Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №1

по курсу «Системное программирование»

по теме «Обработка исключений в Windows»

Выполнил студент гр. 13541/2:
Волкова М.Д.

Проверил преподаватель:
Душутина Е. В.

Санкт-Петербург
2019 г.

1 Цель работы

Познакомится с видами исключений в операционной системе Windows и со способами их обработки.

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*
5. Сгенерировать исключение с помощью функции *RaiseException*
6. Использовать функции *UnhandleExceptionFilter* и *Set UnhandleExceptionFilter* для необработанных исключений
7. Вложенная обработка исключений
8. Выйти из блока *__try* с помощью оператора *goto*
9. Выйти из блока *__try* с помощью оператора *leave*
10. Преобразовать структурное исключение в исключение языка C, используя функцию *translator*
11. Использовать финальный обработчик *__finally*
12. Проверить корректность выхода из блока *__try* с помощью функции *AbnormalTermination* в финальном обработчике *__finally*

3 Ход работы

3.1 Характеристики системы

C:/Users/stakenschneider>systeminfo
Имя узла: MASHA
Название ОС: Майкрософт Windows 10 Pro
Версия ОС: 10.0.17134 Н/Д построение 17134
Изготовитель ОС: Microsoft Corporation
Параметры ОС: Изолированная рабочая станция
Сборка ОС: Multiprocessor Free
Зарегистрированный владелец: Маша
Зарегистрированная организация:
Код продукта: 00331-10000-00001-AA048
Дата установки: 28.09.2018, 23:02:07
Время загрузки системы: 09.01.2019, 19:08:35
Изготовитель системы: Acer
Модель системы: Aspire Z5700
Тип системы: x64-based PC
Процессор(ы): Число процессоров - 1. [01]: Intel64 Family 6 Model 30 Stepping 5 GenuineIntel 2801 МГц
Версия BIOS: American Megatrends Inc. P01-A2, 12.03.2010
Папка Windows: C:/Windows
Системная папка: C:/Windows/system32
Устройство загрузки: /Device/HarddiskVolume4
Язык системы: ru;Русский
Язык ввода: ru;Русский
Часовой пояс: (UTC+03:00) Москва, Санкт-Петербург
Полный объем физической памяти: 8 151 МБ
Доступная физическая память: 2 285 МБ

Виртуальная память: Макс. размер: 19 927 МБ
Виртуальная память: Доступна: 6 794 МБ
Виртуальная память: Используется: 13 133 МБ
Расположение файла подкачки: C:/pagefile.sys
Домен: WORKGROUP
Сервер входа в сеть: MASHA

3.2 Генерация и обработка исключений

В 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.

```
1 __try
2 {
3 // secure code
4 }
5 __except (/* exception filter */)
6 {
7 // exception handler
8 }
```

Если при выполнении защищенного кода из блока __try возникнет исключение, то ОС 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, 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:

- 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.

Для примера генерации обработки исключения будут использованы исключения EXCEPTION_INT_DIVIDE_BY_ZERO (целочисленное деление на ноль) и EXCEPTION_NONCONTINUABLE_EXCEPTION (попытка возобновления исполнения программы после исключения, которое запрещает выполнять такое действие). Первое из них будет брошено системой, из-за попытки выполнения операции деления на ноль. Для получения второго исключения будем использовать вызов функции RaiseException для выбрасавывания любого исключения с указанием флага, что после него продолжение програмы невозможно.

Листинг 1: Деление на ноль

```
int throw_exception_divide_by_zero()
{
  int one = 1;
  int zero = 0;
  return one / zero;
}
```

Листинг 2: Попытка доступа к данным с невыровненным адресом

```
void throw_exception_noncontinuable_exception()
{
    RaiseException(1, EXCEPTION_NONCONTINUABLE, 0, NULL);
}
```

Листинг 3: Обработка ошибок

```
int main()
{
  __try
  {
    throw_exception_divide_by_zero();
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    std::cout << "Caught \"divide by zero\" exception" << std::endl;
  }
  __try
  {
    throw_exception_noncontinuable_exception();
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    std::cout << "Caught \"noncontinuable_exception\"" << std::endl;
  }
  _getch();
  return 0;
}
```

