сишные трюки
(0xC выпуск)

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

сегодняшний выпуск посвящен проблемам удаленной диагностики ошибок. это когда у пользователя падает программа, а воспроизвести ситуацию у себя на месте у нас на месте — не получается. удаленную отладку (по модему и/или интернету) не предлагать, т.к. далеко не всякий пользователь на это согласиться и все что нам остается — внедрить в программу дополнительный проверочный код.

совет 1 — никогда не удаляйте проверки из release

Большинство программистов, напичкивающие отладочную версию программу всевозможными проверками корректности всех значений, какие только можно проверить, словно лемминги, подчиняющиеся законам всеобщей традиции, удаляют их из финального релиза. А зачем?! Отладочную информацию (генерируемую компилятором) удалять, естественно, нужно (поскольку она не только в разы увеличивает размер исполняемого файла, но, облегчает его взлом, но так же в большинстве случаев вырубает многие опции компиляции).

Удаление избыточных проверок практически не сказывается на размере и слабо влияет на производительность (за исключением, быть может, многочисленных проверок в глубоко вложенных циклах). Так зачем же их удалять?! И каким образом выполнять диагностику, если на машине конечного пользователя программа внезапно откажет в работе?! Если программист предполагал, что ошибка может проявиться в отладочной версии и добавил специальную проверку, то почему он считает, что она заведомо не появится в релизе? Где гарантия, что в процессе отладчики были протестированы все возможные состояния программы? Где гарантия, что мы не имеем дело с "наведенной" ошибкой (зависящий от других частей программы, на первый взгляд, не имеющей к ней никакого отношения)?

Чем больше проверок останется в финальной версии, тем легче будет найти источник ошибки при ее возникновении. Конечно, проверка проверке рознь. Одно дело проверить указатель на нуль и совсем другое корректность форматированной строки данных. В этом случае (если программист озабочен производительностью) можно ввести специальный флаг или ключ командной строки, включающий все "тяжеловесные" проверки.

совет 2 — активно используйте самодиагностику

Самотестирование — великая вещь и все сложные электронные устройства (в том числе и процессоры) обязательно включают компоненты, выполняющие самодиагностику. Тот же самый подход может (и должен) применяться в программном коде. Каждая мало-мальски сложная процедура должна поддерживать функцию самотестирования — подавать на свой вход контрольный набор данных (жестко прописанный в файле) и сравнивать полученный результат с эталоном (так же хранящимся в файле). Обычно, таких наборов бывает несколько (один не обеспечивает полного покрытия всех ветвей процедуры).

На стадии отладки польза самодиагностики очевидна, но вот в финальной версии она зачем?! А затем, что мы не можем доверять ни аппаратуре, ни системным библиотекам, ни самой оси, установленной у пользователя. Личный пример из жизни — машинные команды PUSH reg16 в 32-битном режиме у Intel и AMD реализованы неодинаково. Обе они забрасывают на вершину стека двойное слово (как и положено по спецификации), но одна очищает старшие разряды, а другая оставляет их без изменений (со всем мусором, что в них есть). В мыщъх'иной программе была досадная ошибка, при определенных обстоятельствах приводящая к потере нуля в конце ASCIIZ-строки, но, поскольку, за ней следовало двойное слово, заброшенное на стек командой PUSH reg16, и мыщъх отлаживал программу на процессоре, очищающем старший разряд, то все работало более или нормально (два байта "мусора", появляющихся в конце строки, никому не мешали), но вот при запуске на другом процессоре, где завершающего нуля не оказывалось, возникала критическая ситуация, завершающаяся исключением.

Или вот — незначительные различия в реализации "плавающих" команд на различных процессорах, могут привести к странному поведению программы, которое будет невозможно воспроизвести на любом другом процессоре!!!

Про разгон, дефекты памяти и т.д. мы вообще молчим! Никогда нельзя быть уверенным в том, что после выполнения: a = 6; a =a+3; в переменной a окажется именно 9, а не 83737382. И виноват тут может быть не только процессор, но и "удар по памяти" (это когда совершенно посторонняя функция, обратившись по неинициализированному указателю, не запишет что-то в чужую область данных).

Естественно, самотестирование занимает некоторое время и для достижения максимальной производительности мыщъх использовал его в том случае, если предыдущий запуск программы завершился в аварийном режиме. Плюс, на всякий непредвиденный случай, присутствовал недокументированный флаг, форсирующий самотестирование, даже если предыдущий запуск был завершен нормально.

