Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №1

Курс: «Системное программирование»

Тема: «Обработка исключений в ОС Windows»

Выполнил студент:

Бояркин Никита Сергеевич
Группа: 13541/3

Проверил:

Душутина Елена Владимировна

Санкт-Петербург
2017 г.

Содержание

Лабораторная работа №1

1.1 Цель работы

Научиться обрабатывать исключения с помощью встроенных средств WinAPI. Использовать системные утилиты для получения информации о системной активности процесса.

1.2 Программа работы

1. Сгенерировать и обработать исключения с помощью функций WinAPI;
2. Получить код исключения с помощью функции GetExceptionCode.
   - Использовать эту функции в выражении фильтре;
   - Использовать эту функцию в обработчике.
3. Создать собственную функцию-фильтр;
4. Получить информацию об исключении с помощью функции GetExceptionInformation; сгенерировать исключение с помощью функции RaiseException;
5. Использовать функции UnhandleExceptionFilter и Set UnhandleExceptionFilter для необработанных исключений;
6. Обработать вложенные исключения;
7. Выйти из блока __try с помощью оператора goto;
8. Выйти из блока __try с помощью оператора leave;
9. Преобразовать структурное исключение в исключение языка C, используя функцию translator;
10. Использовать финальный обработчик finally;
11. Проверить корректность выхода из блока __try с помощью функции AbnormalTermination в финальном обработчике finally.

1.3 Характеристики системы

Некоторая информация об операционной системе и ресурсах системы:

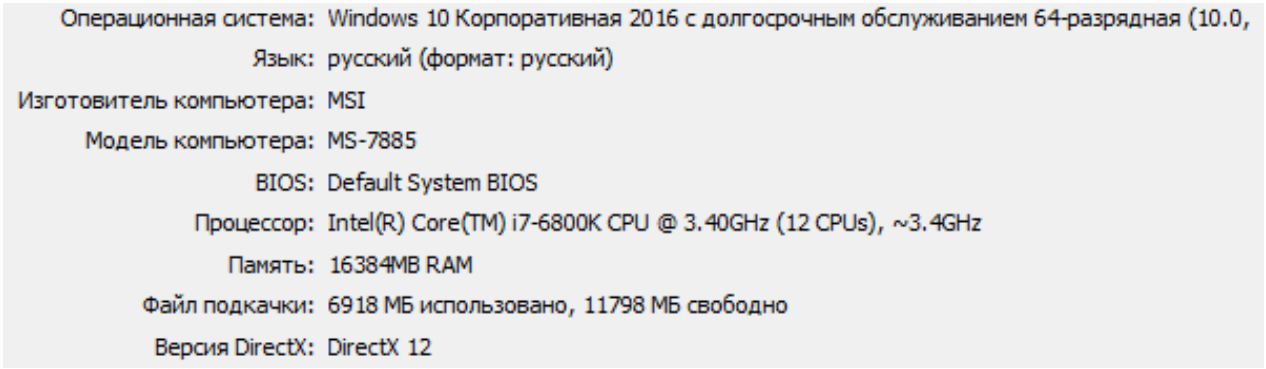

Рис. 1.1: Конфигурация системы

Информация о компиляторе:

```
Оптимизирующий
 компилятор Microsoft (R) C/C++ версии 19.00.24215.1 для x86
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.
```

Информация о компоновщике:

```
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.
```

Версии используемых утилит:

Рис. 1.2: Версия утилиты Auto Debug

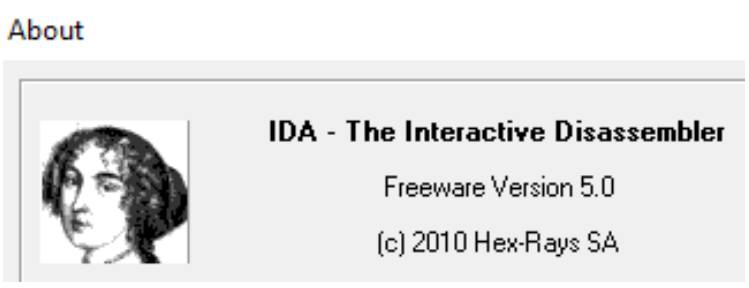

Рис. 1.3: Версия деассемблера Ida

1.4 Ход работы

1.4.1 Генерация и обработка исключения средствами WinAPI

В операционной системе Windows используется механизм структурной обработки исключений (SEH). В отличие от встроенных средств обработки исключений языка C++, SEH позволяет обрабатывать не только программные исключения, но и аппаратные.

```
__try {
  // Защищенный код.
} __except ( /* Фильтр исключений. */ ) {
  // Обработчик исключений.
}
```

Если при выполнении защищенного кода из блока __try возникнет исключение, то операционная система перехватит его и приступит к поиску блока __except. Найдя его, она передаст управление фильтру исключений. Фильтр исключений может получить код исключения и на основе этого кода принять решение, передать управление обработчику или же сказать системе, чтобы она искала предыдущий по вложенности блок __except [1] Фильтр исключений может возвращать одно из трех значений (идентификаторов), которые определены в файле excpt.h:

- Идентификатор EXCEPTION_EXECUTE_HANDLER означает, что для этого блока __try есть обработчик исключения и он готов обработать это исключение.
- Идентификатор EXCEPTION_CONTINUE_SEARCH означает, что для обработки исключения существует предыдущий по вложенности блок __except.
- Идентификатор EXCEPTION_CONTINUE_EXECUTION означает, что выполнение продолжится с того участка кода, который вызвал исключение.

Подобная система обработки исключений позволяет организовывать вложенные исключения, что значительно увеличивает гибкость и читабельность языка программирования.

Некоторые типы исключений, которые могут быть обработаны в фильтре [2]:

- EXCEPTION_ACCESS_VIOLATION – попытка чтения или записи в виртуальную память без соответствующих прав доступа;
- EXCEPTION_BREAKPOINT – встретилась точка останова;
- EXCEPTION_DATATYPE_MISALIGNMENT – доступ к данным, адрес которых не выровнен по границе слова или двойного слова;
- EXCEPTION_SINGLE_STEP – механизм трассировки программы сообщает, что выполнена одна инструкция;
- EXCEPTION_ARRAY_BOUNDS_EXCEEDED – выход за пределы массива, если аппаратное обеспечение поддерживает такую проверку;
- EXCEPTION_FLT_DENORMAL_OPERAND – один из операндов с плавающей точкой является ненормализованным;
- EXCEPTION_FLT_DIVIDE_BY_ZERO – попытка деления на ноль в операции с плавающей точкой;
- EXCEPTION_FLT_INEXACT_RESULT – результат операции с плавающей точкой не может быть точно представлен десятичной дробью;
- EXCEPTION_FLT_INVALID_OPERATION – ошибка в операции с плавающей точкой, для которой не предусмотрены другие коды исключения;
- EXCEPTION_FLT_OVERFLOW – при выполнении операции с плавающей точкой произошло переполнение;
- EXCEPTION_FLT_STACK_CHECK – переполнение или выход за нижнюю границу стека при выполнении операции с плавающей точкой;
- EXCEPTION_FLT_UNDERFLOW – результат операции с плавающей точкой является числом, которое меньше минимально возможного числа с плавающей точкой;

- EXCEPTION_INT_DIVIDE_BY_ZERO – попытка деления на ноль при операции с целыми числами;
- EXCEPTION_INT_OVERFLOW – при выполнении операции с целыми числами произошло переполнение;
- EXCEPTION_PRIV_INSTRUCTION – попытка выполнения привилегированной инструкции процессора, которая недопустима в текущем режиме процессора;
- EXCEPTION_NONCONTINUABLE_EXCEPTION – попытка возобновления исполнения программы после исключения, которое запрещает выполнять такое действие.