Функция *main* генерирует исключения и сразу обрабатывает их, используя оператор __ *except*, а именно выводит на экран сообщение о пойманном исключении.

```
Caught "divide by zero" exception
Caught "noncontinuable exception" exception
Press any key to continue . . .
```

Windows Debugger

```
************* Path validation summary **************
Response                         Time (ms)     Location
eax=00000000 ebx=00000000 ecx=80640000 edx=00000000 esi=009fd000 edi=773cd724
eip=774680c9 esp=00b7f2f0 ebp=00b7f31c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2b:
774680c9 cc              int     3
Breakpoint 0 hit
eax=67d5abe0 ebx=009fd000 ecx=bf68679d edx=67c93a68 esi=00b7f6d0 edi=00b7f7bc
eip=00e32801 esp=00b7f6d0 ebp=00b7f7d4 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
Project!main+0xe1:
00e32801 8bf4            mov     esi,esp
0:000> g
(1840.1ca4): Integer divide-by-zero - code c0000094 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000002 ebx=009fd000 ecx=bf68679d edx=00000000 esi=00b7f6d0 edi=00b7f7bc
eip=00e32838 esp=00b7f6d0 ebp=00b7f7d4 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
Project!main+0x118:
00e32838 f77dd0          idiv    eax,dword ptr [ebp-30h] ss:002b:00b7f7a4=00000000
```

Анализ возникшего исключения с помощью команды !analyze –v

```
0:000> !analyze -v
*******************************************************************************
*                        Exception Analysis                                   *
*******************************************************************************

KEY_VALUES_STRING: 1

    Key  : Timeline.OS.Boot.DeltaSec
    Value: 13633

    Key  : Timeline.Process.Start.DeltaSec
    Value: 84

FAULTING_IP:
Project!main+118 [E:\Sys\Project\main.cpp @ 13]
00e32838 f77dd0          idiv    eax,dword ptr [ebp-30h]
EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 00e32838 (Project1!main+0x00000118)
   ExceptionCode: c0000094 (Integer divide-by-zero)
  ExceptionFlags: 00000000
NumberParameters: 0
FAULTING_THREAD:  00001ca4
DEFAULT_BUCKET_ID:  INTEGER_DIVIDE_BY_ZERO
PROCESS_NAME:  Project.exe
ERROR_CODE: (NTSTATUS) 0xc0000094 - <Unable to get error code text>
EXCEPTION_CODE: (NTSTATUS) 0xc0000094 - <Unable to get error code text>
EXCEPTION_CODE_STR:  c0000094
BUGCHECK_STR:  INTEGER_DIVIDE_BY_ZERO
PRIMARY_PROBLEM_CLASS:  INTEGER_DIVIDE_BY_ZERO
PROBLEM_CLASSES:

    ID:      [0n321]
    Type:    [@APPLICATION_FAULT_STRING]
    Class:   Primary
    Scope:   DEFAULT_BUCKET_ID (Failure Bucket ID prefix)
             BUCKET_ID
    Name:    Omit
    Data:    Add
             String: [INTEGER_DIVIDE_BY_ZERO]
    PID:     [Unspecified]
    TID:     [Unspecified]
    Frame:   [0]

LAST_CONTROL_TRANSFER:  from 00e3320e to 00e32838
STACK_TEXT:
00b7f7d4 00e3320e 00000001 00bf6400 00bfca60 Project1!main+0x118
00b7f7e8 00e33077 d84b3fd1 00e313cf 00e313cf Project1!invoke_main+0x1e
00b7f844 00e32f0d 00b7f854 00e33288 00b7f868 Project1!__scrt_common_main_seh+0x157
00b7f84c 00e33288 00b7f868 76f58484 009fd000 Project1!__scrt_common_main+0xd
00b7f854 76f58484 009fd000 76f58460 ae57d471 Project1!mainCRTStartup+0x8
00b7f868 7742305a 009fd000 afe3de48 00000000 KERNEL32!BaseThreadInitThunk+0x24
00b7f8b0 7742302a ffffffff 7743ecb7 00000000 ntdll!__RtlUserThreadStart+0x2f
00b7f8c0 00000000 00e313cf 009fd000 00000000 ntdll!_RtlUserThreadStart+0x1b
STACK_COMMAND:  ~0s ; .cxr ; kb
FAULT_INSTR_CODE:  89d07df7
FAULTING_SOURCE_LINE_NUMBER:  13
FAULTING_SOURCE_CODE:
    11:   int one = 1;
    12:   int zero = 0;
>   13:   return one / zero;
    14: }
    15:
FAILURE_EXCEPTION_CODE:  c0000094
BUCKET_ID_PREFIX_STR:  INTEGER_DIVIDE_BY_ZERO_
```

```
FAILURE_PROBLEM_CLASS:  INTEGER_DIVIDE_BY_ZERO
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=774d79a0
eip=7742a52c esp=00b7f6bc ebp=00b7f794 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll!NtTerminateProcess+0xc:
7742a52c c20800          ret     8
```

