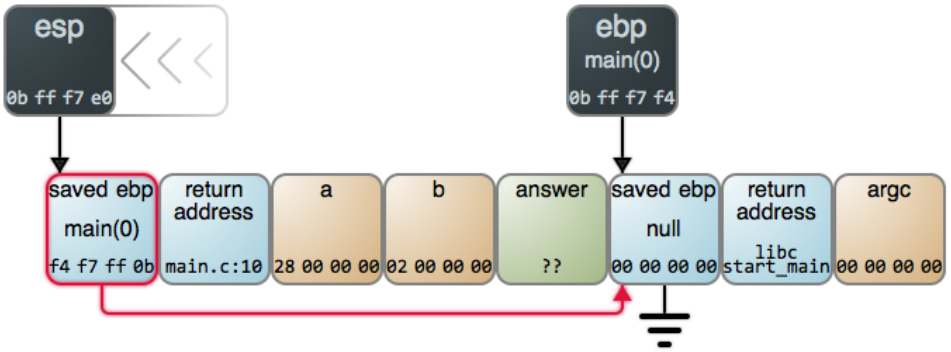



```
6.  call add # помещаем адрес возврата в стек, передаем управление по метке add
```

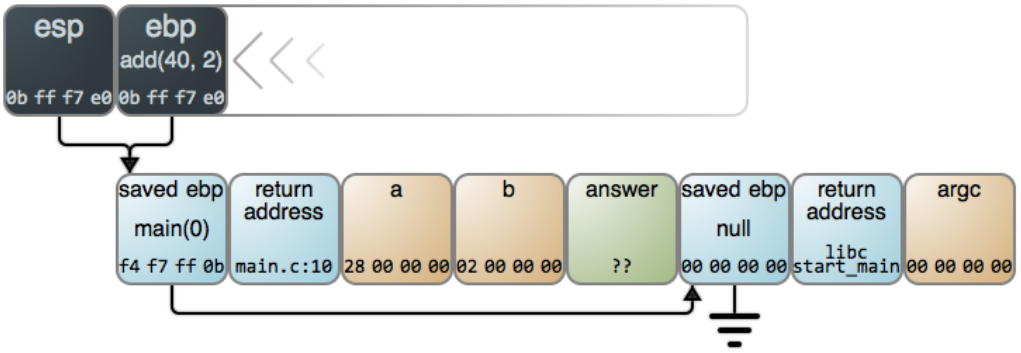


Функция **main()** сначала вычетает 12 из текущего значения в регистре **esp** для выделения нужного ей места и затем присваивает значения переменным **a** и **b**. Значения, хранимые в памяти, изображены на рисунке в шестнадцатеричной форме и с прямым порядком байтов – как и в любом отладчике. После присвоения значений, функция **main()** вызывает функцию **add()**, и та начинает выполняться:

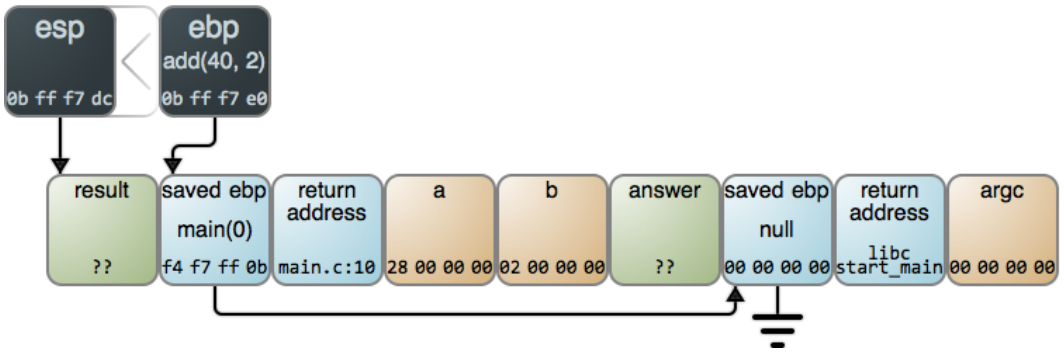
```
7.  pushl %ebp  # сохраняем текущее значение регистра ebp в стек
```




```
8.  movl %esp, %ebp  # копируем значение регистра esp в регистр ebp
```




```
9.  subl $4, %esp  # выделяем место под переменную result
```