Разработаем программу, которая генерирует исключение EXCEPTION_INT_DIVIDE_BY_ZERO из защищенного участка кода и выводит его шестнадцатиричный код в обработчике:

```
#include <iostream>
#include <windows.h>

int main() {
  __try {
    RaiseException(EXCEPTION_INT_DIVIDE_BY_ZERO, NULL, NULL, nullptr);
  } __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    std::cerr << "EXCEPTION_INT_DIVIDE_BY_ZERO: 0x" << std::hex <<
    EXCEPTION_INT_DIVIDE_BY_ZERO << std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_INT_DIVIDE_BY_ZERO: 0xc0000094
```

Попробуем обработать аппаратное исключение EXCEPTION_FLT_OVERFLOW:

```
#include <iostream>
#include <windows.h>

int main() {
  __try {
    RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
  } __except(GetExceptionCode() == EXCEPTION_FLT_OVERFLOW ? EXCEPTION_EXECUTE_HANDLER :
    EXCEPTION_CONTINUE_SEARCH) {
    std::cerr << "EXCEPTION_FLT_OVERFLOW: 0x" << std::hex << EXCEPTION_FLT_OVERFLOW <<
    std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_FLT_OVERFLOW: 0xc0000091
```

Оба исключения были успешно обработаны. В качестве фильтра для обработчика был использован простой тернарный оператор, который вызывает обработчик только при возникновении конкретного исключения. Если исключение не было вызвано, то поиск обработчика продолжится, что для данного случая эквивалентно падению программы.

Для получения более детальной информации о поведении системы при обработке исключений, рассмотрим последовательность произведенных системных вызовов.

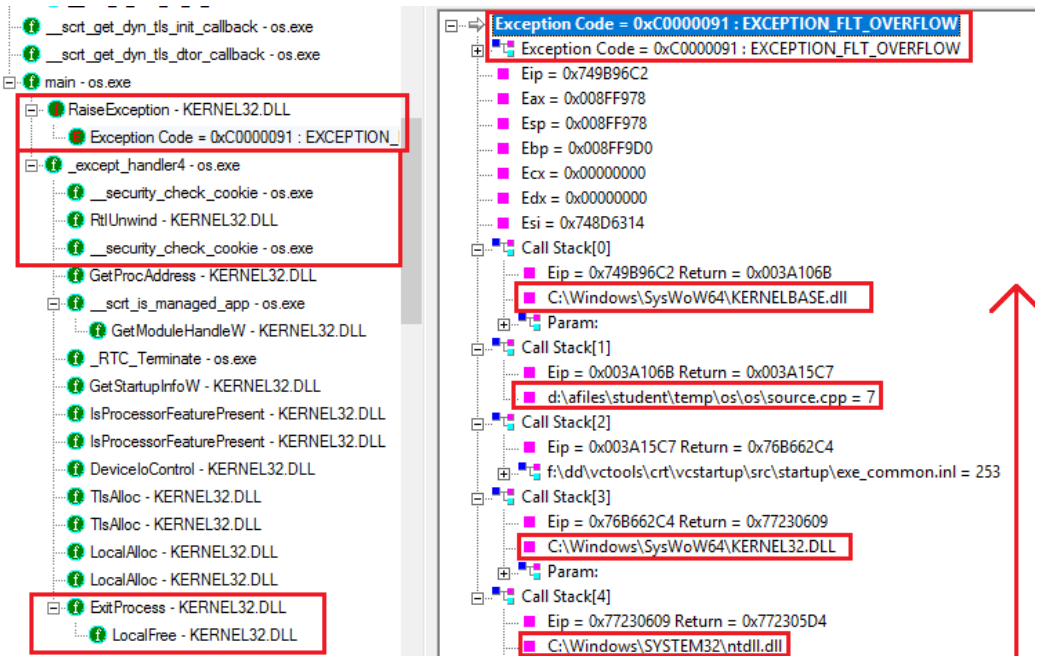

Рис. 1.4: Системные вызовы в программе обработки исключения EXCEPTION_FLT_OVERFLOW

После подготовительной стадии запуска процесса вызывается функция main. Первой выполняемой инструкцией данной программы является обращение к библиотеке ядра Windows KERNEL32.DLL функцией RaiseException. В результате выбрасывается исключение с кодом 0xC0000091 (EXCEPTION_FLT_OVERFLOW). Стек вызовов, производимых при выбрасывании исключения продемонстрирован на рисунке 1.4.

Все последующие операции производятся уже в обработчике исключения (_except_handler4). Стоит отметить, что в первую очередь производится **раскрутка стека** (RtlUnwind) – процесс, в результате которого программа последовательно покидает составные инструкции и определения функций в поисках обработчика, способного обработать возникшее исключение [3]. По мере раскрутки прекращают существование локальные объекты, объявленные в составных инструкциях и определениях функций, из которых произошел выход.

Иллюстрация процесса обработки исключения в деассемблере Ida

Рассмотрим процесс генерации исключения и раскрутки стека подробнее. Ida позволяет получить деассемблированную версию программы с возможностью осуществления пошаговой отладки, просмотра регистров процессора, просмотра кода внешних функций и др.

После прохождения всех подготовительных стадий вызывается главная функция main, из которой вызывается внешняя функция ядра RaiseException с тремя нулевыми аргументами и кодом исключения (0xC0000091 EXCEPTION_FLT_OVERFLOW):

```
.text:003E1050 mov     [ebp+var_18], esp
.text:003E1053 mov     [ebp+var_4], 0
.text:003E105A push    0                               ; lpArguments
.text:003E105C push    0                               ; nNumberOfArguments
.text:003E105E push    0                               ; dwExceptionFlags
.text:003E1060 push    0C0000091h                      ; dwExceptionCode
EIP .text:003E1065 call    ds:__imp__RaiseException@16     ; RaiseException(x,x,x,x)
.text:003E106B mov     [ebp+var_4], 0FFFFFFFEh
.text:003E1072 xor     eax, eax
.text:003E1074 mov     ecx, [ebp+var_10]
.text:003E1077 mov     large fs:0, ecx
.text:003E107E pop     ecx
.text:003E107F pop     edi
.text:003E1080 pop     esi
.text:003E1081 pop     ebx
.text:003E1082 mov     esp, ebp
```

Рис. 1.5: Вызов внешней функции RaiseException из функции main

Стоит отметить, что последующий код функции main соответствует нормальному выходу из охраняемого кода, то есть случаю, когда исключение не сгенерируется. В нашем случае эта ветка кода не будет вызвана.

Как и следовало ожидать, было сгенерировано исключение EXCEPTION_FLT_OVERFLOW:

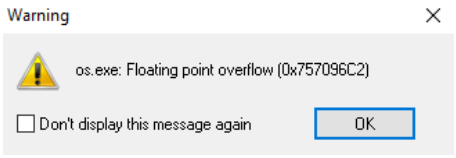

Рис. 1.6: Уведомление о произошедшем исключении

После этого вызывается функция фильтр, которая проверяет код исключения на совпадение с 0xC0000091:

```
ECX .text:003E1086 sub_3E1086 proc near
EIP .text:003E1086 mov     eax, [ebp-14h]
    .text:003E1089 mov     eax, [eax]
    .text:003E108B xor     ecx, ecx
    .text:003E108D cmp     dword ptr [eax], 0C0000091h
    .text:003E1093 setz    cl
    .text:003E1096 mov     eax, ecx
    .text:003E1098 retn
    .text:003E1098 sub_3E1086 endp
```

Рис. 1.7: Функция фильтр

После этого последовательно вызываются дорогостоящие функции раскрутки стека RtlUnwind в модулях ядра kernel32 и ntdll:

```
kernel32.dll:75A0C430 kernel32_RtlUnwind:
kernel32.dll:75A0C430 jmp     ds:off_75A714B8
```

Рис. 1.8: Функция раскрутки стека RtlUnwind в модуле kernel32

```
    ntdll.dll:7727E620 ntdll_RtlUnwind:
EIP ntdll.dll:7727E620 mov     edi, edi
    ntdll.dll:7727E622 push    ebp
    ntdll.dll:7727E623 mov     ebp, esp
    ntdll.dll:7727E625 and     esp, 0FFFFFFF8h
    ntdll.dll:7727E628 sub     esp, 384h
    ntdll.dll:7727E62E mov     eax, ds:dword_773332D4
    ntdll.dll:7727E633 xor     eax, esp
```

Рис. 1.9: Функция раскрутки стека RtlUnwind в модуле ntdll

Количество ассемблерных команд в RtlUnwind достаточно большое, что подтверждает ожидания.