3.3 Получение кода исключения

Для того, чтобы получить какую-либо информацию об обрабатываемом исключении можно использовать функцию *GetExceptionCode()*, которая возвращает код полученного исключения. Функция *GetExceptionCode* может вызываться только в выражении-фильтре или в блоке обработки исключения. Следовательно, эта функция вызывается всегда только в том случае, если исключение произошло. Отсюда можно определить назначение функции *GetExceptionCode*. Если эта функция вызывается в выражении фильтра, то она используется для того, чтобы определить выполняет ли текущий обработчик исключения обработку исключений с данным кодом или нужно продолжить поиск подходящего обработчика исключения. Если же функция *GetExceptionCode* вызывается в блоке обработки исключения, то она также предназначена для проверки кодов исключений, которые обрабатывает текущий обработчик исключения, но в этом случае поиск другого обработчика исключений не выполняется.

```
void handle_exception(DWORD code)
{
  std::cout << "Caught exception with code = " << code << std::endl;
  switch (code)
  {
  case EXCEPTION_INT_DIVIDE_BY_ZERO:
    std::cout << "EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
    break;
  case EXCEPTION_NONCONTINUABLE_EXCEPTION:
    std::cout << "EXCEPTION_NONCONTINUABLE_EXCEPTION" << std::endl;
    break;
  default:
    std::cout << "unrecognized exception" << std::endl;
    break;
  }
}
```

```
int main()
{
  __try
  {
    throw_exception_divide_by_zero();
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    handle_exception(GetExceptionCode());
  }
  __try
  {
    throw_exception_noncontinuable_exception();
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    handle_exception(GetExceptionCode());
  }
  _getch();
  return 0;
}
```

```
Caught exception with code = 3221225620
EXCEPTION_INT_DIVIDE_BY_ZERO
Caught exception with code = 3221225509
EXCEPTION_NONCONTINUABLE_EXCEPTION
Press any key to continue . . .
```

Рис. 1: Вывод программы

Так же можно использовать данную функцию в выражении-фильтре:

```
LONG filter(DWORD actual_code, DWORD expected_code)
{
```

```
  if (actual_code == expected_code)
  {
    return EXCEPTION_EXECUTE_HANDLER;
  }
  else
  {
    return EXCEPTION_CONTINUE_SEARCH;
  }
}
```

```
int main()
{
  __try
  {
    throw_exception_divide_by_zero();
  }
  __except (filter(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO))
  {
    std::cout << "Caught EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
  }
  __try
  {
    throw_exception_noncontinuable_exception();
  }
  __except (filter(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION))
  {
    std::cout << "Caught EXCEPTION_NONCONTINUABLE_EXCEPTION" << std::endl;
  }
  _getch();
  return 0;
}
```

```
Caught EXCEPTION_INT_DIVIDE_BY_ZERO
Caught EXCEPTION_NONCONTINUABLE_EXCEPTION
Press any key to continue . . .
```