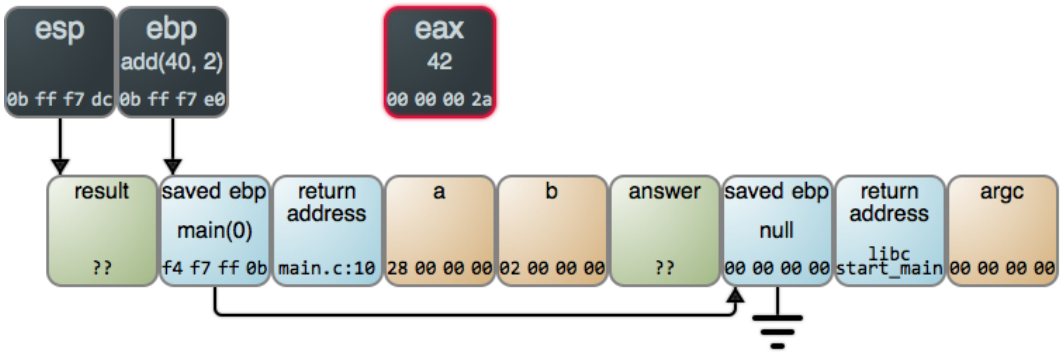


Чем дальше, тем интересней! Перед нами еще один пролог, однако теперь уже четко видно как последовательность стековых кадров в стеке оказывается организованной в связный список, и регистр **ebp** хранит ссылку на первый элемент этого списка. Вот с опорой на это и реализованы трассировка стека в отладчиках и Exception-объекты высокоуровневых языков. Обратим внимание на типичную для начала выполнения функции ситуацию, когда регистры **ebp** и **esp** указывают в одно и то же место. И еще раз вспомним, что для наращивания стека осуществляется вычитание из значения, хранящегося в регистре **esp**.

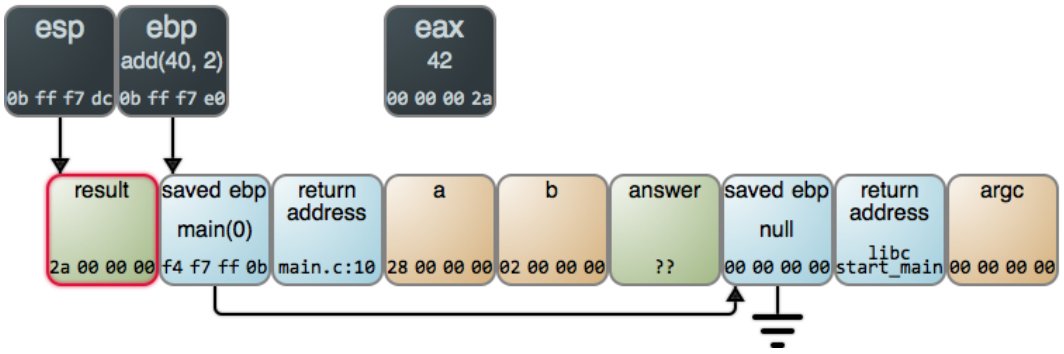
Важно заметить следующее — при копировании данных из регистра **ebp** в память происходит непонятное на первый взгляд изменение порядка хранения байтов. Дело в том, что для регистров такого понятия как «порядок байтов» не существует. Иными словами, рассматривая регистр, мы не можем говорить о том, что в нем есть «старшие или младшие адреса». Поэтому отладчики показывают значения, хранимые в регистрах, в наиболее удобном для человеческого восприятия виде: от более значимых к менее значимым цифрам. Таким образом, имея стандартную нотацию «слева-направо» и «little-endian» машину, создается обманчивое впечатление, что в результате операции копирования из регистра в память байты поменяли порядок на обратный. Я хотел, чтобы картина, показанная на рисунках была максимально приближена к реальности – отсюда и такие рисунки.

Теперь, когда самая сложная часть у нас позади, осуществляем сложение:

```
    movl 12(%ebp), %eax  # помещаем значение из b в регистр eax
10. movl 8(%ebp), %edx   # помещаем значение из a в регистр edx
    addl %edx, %eax # складываем значения регистров edx и eax, помещаем результат 42 в eax
```




```
11. movl %eax, -4(%ebp) # помещаем содержимое регистра eax в result
```



Здесь у нас появляется неизвестный регистр, чтобы помочь со сложением, но в целом ничего особенного или удивительного. Функция **add()** выполняет вою работу и, начиная с этого момента все действия в стеке будут осуществляться в обратном порядке. Но об этом расскажем как-нибудь в другой раз.

Все, кто дочитал до этих строк, заслуживает подарок за стойкость, поэтому я с огромной гиковской гордостью презентую Вам вот эту единую схемку, на которой изображены все вышеописанные шаги.

Не так уж все и сложно, стоит только разложить все по полочкам. Кстати, маленькие квадратики *очень сильно* помогают понимаю. Без «маленьких квадратиков» в информатике вообще никуда. Надеюсь, мои рисунки позволили составить ясную картину происходящего, на которой интуитивно просто показан и рост стека, и изменения содержимого памяти. При ближайшем рассмотрении, наше программное обеспечение не так уж и сильно отличается от простой машины Тьюринга.

На этом завершается первая часть нашего путешествия по стеку. В будущих статьях нас ждут новые погружения в «байтовые» дебри, после чего посмотрим, что же на этом фундаменте способны выстроить высокоуровневые языки программирования. Увидимся на следующей неделе.

Материал подготовлен сотрудниками компании Smart-Soft
smart-soft.ru

function, stack, stack frame, linux, x86, cdecl, program

Комментарии (15)

Keroro 25 августа 2014 в 12:28 # +4

Так вот они какие, заземлённые указатели…

khim 25 августа 2014 в 13:23 # 0