сишные трюки
(1Ah выпуск)

крис касперски ака мыщъх, a.k.a. souriz, a.k.a. nezumi, no-email

си – самый низкоуровневый из всех высокоуровневых языков, идеологически близкий к ассемблеру. настолько близкий, что способный позаимствовать многие чисто ассемблерные фичи, существенно упрощающие решение многих задач и реализуемые без использования ассемблерных вставок, то есть средствами самого языка.

#1 – реверс строки в адресном континууме

Перевернуть строку, не используя дополнительной памяти, — довольно распространенное задание для юниоров, нацеленное на знание указателей и умение работать с ними. Когда же такое задание дают матерым программистам, над "экзаменатором" не грех и постебаться, воскликнув: "а зачем ее разворачивать? она же ведь _уже_ развернута!" после чего пояснить: "в мире все относительно: где конец того начала, что есть начало конца?!" x86 процессоры (как и многие другие) поддерживают флаг направления: просто взводим его, перемещаем указатель на конец строки и движемся в обратном направлении. Некоторые проблемы создает отсутствие завершающего нуля на конце (точнее в начале) строки, но что мешает нам запомнить ее длину? Идея в том, что в адресном пространстве нет понятия "верха" и "низа". До сих пор не утихают споры: куда растет стек и адреса памяти. А потому, всякая последовательность байт _одновременно_ существует в двух состояниях — прямом и развернутом.

Увы, при всей красоте этой концепции библиотечным функциям ее не объяснишь. В частности, fopen, printf, MessageBox и еще куча остальных всегда движутся от младших адресов к старшим и без полноценного реверса тут никак не обойтись, однако, в своих собственных функциях мы можем воспользоваться этим "подарком" относительности, передавая им в качестве аргумента флаг в каком направлении следует двигаться — увеличивать указатель или уменьшать его?

Кто-то может презрительно хмыкнуть: флаг направления и в чистом ассемблере редко используется, а уж на языках высокого уровня ему и вовсе не место, ну разве, что в академических задачках, не имеющих никакого практического применения, однако, это не так. Достаточно часто массив элементов, отсортированный по возрастанию, требуется превратить в массив, отсортированный по убыванию и реверс элементов с перемещением их по памяти – не самая быстрая операция, особенно если этих элементов у нас много, а переупорядочивать их приходится достаточно часто. Флаг направления в этом случае экономит кучу процессорных тактов, ценой незначительного усложнения алгоритма.

Единственный недостаток предложенного способа — падение производительности при движении взад, т. е. от старших адресов к младших. Так уж повелось, что вся подсистема памяти от кэш-контроллера первого уровня до DRAM-модулей ориентирована на чтение вперед, иначе начинает тормозить, облагая нас "штрафными" тактами. А потому, многократное чтение массива в обратном направлении (особенно большого массива) не есть Zen-way и лучше один раз затратить время на однократный "честный" реверс, а потом читать вперед сколько влезет. С другой стороны, реверс + однократное чтение массива в прямом направлении, _намного_ медленнее однократного чтения того же массива в обратном направлении.

#2 – принудительная проверка успешности операции

Ассеблерщикам хорошо! В их распоряжении есть флаги процессора, сплошь и рядом используемые для индикации ошибок выполнения функций (обычно, за это отвечает флаг переноса, реже — флаг нуля). А вот на Си… если функция возвращает int (а возвращает она его предательски часто), то в качестве индикатора успешности выполнения операции приходиится использовать значение, не входящее в диапазон "валидных" ответов. Применительно к malloc, — это ноль (нулевой указатель не может быть валидным, во всяком случае в Си), если же ноль входит в область допустимых значений, приходится возвращать -1 или выкручиваться как-то еще.

Как следствие, мы имеем полный разброд без всяких признаков стандартизации. Попробуй удержи в голове все эти подробности! Неудивительно, что многие программисты "забывают" о проверке, передавая возвращенные (некорректные) значения другой функции, в результате чего программа падает (в лучшем случае!), а в худшем – ведет себя некорректно, но даже если и падает, то совсем не в месте ошибки, а довольно далеко от него. Положение осложняется тем, что многие проекты пишутся кучей людей, среди которых попадаются откровенные "вредители", не выполняющие никаких проверок вообще, а "аукается" это в чужих модулях, высаживая коллег на измену.

Как _гарантированно_ заставить "пионеров" выполнять проверки или хотя бы добиться того, чтобы программа стабильно грохалась именно в том месте, где возникает ошибка?! Да очень просто — достаточно вместо значения возвращать _указатель_ на память, где это значение лежит или ноль (при ошибке). Обращение к нулевому указателю приводит к немедленному выбросу исключения, за которое "пионеру" легко надавать по ушам.