Рис. 2: Вывод программы

Windows Debugger

```
************* Path validation summary **************
Response                         Time (ms)     Location
eax=00000000 ebx=00000000 ecx=3c6c0000 edx=00000000 esi=00f22000 edi=7779d6b4
eip=77838679 esp=010ff380 ebp=010ff3ac iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2b:
77838679 cc              int     3
(1a5c.1f94): Integer divide-by-zero - code c0000094 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000006 ebx=00f22000 ecx=1e8bd75b edx=00000000 esi=010ff754 edi=010ff84c
eip=00b62838 esp=010ff754 ebp=010ff864 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
Project!main+0x118:
00b62838 f77dd0          idiv    eax,dword ptr [ebp-30h] ss:002b:010ff834=00000000
0:000> !analyze -v
*******************************************************************************
*                        Exception Analysis                                   *
*******************************************************************************

KEY_VALUES_STRING: 1

    Key  : Timeline.OS.Boot.DeltaSec
    Value: 827

    Key  : Timeline.Process.Start.DeltaSec
    Value: 41
FAULTING_IP:
Project!main+118 E:\Sys\Project\main.cpp programming\lab1\Project1\main.cpp @ 13]
00b62838 f77dd0          idiv    eax,dword ptr [ebp-30h]

EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 00b62838 (Project!main+0x00000118)
   ExceptionCode: c0000094 (Integer divide-by-zero)
  ExceptionFlags: 00000000
NumberParameters: 0

FAULTING_THREAD:  00001f94
DEFAULT_BUCKET_ID:  INTEGER_DIVIDE_BY_ZERO
PROCESS_NAME:  Project.exe
```

```
ERROR_CODE: (NTSTATUS) 0xc0000094 - <Unable to get error code text>
EXCEPTION_CODE: (NTSTATUS) 0xc0000094 - <Unable to get error code text>
EXCEPTION_CODE_STR:  c0000094
BUGCHECK_STR:  INTEGER_DIVIDE_BY_ZERO
PRIMARY_PROBLEM_CLASS:  INTEGER_DIVIDE_BY_ZERO
PROBLEM_CLASSES:

    ID:     [0n321]
    Type:   [@APPLICATION_FAULT_STRING]
    Class:  Primary
    Scope:  DEFAULT_BUCKET_ID (Failure Bucket ID prefix)
            BUCKET_ID
    Name:   Omit
    Data:   Add
            String: [INTEGER_DIVIDE_BY_ZERO]
    PID:    [Unspecified]
    TID:    [Unspecified]
    Frame:  [0]
LAST_CONTROL_TRANSFER:  from 00b6322e to 00b62838
STACK_TEXT:
010ff864 00b6322e 00000001 013346d0 01334878 Project2!main+0x118
010ff878 00b63097 7e8f510d 00b613cf 00b613cf Project2!__scrt_narrow_environment_policy::
    initialize_environment+0x2e
010ff8d4 00b62f2d 010ff8e4 00b632a8 010ff8f8 Project2!_except_handler4+0x2b7
010ff8dc 00b632a8 010ff8f8 740f8484 00f22000 Project2!_except_handler4+0x14d
010ff8e4 740f8484 00f22000 740f8460 0a20975a Project!mainCRTStartup+0x8
010ff8f8 777f2ec0 00f22000 09b36137 00000000 KERNEL32!BaseThreadInitThunk+0x24
010ff940 777f2e90 ffffffff 7780debd 00000000 ntdll!__RtlUserThreadStart+0x2f
010ff950 00000000 00b613cf 00f22000 00000000 ntdll!_RtlUserThreadStart+0x1b
STACK_COMMAND:  ~0s ; .cxr ; kb
FAULT_INSTR_CODE:  89d07df7
FAULTING_SOURCE_LINE:
E:\Sys\Project\main.cpp
FAULTING_SOURCE_FILE:
E:\Sys\Project\main.cpp
FAULTING_SOURCE_LINE_NUMBER:  13
FAULTING_SOURCE_CODE:
    11:     int one = 1;
    12:     int zero = 0;
>   13:     return one / zero;
    14: }
    15:
FAILURE_EXCEPTION_CODE:  c0000094
SYMBOL_STACK_INDEX:  0
BUCKET_ID:  INTEGER_DIVIDE_BY_ZERO_Project2!main+118
FAILURE_EXCEPTION_CODE:  c0000094
BUCKET_ID_PREFIX_STR:  INTEGER_DIVIDE_BY_ZERO_
FAILURE_PROBLEM_CLASS:  INTEGER_DIVIDE_BY_ZERO
```