Функции самотестирования не раз выручали мыщъх'а и помогли ему сэкономить колоссальное количество времени, поскольку ряд ошибок был связан с "особенностями" конкретного оборудования, то есть другими словами, причина лежала вне исходного кода программы.

совет 3 — секреты отладочной печати

Отладочная печать — великолепное изобретение, появившееся еще в те времена, когда интерактивных отладчиков не существовало и в помине, а отлаживать было надо. Большинство программистов используют тривиальную запись в текстовой log-файл или API-функцию OutputDebugString. Первый метод, естественно, лучше, т.к. он не требует наличия отладчика или специальной утилиты для перехвата отладочной печати, которую конечному пользователю придется устанавливать на своей компьютер. Мы же ведь не собираемся исключать отладочную печать из финальной версии, верно? Естественно, не собираемся! Достаточно добавить специальный ключ командой строки, "секретную" комбинацию клавиш или "честную" опцию в настройках программы. Лог лучше всего вести в текстовой форме. Так и пользователю будет проще его пересылать нам по почте, да и сам пользователь сможет убедиться, что там нет ничего такого, чего бы он не хотел разглашать.

Вот только… при возникновении критической ошибки, система завершает работу приложения еще _до_ того, как будут сброшены дисковые буфера. Даже использование функции fflush ничего не решает (а вот скорость программы замедляет весьма радикально). Как же быть?! Да очень просто — создать в shared-memory кольцевой буфер заданного размера и весь отладочный вывод направлять туда, читая ее с помощью дочернего процесса. Тогда, при аварийном завершении материнского процесса, shared-memory _не_ будет освобождена системой и дочерний процесс успеет принять последнее отладочное сообщение, отправленное упавшей программой. К тому же, этот метод работает намного быстрее прямой записи на диск.

А почему буфер должен быть именно кольцевым?! Ну, вообще-то, это не требование, а скорее так, простое пожелание (обычно нас интересует не весь отладочный вывод целиком, а события непосредственно предшествующие падению), но при интенсивном отладочном выводе полный размер лога может достигать десятков мегабайт, большая часть из которых не несет никакой полезной нагрузки, так что лучше заранее исключить ее, замкнув буфер в кольцо.

совет 4 — автоматический трассировщик это просто

В самых ответственных случаях программу, поставляемую заказчику, имеет смысл снабдить простейшим автоматическим трассировщиком, на создание которого уйдет не больше одного вечера. Просто взводим флаг трассировки (TF) и отлавливаем отладочные исключения штатными средствами операционной системы (через SEH), записывая: а) адрес машинной команды; б) содержимое регистров; в) адрес ячейки памяти к которой она обращается.

Когда трассировка из прикладного уровня дойдет до ядра, процессор самостоятельно "опустит" флаг трассировки на время прохождения нулевого кольца, и потом "поднимет" его при возвращении на прикладной уровень, так что предусматривать специальную обработку для исключения системных вызов из списка трассируемых функций — не надо.

Естественно, трассировка на несколько порядков (!) замедляет скорость работы программы и потому должна включаться специальной комбинацией клавиш, которую пользователь нажимает в тот момент, когда программа приближается к месту сбоя на максимально близкое расстояние, а для этого, пользователю придется _воспроизвести_ ситуацию при которой возникает ошибка. Если же ему это сделать не удастся, что ж! Включаем трассировщик при старте программы специальным ключом командой строки и пишет результат трассировки в кольцевой буфер, который внимательно изучаем.

Располагая информацией о ходе выполнения программы, содержимом регистров и ячеек памяти, мы сможем любую ошибку, какой бы заковыристой она ни была (ведь фактически мы отлаживаем программу на клиентской стороне в не интерактивном режиме). При желании (если жаба душит) трассировщик можно реализовать в виде отдельной динамической библиотеки, высылаемой клиенту только при возникновении серьезных проблем. Тоже самое, кстати, относится к функциям самодиагностики.

Понятное дело, программа, защищенная протекторами, содержащими анти-отладочные приемы с автоматическим трассировщиком работать не будет, так что приходится отказывается либо от протекторов, либо от трассировки. Либо же писать "умный" трассировщик, обходящий анти-отладочные приемы, но это уже серьезная задача, решение которой может затянуться не на одну неделю.