Так как фильтр сработал, вызывается код обработчика, вызывающий внешнюю функцию вывода сообщения на экран:

```
ESI .text:003E1099 sub_3E1099 proc near                ; DATA XREF: .rdata:003E3528↓o
EIP .text:003E1099 mov     esp, [ebp-18h]
    .text:003E109C push    offset std__endl_char_std__char_traits_char___
    .text:003E10A1 push    0C0000091h
    .text:003E10A6 push    offset std__hex
    .text:003E10AB mov     ecx, ds:__imp_?cerr@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ;
    .text:003E10B1 call    std__operator___std__char_traits_char___
    .text:003E10B6 mov     ecx, eax
    .text:003E10B8 call    ds:__imp_??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@P6AAAVi
```

Рис. 1.10: Код обработчика исключения

1.4.2 Получить код исключения с помощью функции GetExceptionCode

Использовать эту функцию в выражении фильтре

Для получения кода выброшенного исключения, вызывается функция GetExceptionCode, имеющая следующий прототип:

```
1 DWORD GetExceptionCode(void);
```

Функцию можно вызывать внутри фильтра, а также непосредственно внутри обработчика исключения. Правилом хорошего тона при программировании является явное указание в фильтре всех возможных для данного охраняемого кода исключений. Таким образом те исключения, о которых программист не подумал дадут о себе знать на этапе отладки.

Разработаем программу, которая использует функцию GetExceptionCode внутри тернарного оператора фильтра:

```
#include <iostream>
#include <windows.h>

int main() {
  auto zero = 0, one = 1;
  __try {
    // Инициируем деление на ноль.
    const auto number = one / zero;
  } __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    std::cerr << "EXCEPTION_INT_DIVIDE_BY_ZERO: 0x" << std::hex <<
    EXCEPTION_INT_DIVIDE_BY_ZERO << std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_INT_DIVIDE_BY_ZERO: 0xc0000094
```

По умолчанию аппаратные исключения, связанные с числами с плавающей точкой, не нуждаются в обработке. Для того, чтобы явно указать необходимость обработки исключений с плавающей точкой, разработаем функцию activateFloatExceptions:

```
#pragma warning(disable : 4996)

#include <iostream>
#include <windows.h>

void activateFloatExceptions();

int main() {
  activateFloatExceptions();

  __try {
    // Инициируем переполнение переменной с плавающей точкой.
    pow(1e3, 1e3);
  } __except(GetExceptionCode() == EXCEPTION_FLT_OVERFLOW ? EXCEPTION_EXECUTE_HANDLER :
    EXCEPTION_CONTINUE_SEARCH) {
    std::cerr << "EXCEPTION_FLT_OVERFLOW: 0x" << std::hex << EXCEPTION_FLT_OVERFLOW <<
    std::endl;
    return 0x1;
  }

  return 0x0;
}

void activateFloatExceptions() {
  // Включаем обработку исключений с плавающей точкой.
  auto control = _controlfp(NULL, NULL);
  control &= ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL);
  _controlfp(control, _MCW_EM);
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_FLT_OVERFLOW: 0xc0000091
```

Таким образом, средствами WinAPI можно обрабатывать не только программные исключения, а также аппаратные, которые игнорируются по-умолчанию.

Использовать эту функцию в обработчике

Использование функции GetExceptionCode возможно также непосредственно внутри обработчика, однако, не стоит подменять фильтр этой возможностью, особенно при работе со вложенными исключениями.

Разработаем программу, получающую код исключения непосредственно внутри обработчика:

```
#include <iostream>
#include <windows.h>

int main() {
  auto zero = 0, one = 1;
  __try {
    // Инициируем деление на ноль.
    const auto number = one / zero;
  } __except(EXCEPTION_EXECUTE_HANDLER) {
    const auto exceptionCode = GetExceptionCode();
    std::cerr << "EXCEPTION_CODE: 0x" << std::hex << exceptionCode << std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_CODE: 0xc0000094
```

Теперь разработаем аналогичную программу для исключения с плавающей точкой:

```
#pragma warning(disable : 4996)

#include <iostream>
#include <windows.h>

void activateFloatExceptions();

int main() {
  activateFloatExceptions();

  __try {
    // Инициируем переполнение переменной с плавающей точкой.
    pow(1e3, 1e3);
  } __except(EXCEPTION_EXECUTE_HANDLER) {
    const auto exceptionCode = GetExceptionCode();
    std::cerr << "EXCEPTION_CODE: 0x" << std::hex << exceptionCode << std::endl;
    return 0x1;
  }

  return 0x0;
}

void activateFloatExceptions() {
  // Включаем обработку исключений с плавающей точкой.
  auto control = _controlfp(NULL, NULL);
  control &= ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL);
  _controlfp(control, _MCW_EM);
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_CODE: 0xc0000094
```

В результате эксперимента было выявлено, что функция GetExceptionCode выдает корректный результат при вызове непосредственно внутри обработчика.

1.4.3 Создать собственную функцию-фильтр

Для больших программ, в которых использование фильтрующего тернарного оператора недостаточно, функциональность фильтра бывает удобно перенести во внешнюю функцию. При этом данная функция должна возвращать один из трех идентификаторов, определяющих поведение программы.

Разработаем программу с отдельной фильтрующей функцией:

```
#include <iostream>
#include <windows.h>

DWORD filterException(const DWORD exceptionCode);

int main() {
  auto divider = 0, dividend = 1;
  auto number = dividend;
  __try {
    // Инициируем деление на ноль.
    number /= divider;
  } __except(filterException(GetExceptionCode())) {
    std::cerr << "Something goes wrong!" << number << std::endl;
    return 0x1;
  }

  return 0x0;
}

DWORD filterException(const DWORD exceptionCode) {
  return exceptionCode == EXCEPTION_INT_DIVIDE_BY_ZERO ? EXCEPTION_EXECUTE_HANDLER :
    EXCEPTION_CONTINUE_SEARCH;
}
```

Результат работы программы аналогичен фильтру с тернарным оператором:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
Something goes wrong!
```

Модифицируем программу, порождающую исключение переполнения числа с плавающей точкой. Программист пытается сгенерировать очень большое число функцией возведения в степень, однако, натыкается на переполнение в охраняемом коде. Функция фильтр фиксирует это исключение и изменяет эту переменную на другое очень большое число, после чего продолжает выполнение программы. Все остальные исключения обрабатываются непосредственно внутри обработчика.

```
#pragma warning(disable : 4996)

#include <iostream>
#include <windows.h>

void activateFloatExceptions();
DWORD filterException(const DWORD exceptionCode, float &wrongValue);

int main() {
  activateFloatExceptions();

  auto number = 1.f;
  __try {
    // Инициируем переполнение переменной с плавающей точкой.
    number = pow(1e3, 1e3);
  }
  __except(filterException(GetExceptionCode(), number)) {
    std::cerr << "Failed, number value is " << number << std::endl;
    return 0x1;
  }

  std::cerr << "Successfully, number value is " << number << std::endl;
  return 0x0;
}

```

```
26 void activateFloatExceptions() {
27   // Включаем обработку исключений с плавающей точкой.
28   auto control = _controlfp(NULL, NULL);
29   control &= ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL);
30   _controlfp(control, _MCW_EM);
31 }
32
33 DWORD filterException(const DWORD exceptionCode, float &wrongValue) {
34   if(exceptionCode == EXCEPTION_FLT_OVERFLOW) {
35     std::cout << "EXCEPTION_FLT_OVERFLOW: 0x" << std::hex << EXCEPTION_FLT_OVERFLOW <<
      std::endl;
36     std::cout << "Wrong float value: " << wrongValue << std::endl;
37
38     // Не можем установить pow(1e3, 1e3), значит установим другое большое число.
39     wrongValue = INT64_MAX;
40
41     // Исключение корректно обработано, продолжаем выполнение программы.
42     return EXCEPTION_CONTINUE_EXECUTION;
43   }
44
45   // Инициируем запуск обработчика.
46   return EXCEPTION_EXECUTE_HANDLER;
47 }
```

Результат работы программы:

```
1 PS D:\afiles\student\temp\os\Release> .\os.exe
2 EXCEPTION_FLT_OVERFLOW: 0xc0000091
3 Wrong float value: 1
4 Successfully, number value is inf
```

Исключение было выброшено внутри функции возведения в степень. После чего была вызвана функция фильтра. Значение переменной не было перезаписано, так как исключение было выброшено до этого. Затем внутри фильтра значение переменной перезаписывается и блок охраняемого кода нормально завершается.

1.4.4 Получить информацию об исключении, сгенерировать исключение

Помимо функции GetExceptionCode внутри фильтра или обработчика можно вызвать функцию GetExceptionInfo которая возвращает заполненную структуру, в полях которой содержится детальная информация об исключении. Сигнатура функции выглядит следующим образом:

```
1 LPEXCEPTION_POINTERS GetExceptionInformation(void);
```

Возвращаемая структура содержит указатели на структуру EXCEPTION_RECORD и на структуру CONTEXT. Структура EXCEPTION_RECORD содержит машинно-независимое описание исключения, CONTEXT содержит специфическое состояние процессора на момент исключения [4]

В большинстве случаев интересующая программиста информация содержится в структуре EXCEPTION_RECO со следующими полями:

- ExceptionCode – код исключения, совпадающий с результатом функции GetExceptionCode.
- ExceptionFlags – флаги исключения.
- ExceptionAddress – адрес, по которому было вызвано исключение.
- NumberParameters – количество элементов массива ExceptionInformation.
- ExceptionInformation – массив, содержащий наиболее детальную информацию о исключении. Может быть задан функцией RaiseException, а также определен для специфических исключений (например EXCEPTION_ACCESS_VIOLATION). В остальных случаях массив не определен.
- ExceptionRecord – указатель на связанную структуру EXCEPTION_RECORD.

Исключение можно сгенерировать функцией RaiseException со следующей сигнатурой [5]:

```
void WINAPI RaiseException(
    _In_       DWORD      dwExceptionCode,
    _In_       DWORD      dwExceptionFlags,
    _In_       DWORD      nNumberOfArguments,
    _In_ const ULONG_PTR *lpArguments
);
```

В качестве параметров обязательно указывается код исключения, который может быть собственным. Необязательными параметрами являются флаги исключения, а также массив детализированной информации с указанием количества аргументов.

Разработаем фильтрующую функцию, которая записывает в глобальную переменную информацию о последнем возникшем исключении, после чего в обработчике использует эти данные, если необходимо:

```
#include <iostream>
#include <windows.h>

EXCEPTION_RECORD exceptionRecord;

DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
    exceptionPointers);

int main() {
  __try {
    RaiseException(EXCEPTION_INT_DIVIDE_BY_ZERO, NULL, NULL, nullptr);
  } __except(filterException(GetExceptionCode(), GetExceptionInformation())) {
    std::cerr << "EXCEPTION_INT_DIVIDE_BY_ZERO: 0x" << std::hex <<
    EXCEPTION_INT_DIVIDE_BY_ZERO << std::endl;
    std::cout << "Exception code: 0x" << std::hex << exceptionRecord.ExceptionCode << std
    ::endl;
    std::cout << "Exception flags: 0x" << std::hex << exceptionRecord.ExceptionFlags <<
    std::endl;
    std::cout << "Exception record: 0x" << std::hex << exceptionRecord.ExceptionRecord <<
     std::endl;
    std::cout << "Exception address: 0x" << std::hex << exceptionRecord.ExceptionAddress
    << std::endl;
    std::cout << "Number parameters: " << exceptionRecord.NumberParameters << std::endl;
    return 0x1;
  }

  return 0x0;
}

DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
    exceptionPointers) {
  memcpy(&exceptionRecord, exceptionPointers->ExceptionRecord, sizeof(exceptionRecord));
  return exceptionCode == EXCEPTION_INT_DIVIDE_BY_ZERO ? EXCEPTION_EXECUTE_HANDLER :
    EXCEPTION_CONTINUE_SEARCH;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_INT_DIVIDE_BY_ZERO: 0xc0000094
Exception code: 0xc0000094
Exception flags: 0x0
Exception record: 0x00000000
Exception address: 0x749B96C2
Number parameters: 0
```

Разработаем аналогичную программу для исключения EXCEPTION_FLT_OVERFLOW:

```
#include <iostream>
#include <windows.h>

EXCEPTION_RECORD exceptionRecord;

```

```
6  DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
       exceptionPointers);
7
8  int main() {
9    __try {
10     RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
11   } __except(filterException(GetExceptionCode(), GetExceptionInformation())) {
12     std::cerr << "EXCEPTION_FLT_OVERFLOW: 0x" << std::hex << EXCEPTION_FLT_OVERFLOW <<
       std::endl;
13     std::cout << "Exception code: 0x" << std::hex << exceptionRecord.ExceptionCode << std
       ::endl;
14     std::cout << "Exception flags: 0x" << std::hex << exceptionRecord.ExceptionFlags <<
       std::endl;
15     std::cout << "Exception record: 0x" << std::hex << exceptionRecord.ExceptionRecord <<
        std::endl;
16     std::cout << "Exception address: 0x" << std::hex << exceptionRecord.ExceptionAddress
       << std::endl;
17     std::cout << "Number parameters: " << exceptionRecord.NumberParameters << std::endl;
18     return 0x1;
19   }
20
21   return 0x0;
22 }
23
24 DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
       exceptionPointers) {
25   memcpy(&exceptionRecord, exceptionPointers->ExceptionRecord, sizeof(exceptionRecord));
26   return exceptionCode == EXCEPTION_FLT_OVERFLOW ? EXCEPTION_EXECUTE_HANDLER :
       EXCEPTION_CONTINUE_SEARCH;
27 }
```

Результат работы программы:

```
1 PS D:\afiles\student\temp\os\Release> .\os.exe
2 EXCEPTION_FLT_OVERFLOW: 0xc0000091
3 Exception code: 0xc0000091
4 Exception flags: 0x0
5 Exception record: 0x00000000
6 Exception address: 0x749B96C2
7 Number parameters: 0
```

Таким образом, для простых исключений с целыми и нецелыми числами поля ExceptionFlags, ExceptionRecord и ExceptionInformation не определены. Это объясняется тем, что как правило такие исключения простые и не требуют каких либо дополнительных пояснений.

Рассмотрим стек вызовов при генерации исключения деления на ноль. Видно, что последний вызов, породивший исключение, имеет адрес 0x749B96C2 (поле Eip), что совпадает с информацией полученной в результате GetExceptionInformation. Более подробное описание регистров процессора находится в структуре CONTEXT, значение которых совпадает с информацией отладчика.

1.4.5 Использовать функции UnhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений

Помимо конструкции __try и __except в Windows имеется возможность превратить весь код в охраняемый, посредством установления фильтра на необработанные исключения. По большей части это считается плохим тоном, однако может быть полезно на стадии релиза приложения, когда при возникновении необработанной ошибки программа должна показывать диалоговое окно с ее описанием.

Установление фильтра на необработанные исключения производится функцией SetUnhandleExceptionFilter, которая имеет следующую сигнатуру [6]:

```
1 LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter(
2     _In_ LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
3 );
```

В качестве единственного аргумента принимается указатель на функцию фильтр. Возвращаемое значение содержит указатель на предыдущую функцию обработчик.

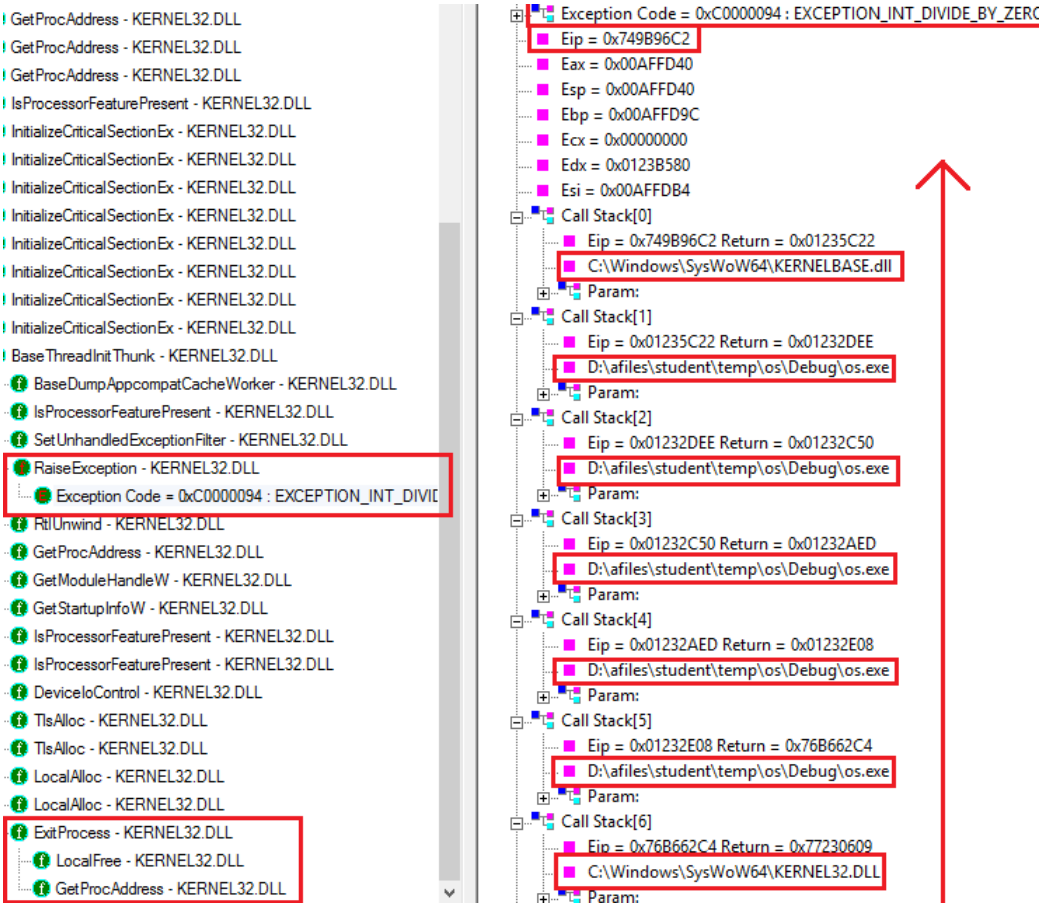

Рис. 1.11: Состояние регистров процессора и стек вызовов

Разработаем программу, которая устанавливает фильтр необработанных исключений, выводит адрес старого фильтра, нового фильтра и после этого выбрасывает исключение.