3.4 Получение информации об исключении с помощью GetExceptionInformation

Более подробную информацию об исключении можно получить при помощи вызова функции GetExceptionlnformation, которая имеет следующий прототип:

```
LPEXCEPTION_POINTERS GetExceptionlnformation(VOID);
```

Эта функция возвращает указатель на структуру типа:

```
typedef struct EXCEPTION_POINTERS {
    PEXEPTION_RECORD Except ionRecord;
    PCONTEXT Context;
} EXCEPTION_POINTERS, * PEXCEPTION_POINTERS;
```

которая, в свою очередь, содержит два указателя: ExceptionRecord и context на структуры типа exception_record и context соответственно.

В структуру типа context система записывает содержимое всех регистров процессора на момент исключения. Эта структура имеет довольно громоздкое описание, которое можно найти в заголовочном файле WinNt.h.

Структура типа exception_record имеет следующий формат:

```
typedef struct EXCEPTION_RECORD {
    DWORD Except i onCode;
    DWORD ExceptionFlags,
    strict _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress,
    DWORD Number Parameters,
    ULONG_PTR Exceptionlnf ormation [ EXCEPTION_MAXIMUM_PARAMETERS ] ;
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

В нее система записывает информацию об исключении. Поля этой структуры имеют следующее назначение. Поле ExceptionCode содержит код исключения, который может принимать такие же значения, как

и код исключения, возвращаемый функцией GetExceptionCode. Поле ExceptionFlags может принимать одно из двух значений:

- 0 — которое обозначает, что после обработки исключения возможно возобновление выполнения программы;
- EXCEPTION_NONCONTINUABLE — которое обозначает, что после обработки исключения возобновление выполнения программы невозможно.

Если установлено значение EXCEPTION_NONCONTINUABLE и выполнена попытка возобновления выполнения программы, то система выбросит исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

Поле ExceptionRecord содержит указатель на следующую структуру типа exception_record, которая может быть создана в случае вложенных исключений.

Поле ExceptionAddress содержит адрес инструкции в программе, на которой произошло исключение.

Поле NumberParameters содержит количество параметров, заданных в поле Exceptioninformation, которое является последним в этой структуре.

Поле Exceptionlnformation[EXCEPTION_MAXIMUM_PARAMETERS] Определяет массив 32-битных аргументов, которые описывают исключение. Элементы этого массива могут использоваться функцией генерации программных исключений RaiseException.

Сделаем важное замечание о том, что функция GetExceptionlnformation может вызываться только в выражении фильтра. Поэтому эта функция вызывается всегда только в том случае, если исключение произошло. Кроме того, структуры типа exception_pointers, exception_record и context действительны только на время вычисления выражения-фильтра. Чтобы использовать содержимое структур типа exception_record и context в блоке обработки исключения, его нужно сохранить в объявленных в программе переменных такого же типа. Как видно из описания структуры EXCEPTION_RECORD, функцию GetExceptionlnformation можно использовать для двух целей: первая цель заключается в получении более подробной информации об исключении, учитывая содержимое структуры типа context; вторая цель состоит в обработке вложенных исключений.

```
1  EXCEPTION_RECORD er;
2
3  LONG filter(DWORD actualCode, DWORD expectedCode, EXCEPTION_POINTERS* exceptionPointers)
4  {
5    er = *(exceptionPointers->ExceptionRecord);
6    return filter(actualCode, expectedCode);
7  }
8
9  void show_info()
10 {
11   std::cout << "code: " << er.ExceptionCode << std::endl;
12   std::cout << "flags: " << er.ExceptionFlags << std::endl;
13   std::cout << "address: " << er.ExceptionAddress << std::endl;
14   std::cout << "record: " << er.ExceptionRecord << std::endl;
15   std::cout << "NumberParameters: " << er.NumberParameters << std::endl;
16 }
```

```
1  int main()
2  {
3    __try
4    {
5      throw_exception_divide_by_zero();
6    }
7    __except (filter(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO, GetExceptionInformation()))
8    {
9      std::cout << "Caught \"divide by zero\" exception\n" << std::endl;
10     show_info();
11   }
12   std::cout << "\n\n\n" << std::endl;
13   __try
14   {
15     throw_exception_noncontinuable_exception();
16   }
17   __except (filter(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION, GetExceptionInformation()))
18   {
19     std::cout << "Caught \"noncontinuable exception\"\n" << std::endl;
20     show_info();
21   }
22   _getch();
23   return 0;
24 }
```

В результате выполнения программы получаем:

```
Caught "divide by zero" exception

code: 3221225620
flags: 0
address: 00007FF62D12281C
record: 0000000000000000
NumberParameters: 0



Caught "noncontinuable exception" exception

code: 3221225509
flags: 0
address: 00007FF8B9FCA388
record: 0000000000000000
NumberParameters: 0
```

Рис. 3: Вывод программы

3.5 Программная генерация исключения RaiseException

Для программной генерации исключений можно использовать функцию RaiseException(). Данная функция принимает 4 аргумента:

- dwExceptionCode - код исключения
- dwExceptionFlags - флаг возобновляемого исключения
- nNumberOfArguments - количество аргументов
- lpArgumens - массив аргументов

Листинг 4: Прототип RaiseException

```
WINBASEAPI
__analysis_noreturn
VOID
WINAPI
RaiseException(
    _In_ DWORD dwExceptionCode,
    _In_ DWORD dwExceptionFlags,
    _In_ DWORD nNumberOfArguments,
    _In_reads_opt_(nNumberOfArguments) CONST ULONG_PTR* lpArguments
    );
```

```
LONG filter(EXCEPTION_POINTERS* exceptionPointers) {
  er = *(exceptionPointers->ExceptionRecord);
  return EXCEPTION_EXECUTE_HANDLER;
}
```

```
int main()
{
  int a = 3;
  int b = 5;
  DWORD Arguments[2];
  __try
  {
    Arguments[0] = a;
    Arguments[1] = b;
    RaiseException(0xFF, 0, 2, Arguments);
  }
  __except (filter(GetExceptionInformation()))
  {
    std::cout << "Exception code: " << std::hex << er.ExceptionCode << std::endl;
    std::cout << "Number parameters: " << er.NumberParameters << std::endl;
    std::cout << "1 parameter: " << std::dec << er.ExceptionInformation[0] << std::endl;
    std::cout << "2 parameter: " << er.ExceptionInformation[1] << std::endl;
  }
  _getch();
  return 0;
}
```

В результате программы получаем:

```
Exception code: ff
Number parameters: 2
1 parameter: 3
2 parameter: 5
```

Рис. 4: Вывод программы

3.6 Обработка необработанных исключений

Если в программе произошло исключение, для которого не существует обработчика исключений, то в этом случае вызывается функция-фильтр системного обработчика исключений, которая выводит на экран окно сообщений с предложением пользователю закончить программу аварийно или выполнить отладку приложения. Системная функция-фильтр UnhandledExceptionFilter имеет следующий прототип:

Листинг 5: Сигнатура UnhandledExceptionFilter

```
1 LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

Эта функция имеет один параметр, который указывает на структуру с типом exception_info и возвращает одно из следующих значений:

- EXCEPTION_CONTINUE_SEARCH - передать управление отладчику приложения
- EXCEPTION_EXECUTE_HANDLER - передать управление обработчику исключений

Приложение может заменить системную функцию-фильтр с помощью функции SetUnhandledExceptionFilter, которая имеет следующий прототип:

Листинг 6: Сигнатуры SetUnhandledExceptionFilter

```
1 WINBASEAPI
2 LPTOP_LEVEL_EXCEPTION_FILTER
3 WINAPI
4 SetUnhandledExceptionFilter(_In_opt_ LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter);
```

Эта функция возвращает адрес старой функции фильтра или NULL, если установлен системный обработчик исключений. Единственным параметром этой функции является указатель на новую функцию-фильтр, которая будет установлена вместо системной. Эта функция-фильтр должна иметь прототип, соответствующий системной функции фильтра UnhandledExceptionFilter, и возвращать одно из следующих значений:

- EXCEPTION_EXECUTE_HANDLER - выполнение программы прекращается
- EXCEPTION_CONTINUE_EXECUTION - возобновить исполнение программы с точки исключения
- EXCEPTION_CONTINUE_SEARCH - выполняется системная функция UnhandledExceptionFilter

```
1 LONG filterWithPrint(PEXCEPTION_POINTERS exceptionPointers)
2 {
3   er = *(exceptionPointers->ExceptionRecord);
4   show_info();
5   system("pause");
6   return EXCEPTION_EXECUTE_HANDLER;
7 }
```

```
1 int main()
2 {
3   auto old_filter = SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)filterWithPrint);
4   throw_exception_noncontinuable_exception();
5   std::cout << "After exception" << std::endl;
6   system("pause");
7   return 0;
8 }
```

В результате выполнения программы:

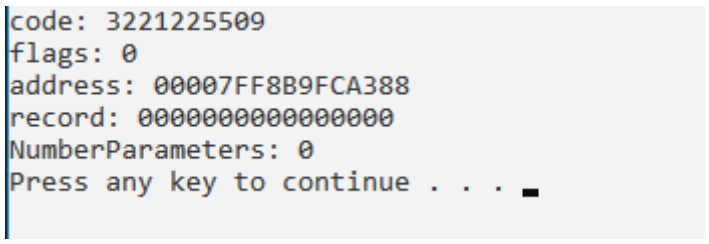

Рис. 5: Вывод программы

Продемонстрируем работу Proccess Monitor и покажем вызовы на стеке. Запустив утилиту, будет выдано большое количество процессов. Выбрав нашу программу и добавив её в основное дерево процессов утилиты, можно увидеть стек вызовов в окне <Stack Summary>:

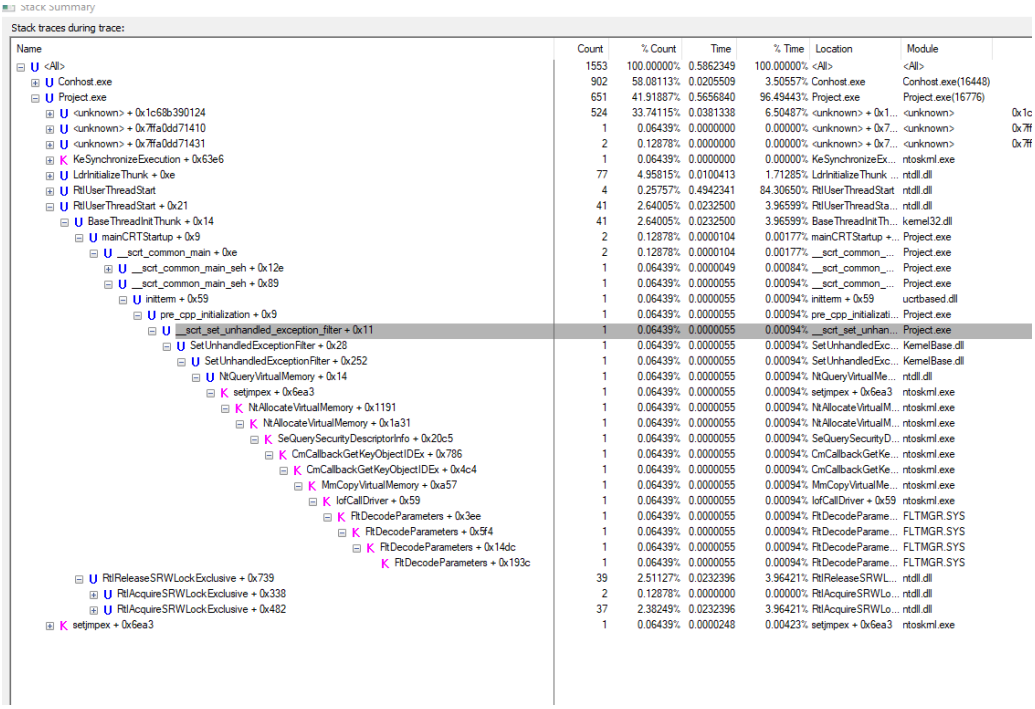

Рис. 6: Стек вызовов

Также продемонстрируем работу отладчика OllyDbg. С помощью данного отладчика можно проанализировать работу программы. Можно пошагово исполнить программу, поставить точки остановки и отслеживать всю интересующую информацию.