Конечно, при этом возникает "лишняя" операция обращения к памяти, но это не проблема, поскольку, в общем зачете накладные расходы стремятся к нулю. Даже если функция целиком состоит из одного return и принимает параметры через регистры по fastcall-соглашению, она все равно заталкивает адрес возврата на стек, обращаясь к памяти… 50% "оверхид" на пустой функции — не такой уж плохой результат. Настоящая проблема в том, что найти место для размещения возвращаемых данных не так-то просто! Если выделять блок при помощи malloc, то это, во-первых, _слишком_ медленно, а во-вторых, если "пионер" забудет освободить возращенный указатель (а он забудет), память потечет рекой.

А что если возвращать указатель на статическую переменную? Это снимает проблемы с освобождением памяти, но функция становится нереентерабельной. В некоторых случаях это обходится использованием локальной памяти потока, но локальная память потока бессильна против рекурсивных вызовов функции, однако, рекурсия встречается не так уж и часто, поэтому, данный способ имеет право на существование.

Точно так же, если функция возвращает данные по указателю, мы можем "навязать" проверку успешности выполнения операции путем возвращения указателя на указатель, возвращая в случае ошибки ноль.

#3 – имитация INT

Во времена MS-DOS большинство системных функций вызывалось путем генерации программного прерывания командой INT 21h, а UNIX системы используют этот путь и сегодня (только вместо вектора 21h у них 80h). Достоинство такого подхода в том, что код, вызывающий INT, не имеет ни малейшего представления о том, по какому адресу находится системный обработчик, более того, этот адрес может динамически меняться (например, в MS-DOS появился новый резидентный вирус, хе-хе).

Windows NT вплоть до XP так же использовала INT в качестве "моста" между user-land и kernel-land, позволяя прикладному коду делать системные вызовы, но, начиная с XP, медленная команда INT сменилась более быстрой SYSENTER/SYSCALL (Intel/AMD соответственно), однако, на прикладном уровне основным средством межмодульных вызовов стал экспорт/импорт эффективных адресов. Именно так и работают динамические библиотеки.

Экспорт/импорт прекрасно работает в рамках одной программы, но когда мы пытаемся прикрутить к ней поддержку plug-in'ов, возникает куча проблем. Фактически основная программа с точки зрения plag-in'а превращается в операционную систему и необходимо как-то передать адреса всех функций, чтобы plug-in их мог вызывать. Обычно для этого используется готовый механизм и plug-in'ы реализуется как динамические библиотеки, что накладывает на разработчика программы множество ограничений, призванных обеспечить обратную совместимость. Но это еще что! Отсутствует возможность (легальная) написания plug-in'ов-фильтров, встраивающихся между уже загруженным plug-in'ом и основной программой.

Вот тут бы как нельзя кстати оказался INT, но, во-первых, это системно-зависимо и абсолютно непереносимо, а, во-вторых, вызывать INT с прикладного уровня для передачи управления на прикладной уровень — негуманно. Вот намного более элегантный способ: основная программа устанавливает обработчик исключений, а plug-in для вызова функций программы производит запись определенной структуры данных по нулевому указателю, что ведет к генерации исключения, перехватываемого (и обрабатываемого) основной программой.

Программа может динамически переназначать обработчики исключения в зависимости от текущего режима работы (например, запуске встроенного редактора), тоже самое могут делать и plug-in'ы. Устанавливая свой обработчик исключения, перекрывающий предыдущий,

они перехватывают общение основной программы со всей цепочкой plug-in'ов-фильтров. Ну разве не красота?!

Пару слов о структуре, записываемой по нулевому указателю. Количество возможных решений намного больше одного (все зависит от вкусов программиста, но общий принцип таков):

```
struct leben
{
        char    magic[]="NZM1";         // магический "пирожок"
        int     syscall_id;             // номер "системного вызова"
        void    *list;                  // указатель на список аргументов "сис. вызова"
}
```

Листинг 1 "магическая" таблица, записываемая по нулевому адресу

magic – хранит магическое слово, проверяемое обработчиком исключения, чтобы удостовериться, что это не случайная операция записи, а преднамеренный "системный вызов", номер которого хранится в syscall_id, впрочем, вместо номера можно использовать имена "системных вызовов" — это уже на усмотрение программиста. Аргументы передаются (и возвращаются) через указатель на область памяти (в данном случае *list) формат которой варьируется от одного "системного вызова" к другому.