```
#include <iostream>
#include <windows.h>

EXCEPTION_RECORD exceptionRecord;

LONG WINAPI exceptionFilter(EXCEPTION_POINTERS* exceptionInformation);

int main() {
  // Устанавливаем обработчик исключений для всех необработанных исключений.
  const auto filterPointer = SetUnhandledExceptionFilter(exceptionFilter);
  std::cout << "OLD_EXCEPTION_HANDLER: 0x" << std::hex << filterPointer << std::endl
          << "NEW_EXCEPTION_HANDLER: 0x" << std::hex << exceptionFilter << std::endl;

  RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
  return 0x0;
}

LONG WINAPI exceptionFilter(EXCEPTION_POINTERS* exceptionInformation) {
  std::cout << "EXCEPTION_CODE: 0x" << std::hex << exceptionInformation->ExceptionRecord
    ->ExceptionCode << std::endl;
  return EXCEPTION_EXECUTE_HANDLER;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
OLD_EXCEPTION_HANDLER: 0x00F61C95
NEW_EXCEPTION_HANDLER: 0x00F610C0
EXCEPTION_CODE: 0xc0000091
```

Рассмотрим системные вызовы, произведенные в этой программе. Перед запуском функции main операционная система сама устанавливает стандартный фильтра необработанных исключений.

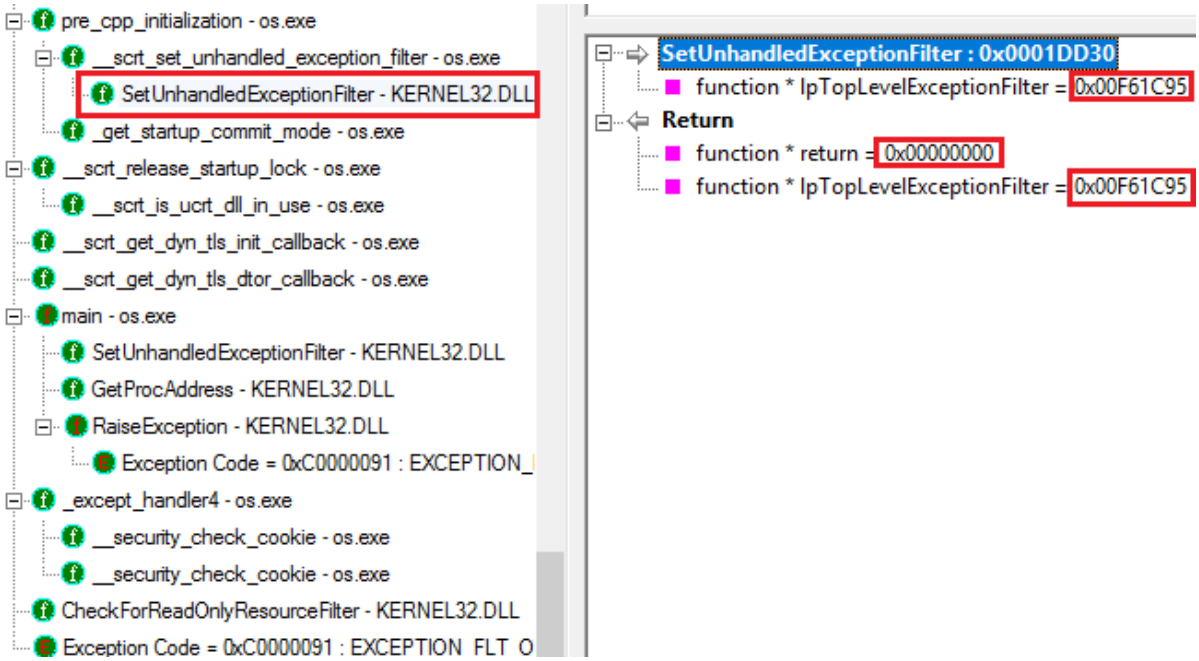

Рис. 1.12: Установка стандартного фильтра необработанных исключений при запуске процесса

Стоит отметить, что адрес этой функции соответствует адресу, выведенному в результате выполнения программы (OLD_EXCEPTION_HANDLER). Возвращаемое значение этой функции является нулевым, потому что до этого не было установлено ни одного фильтра необработанных исключений.

После этого, уже в теле функции main, вызывается функция SetUnhandleExceptionFilter, которая была объявлена в программном коде:

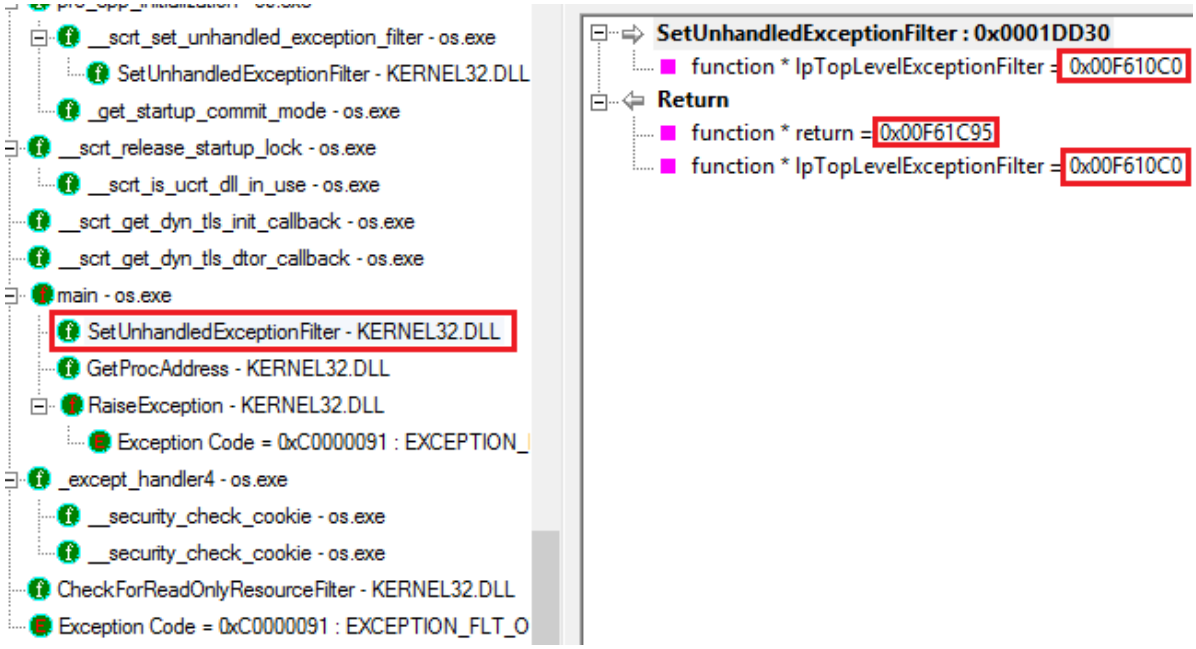

Рис. 1.13: Установка собственного фильтра необработанных исключений

Адрес этой функции соответствует адресу, выведенному в результате выполнения программы (NEW_EXCEPTION_HANDLER). Возвращаемое значение этой функции указывает на предыдущий фильтр, установленный операционной системой.

1.4.6 Обработать вложенные исключения

Архитектура обработки исключений в WinAPI позволяет обрабатывать вложенные блоки __try, __except. Для того, чтобы передать управление внешнему обработчику исключений из внутреннего, фильтр внутреннего обработчика должен возвращать EXCEPTION_CONTINUE_SEARCH.

Разработаем программу, которая содержит один блок __try, __except внутри другого. Внутренний обработчик вызывается на исключение EXCEPTION_INT_DIVIDE_BY_ZERO, в противном случае продолжит искать обработчик дальше. Внешний обработчик вызывается на исключение EXCEPTION_FLT_OVERFLOW, в противном случае продолжит искать обработчик дальше. Внутри охраняемого кода вызывается соответствующее исключение, в зависимости от аргумента командной строки.

```
#include <iostream>
#include <windows.h>

DWORD filterException(const DWORD exceptionCode, const DWORD exceptionExpect);

int main(const int argc, const char** argv) {
  __try {
    __try {
      // Если первый аргумент равен '1', то вызываем исключение EXCEPTION_FLT_OVERFLOW, если нет, то EXCEPTION_INT_DIVIDE_BY_ZERO.
      const auto exceptionCode = (argc > 1 && std::strcmp(argv[1], "1") == 0) ? EXCEPTION_FLT_OVERFLOW : EXCEPTION_INT_DIVIDE_BY_ZERO;
      RaiseException(exceptionCode, NULL, NULL, nullptr);
    }
    __except(filterException(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO)) {
      std::cout << "EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
      return 0x1;
    }
  } __except(filterException(GetExceptionCode(), EXCEPTION_FLT_OVERFLOW)) {
    std::cout << "EXCEPTION_FLT_OVERFLOW" << std::endl;
    return 0x2;
  }

  return 0x0;
}

DWORD filterException(const DWORD exceptionCode, const DWORD exceptionExpect) {
  if(exceptionCode == exceptionExpect) {
    // Вызываем обработчик.
    std::cout << "EXCEPTION_EXECUTE_HANDLER" << std::endl;
    return EXCEPTION_EXECUTE_HANDLER;
  }

  // Продолжаем поиск обработчика выше.
  std::cout << "EXCEPTION_CONTINUE_SEARCH" << std::endl;
  return EXCEPTION_CONTINUE_SEARCH;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe 0
EXCEPTION_EXECUTE_HANDLER
EXCEPTION_INT_DIVIDE_BY_ZERO

PS D:\afiles\student\temp\os\Release> .\os.exe 1
EXCEPTION_CONTINUE_SEARCH
EXCEPTION_EXECUTE_HANDLER
EXCEPTION_FLT_OVERFLOW
```

В первом варианте (аргумент 0 – EXCEPTION_INT_DIVIDE_BY_ZERO) внутренний фильтр вернул EXCEPTION_EXECUTE_HANDLER после чего был вызван обработчик и приложение завершилось.

Во втором варианте (аргумент 1 – EXCEPTION_FLT_OVERFLOW) внутренний фильтр вернул EXCEPTION_C после чего был найден внешний обработчик. Внешний фильтр вернул EXCEPTION_EXECUTE_HANDLER, был вызван обработчик и приложение завершилось.

Рассмотрим обработку вложенного исключение более подробно на примере второго варианта (аргумент 1 – EXCEPTION_FLT_OVERFLOW):

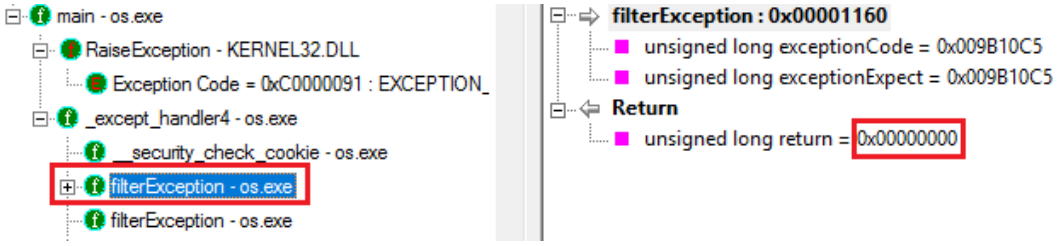

Рис. 1.14: Вызов фильтра внутреннего обработчика

Во внутреннем обработчике фильтр вернул 0 (EXCEPTION_CONTINUE_SEARCH).

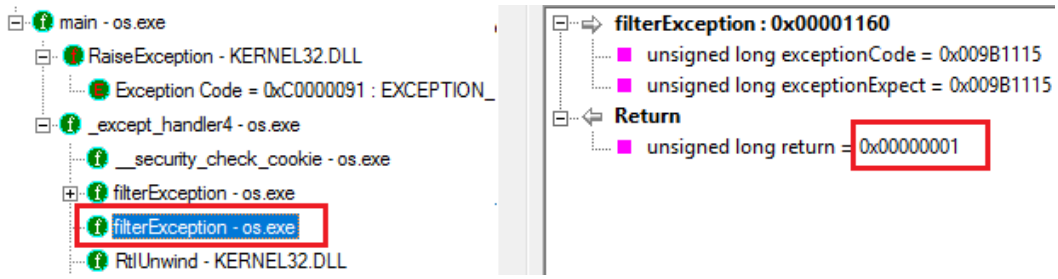

Рис. 1.15: Вызов фильтра внешнего обработчика

Во внешнем обработчике фильтр вернул 1 (EXCEPTION_EXECUTE_HANDLER). Кроме того, раскрутка стека (RtlUnwind) была произведена только при нахождении обработчика (EXCEPTION_EXECUTE_HANDLER при возвращении EXCEPTION_CONTINUE_SEARCH из внутреннего фильтра раскрутка стека не производится.

Иллюстрация процесса обработки вложенных исключения в деассемблере Ida

Рассмотрим процесс поиска обработчика для вложенных исключений с помощью деассемблера Ida для исключения 0xC0000091 EXCEPTION_FLT_OVERFLOW. После прохождения всех подготовительных стадий вызывается главная функция main, из которой вызывается внешняя функция ядра RaiseException с тремя нулевыми аргументами и кодом исключения:

```
.text:012D1085 loc_12D1085:                            ; CODE XREF: main+7E↑j
.text:012D1085 push    0                               ; lpArguments
.text:012D1087 push    0                               ; nNumberOfArguments
.text:012D1089 push    0                               ; dwExceptionFlags
.text:012D108B push    eax                             ; dwExceptionCode
EIP .text:012D108C call    ds:__imp__RaiseException@16 ; RaiseException(x,x,x,x)
.text:012D1092 mov     [ebp+var_4], 0
.text:012D1099 mov     [ebp+var_4], 0FFFFFFFEh
.text:012D10A0 xor     eax, eax
.text:012D10A2 mov     ecx, [ebp+var_10]
.text:012D10A5 mov     large fs:0, ecx
.text:012D10AC pop     ecx
.text:012D10AD pop     edi
.text:012D10AE pop     esi
.text:012D10AF pop     ebx
.text:012D10B0 mov     esp, ebp
.text:012D10B2 pop     ebp
.text:012D10B3 retn
.text:012D10B3 main endp
```

Рис. 1.16: Вызов внешней функции RaiseException из функции main

Стоит отметить, что последующий код функции main соответствует нормальному выходу из охраняемого кода, то есть случаю, когда исключение не сгенерируется. В нашем случае эта ветка кода не будет вызвана.

Как и следовало ожидать, было сгенерировано исключение EXCEPTION_FLT_OVERFLOW:

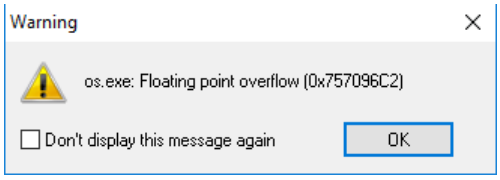

Рис. 1.17: Уведомление о произошедшем исключении

После этого мы попадаем в функцию except_handler, что свидетельствует о том, что исключение было успешно выброшено и начался поиск обработчика:

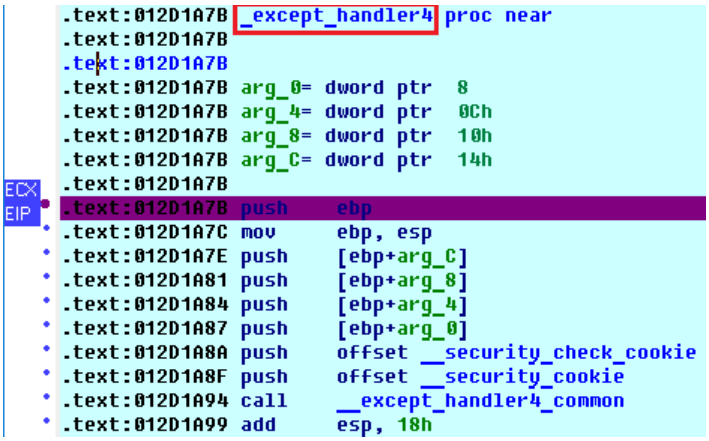

Рис. 1.18: Вызов функции except_handler, предшествующей опросу фильтров

В первую очередь вызывается первый фильтр, который проверяет код исключения на совпадение с 0xC0000094 EXCEPTION_INT_DIVIDE_BY_ZERO:

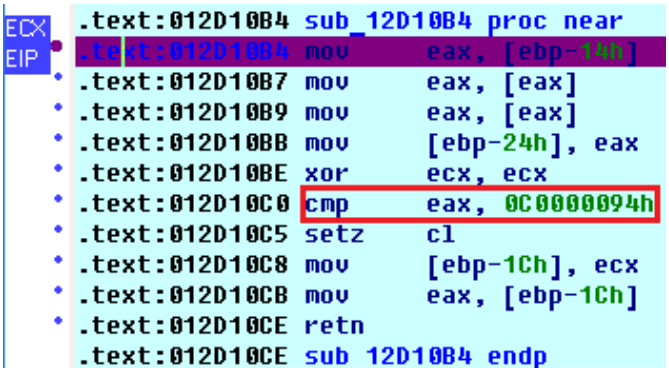

Рис. 1.19

Код выброшенного исключения не совпадает с кодом фильтра, поэтому вызывается фильтр для следующего обработчика, который проверяет код исключения на совпадение с 0xC0000091 EXCEPTION_FLT_OVERFLOW

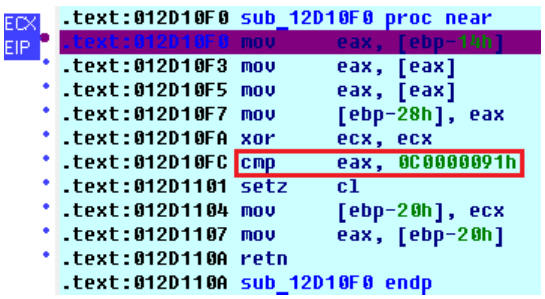

Рис. 1.20

Фильтр сработал и обработчик был успешно найден, о чем свидетельствует последующий вызов функций раскрутки стека RtlUnwind в модулях ядра kernel32 и ntdll:

```
kernel32.dll:75A0C430 kernel32_RtlUnwind:
kernel32.dll:75A0C430 jmp     ds:off_75A714B8
```

Рис. 1.21: Функция раскрутки стека RtlUnwind в модуле kernel32

```
EIP ntdll.dll:7727E620 ntdll_RtlUnwind:
    ntdll.dll:7727E620 mov     edi, edi
    ntdll.dll:7727E622 push    ebp
    ntdll.dll:7727E623 mov     ebp, esp
    ntdll.dll:7727E625 and     esp, 0FFFFFFF8h
    ntdll.dll:7727E628 sub     esp, 384h
    ntdll.dll:7727E62E mov     eax, ds:dword_773332D4
    ntdll.dll:7727E633 xor     eax, esp
    ntdll.dll:7727E635 mov     [esp+380h], eax
```

Рис. 1.22: Функция раскрутки стека RtlUnwind в модуле ntdll

Количество ассемблерных команд в RtlUnwind достаточно большое, что подтверждает ожидания.

1.4.7 Выйти из блока __try с помощью оператора goto

В WinAPI, а также в стандарте языка C++ разрешается выходить из охраняемого кода при помощи оператора goto. Использование оператора goto является плохой практикой в современных языках программирования, не только из-за абсолютного читабельности кода, но и из-за своей неэффективности. Каждый раз при использовании goto вызывается раскрутка стека, что делает данную операцию весьма медленной.

Однако, различные компиляторы языка C и C++ демонстрируют различное поведение ввиду внутренних оптимизаций. Например, компилятор GCC каждый раз раскручивает стек, в то время как MSVC от Microsoft добавляет оптимизацию выхода через goto из охраняемого кода, не раскручивая стек (скорее всего используется оптимизация через __leave). Таким образом, для языка C goto ускоряется, а для языка C++ становится опасным, ведь нет гарантии вызова деструкторов для объектов охраняемого кода.

Разработаем программу, которая в зависимости от аргумента командной строки, выходит из охраняемого кода через goto или __leave:

```
#include <iostream>
#include <windows.h>

int main(const int argc, const char** argv) {
  __try {

    // Если первый аргумент равен '1', то выходим через goto, если нет, то через __leave.
    const auto isGoto = argc > 1 && std::strcmp(argv[1], "1") == 0;
    if(isGoto)
      goto out;
    else
      __leave;

    // Программа должна завершиться до вызова исключения.
    RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
  } __except(EXCEPTION_EXECUTE_HANDLER) {
    std::cout << "Exception!" << std::endl;
    return 0x1;
  }

out:
  std::cout << "Without exception!" << std::endl;
  return 0x0;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe 1
Without exception!
```

Таким образом, goto позволил выйти из охраняемого кода. Раскрутка стека при этом не была вызвана ввиду внутренней оптимизации компилятора MSVC. Однако, goto по-прежнему считается не нормальным выходом из блока __try, __except, поэтому функция AbnormalTermination вернет 1.

1.4.8 Выйти из блока __try с помощью оператора __leave

В отличие от goto, команда __leave позволяет выходить только из охраняемого кода. Использование __leave в другом месте вызовет ошибку компиляции. Кроме того, __leave не позволяет выходить из вложенных блоков __try, __except. Однако, использование этой команды рекомендуется при работе с исключениями, потому что не вызывает раскрутку стека, а также увеличивает читабельность кода [7]

Подадим на вход командной строки 0, для использования команды __leave:

```
PS D:\afiles\student\temp\os\Release> .\os.exe 0
Without exception!
```

Результат аналогичен goto, однако __leave для всех компиляторов гарантированно не вызовет раскрутку стека.

1.4.9 Преобразовать структурное исключение в исключение языка C, используя функцию translator

Использовать одновременно исключения обоих типов, в программе на C++ проблематично, так как прийдется их обрабатывать по отдельности. Что-бы этого избежать SEH исключение нужно транслировать в обычное исключение. Делается это с помощью функции _set_se_translator стандартной библиотеки, сама эта функция стандартной не является. Она получает указатель на функцию транслятор, которая получает структуру описывающую исключение и в ответ, должна бросить типизированное исключение.

Сигнатура функции _set_se_translator выглядит следующим образом:

```
_se_translator_function _set_se_translator(
    _se_translator_function seTransFunction
);
```

Функция принимает на вход функцию транслятора и не возвращает значение. Функция транслятор принимает код структурного исключения и информацию о нем. Транслятор должен выбрасывать соответствующее исключение C++. Для того, чтобы компилятор MSVC разрешил использовать _set_se_translator, обязательно использование флага компилятора /EHa [8]

Разработаем программу, которая транслирует структурное исключение, вызванное функцией RaiseException в исключение языка C++. При этом код исключения передается как указатель через throw:

```
#include <iostream>
#include <windows.h>

void translator(const UINT exceptionCode, EXCEPTION_POINTERS* exceptionInformation);

int main() {
  // Работает только со включенной опцией /EHa в компиляторе VS.
  _set_se_translator(translator);

  try {
    RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
  } catch(const UINT exceptionCode) {
    std::cout << "EXCEPTION_FLT_OVERFLOW: 0x" << std::hex << exceptionCode << std::endl;
    return 0x2;
  }

  return 0x0;
}

void translator(const UINT exceptionCode, EXCEPTION_POINTERS* exceptionInformation) {
  throw exceptionCode;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
EXCEPTION_FLT_OVERFLOW: 0xc0000091
```

Доработаем программу, добавив возможность передачи структуры с информацией об исключении. Стоит отметить, что структуру нужно скопировать перед отправкой через throw, иначе она будет вычищена.

```
#include <iostream>
#include <windows.h>

void translator(const UINT exceptionCode, EXCEPTION_POINTERS* exceptionInformation);

int main() {
  // Работает только со включенной опцией /EHa в компиляторе VS.
  _set_se_translator(translator);

  try {
    RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
  } catch(const EXCEPTION_POINTERS* exceptionInformation) {
    const auto exceptionRecord = exceptionInformation->ExceptionRecord;
    std::cout << "Exception code: 0x" << std::hex << exceptionRecord->ExceptionCode <<
    std::endl;
    std::cout << "Exception flags: 0x" << std::hex << exceptionRecord->ExceptionFlags <<
    std::endl;
    std::cout << "Exception record: 0x" << std::hex << exceptionRecord->ExceptionRecord
    << std::endl;
    std::cout << "Exception address: 0x" << std::hex << exceptionRecord->ExceptionAddress
    << std::endl;
    std::cout << "Number parameters: " << exceptionRecord->NumberParameters << std::endl;
    return 0x2;
  }

  return 0x0;
}

void translator(const UINT exceptionCode, EXCEPTION_POINTERS* exceptionInformation) {
  EXCEPTION_POINTERS result;
  std::memcpy(&result, exceptionInformation, sizeof(result));
  throw &result;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe
Exception code: 0xc0000091
Exception flags: 0x0
Exception record: 0x00000000
Exception address: 0x749B96C2
Number parameters: 0
```

1.4.10 Использовать финальный обработчик finally

Помимо конструкции __try, __except поддерживается также конструкция __try, __finally. Блок __finally был создан для задачи высвобождения ресурсов и будет вызван **в любом случае** после завершения охраняемого кода [9] Докажем это, разработаем программу, которая выходит из охраняемого кода пятью различными способами:

```
#include <iostream>
#include <windows.h>

int main(const int argc, const char** argv) {
  if(argc != 2 || std::strlen(argv[1]) != 1)
    return 0x1;

  __try {
    std::cout << "Try block." << std::endl;

    switch(argv[1][0]) {
      case '0':
        // Самостоятельное завершение __try.
```

```
14          break;
15
16        case '1':
17          // Нормальное завершение __try.
18          __leave;
19
20        case '2':
21          // Безусловное завершение __try.
22          goto out;
23
24        case '3':
25          // Завершение __try с исключением.
26          RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
27          break;
28
29        case '4':
30          // Завершение __try выходом из функции.
31          return 0x0;
32
33        default:
34          return 0x2;
35      }
36    } __finally {
37      std::cout << "Finally block." << std::endl;
38    }
39
40 out:
41   return 0x0;
42 }
```

Вне зависимости от способа выхода из охраняемого кода, блок __finally должен выполняться. Результат работы программы для всех пяти способов:

```
1  PS D:\afiles\student\temp\os\Release> .\os.exe 0
2  Try block.
3  Finally block.
4
5  PS D:\afiles\student\temp\os\Release> .\os.exe 1
6  Try block.
7  Finally block.
8
9  PS D:\afiles\student\temp\os\Release> .\os.exe 2
10 Try block.
11 Finally block.
12
13 PS D:\afiles\student\temp\os\Release> .\os.exe 3
14 Try block.
15 Finally block.
16
17 PS D:\afiles\student\temp\os\Release> .\os.exe 4
18 Try block.
19 Finally block.
```

Таким образом блок __finally был вызван во всех пяти различных вариантах. В данном примере не были рассмотрены варианты выхода через break и continue, но результаты аналогичны.

1.4.11 Проверить корректность выхода из блока __try с помощью функции AbnormalTermination в финальном обработчике

Корректность выхода из охраняемого кода может повлиять на освобождение ресурсов в блоке __finally. Для этого была создана функция AbnormalTermination, имеющая следующую сигнатуру [10]:

```
1 BOOL AbnormalTermination(void);
```

Функция возвращает 0, если завершение нормальное и 1, если нет. Данная функция может быть вызвана только из блока __finally.

Корректным выходом из охраняемого кода считается самостоятельное завершение и команда __leave. Все остальные варианты выхода являются не нормальным завершением и должны вызывать раскрутку стека. Однако, MSVC от Microsoft добавляет оптимизацию выхода из охраняемого кода, не раскручивая стек (скорее всего используя __leave), что ускоряет работу кода, но паразитно для проведения экспериментов. Тем не мение, AbnormalTermination работает правильно, согласно спецификации.

Дополним программу из предыдущего пункта, добавив обработку нормального и не нормального завершения в блок __finally:

```
#include <iostream>
#include <windows.h>

int main(const int argc, const char** argv) {
  if(argc != 2 || std::strlen(argv[1]) != 1)
    return 0x1;

  __try {
    std::cout << "Try block." << std::endl;

    switch(argv[1][0]) {
      case '0':
        // Самостоятельное завершение __try.
        break;

      case '1':
        // Нормальное завершение __try.
        __leave;

      case '2':
        // Безусловное завершение __try.
        goto out;

      case '3':
        // Завершение __try с исключением.
        RaiseException(EXCEPTION_FLT_OVERFLOW, NULL, NULL, nullptr);
        break;

      case '4':
        // Завершение __try выходом из функции.
        return 0x0;

      default:
        return 0x2;
    }
  } __finally {
    std::cout << "Finally block." << std::endl;

    if(AbnormalTermination()) {
      std::cout << "Abnormal termination." << std::endl;
    } else {
      std::cout << "Normal termination." << std::endl;
    }
  }

out:
  return 0x0;
}
```

Результат работы программы:

```
PS D:\afiles\student\temp\os\Release> .\os.exe 0
Try block.
Finally block.
Normal termination.

PS D:\afiles\student\temp\os\Release> .\os.exe 1
Try block.
Finally block.
```

```
Normal termination.

PS D:\afiles\student\temp\os\Release> .\os.exe 2
Try block.
Finally block.
Abnormal termination.

PS D:\afiles\student\temp\os\Release> .\os.exe 3
Try block.
Finally block.
Abnormal termination.

PS D:\afiles\student\temp\os\Release> .\os.exe 4
Try block.
Finally block.
Abnormal termination.
```

Только в первых двух случаях (самостоятельное завершение и команда __leave) охраняемый код завершился нормально, в то время как в остальных случаях завершение было не нормальным.

1.5 Вывод

В результате работы были изучены структурные исключения SEH. Из преимуществ данного способа обработки исключений, по сравнению со встроенными средствами языка C++ является:

- возможность обработки аппаратных исключений и просмотра регистров процессора на момент их возникновения;
- поддерживаются как в языке C, так и в C++;
- возможность транслирования исключений в исключения языка C++.

Из минусов стоит отметить:

- зависимость от конкретной платформы, в то время как исключения языка C++ стандартизованы.

Кроме того, не стоит забывать, что раскрутка стека является достаточно трудоемкой операцией, поэтому программист должен отдавать себе отчет в том, какие операции вызывают нормальное завершение охраняемого кода, а какие нет. Стоит отметить, что иногда компилятор может встроить дополнительную оптимизацию, для того чтобы не раскручивать стек в таких ситуациях, однако, такие улучшения обычно не стандартизированы, не надежны и тяжело отслеживаются на этапе отладки.

1.6 Список литературы

[1] Эксплуатирование SEH в среде Win32 [Электронный ресурс]. — URL: http://www.securitylab.ru/contest/212085.php (дата обращения 05.10.2017).

[2] ОБРАБОТКА ИСКЛЮЧЕНИЙ В VISUAL C++ [Электронный ресурс]. — URL: http://www.avprog.narod.ru/progs/exceptions.htm (дата обращения 05.10.2017).

[3] Раскрутка стека. C++ для начинающих [Электронный ресурс]. — URL: https://it.wikireading.ru/35947 (дата обращения 05.10.2017).

[4] EXCEPTION_RECORD structure (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/aa363082(v=vs.85).aspx (дата обращения 05.10.2017).

[5] RaiseException function (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms680552(v=vs.85).aspx (дата обращения 05.10.2017).

[6] SetUnhandledExceptionFilter function (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms680634(v=vs.85).aspx (дата обращения 05.10.2017).

[7] Оператор try-except [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/s58ftw19.aspx (дата обращения 05.10.2017).

[8] Очень серьезный блог: Обработка исключений и корректность программ на C++. [Электронный ресурс]. — URL: http://evgeny-lazin.blogspot.ru/2008/07/blog-post.html (дата обращения 05.10.2017).

[9] Оператор try-finally [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/9xtt5hxz.aspx (дата обращения 05.10.2017).

[10] AbnormalTermination macro (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms679265(v=vs.85).aspx (дата обращения 05.10.2017).