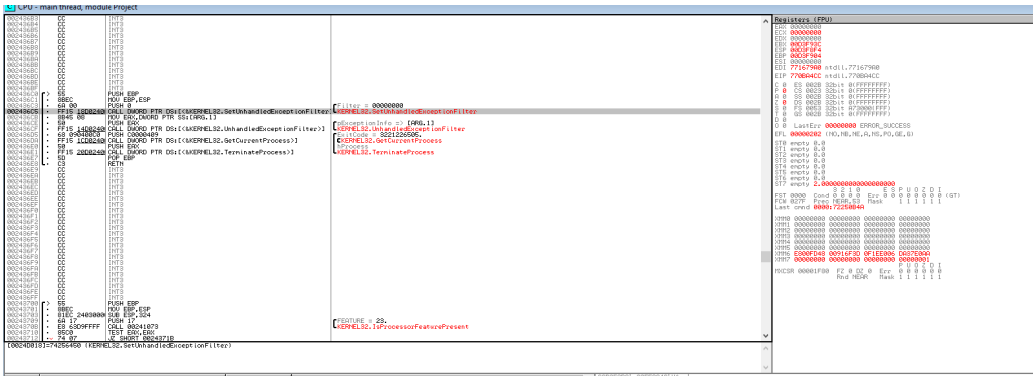

Рис. 7: OllyDbg

3.7 Вложенная обработка исключений

При использовании структурной обработки исключений возможно вкладывать блоки __try и __except в другой блок __try. В этом случае если функция-фильтр внутреннего блока __except возвращает значение EXCEPTION_CONTINUE_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам __try и __except, и продолжает поиск обработчика исключений во внешних блоках __try и __except. Так как локальные объекты, определенные внутри любого блока, хранятся в стеке процесса, то фактически система очищает стек процесса. Область стека, которую занимают локальные объекты одного блока, называется фреймом стека. Поэтому можно сказать, то при обработке вложенных исключений выполняется очистка стека процесса от локальных объектов, определенных внутри вложенных блоков __try и __except. Такая очистка стека от локальных объектов называется глобальной раскруткой стека или просто раскруткой стека.

```
1 int main()
```

```
{
  __try
  {
    __try
    {
      throw_exception_divide_by_zero();
    }
    __except (filter(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION))
    {
      std::cout << "Caught \"noncontinuable exception\"" << std::endl;
    }
  }
  __except (filter(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO))
  {
    std::cout << "Caught \"divide by zero\" exception" << std::endl;
  }
  system("pause");
  return 0;
}
```

В результате выполнения программы:

```
Caught "divide by zero" exception
Press any key to continue . . .
```

Рис. 8: Вывод программы

3.8 Завершение блока __try, используя goto и __leave

Для передачи управления из фрейма можно использовать инструкцию goto из C++. В этом случае система считает, что блок с охраняемым кодом завершился аварийно и поэтому выполняет глобальное раскручивание стека. Следовательно, использование инструкции goto вызывает исполнение дополнительного программного кода, что замедляет выполнение программы.

Программа, использующая goto для выхода из блока __try.

```
int main()
{
  __try
  {
    goto label;
    throw_exception_divide_by_zero();
  }
  __finally
  {
    printf("__finally \n");
  }
label:
  system("pause");
  return 0;
}
```

Её ассемблерный код приведен ниже. Из этого кода видно, что на месте вызова goto, стоит вызов функции _local_unwind, которая выполняет раскрутку стека и переводит исполнение программы к маркеру $label$12.

Листинг 7: Скомпилированный ассемблерный код

```
$T1 = 32
argc$ = 64
argv$ = 72
main    PROC
$LN11:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 56                                 ; 00000038H
        mov     QWORD PTR $T1[rsp], rsp
        lea     rdx, $label$12
        mov     rcx, QWORD PTR $T1[rsp]
        call    _local_unwind
        call    int throw_exception_divide_by_zero(void)        ; throw_exception_divide_by_zero()
        npad    1
$LN9@main:
        lea     rcx, OFFSET FLAT:$SG27156
        call    printf
$label$12:
        xor     eax, eax
$LN5@main:
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
```

Если необходимо просто завершить выполнение блока __try без аварийного выхода, то есть не начиная раскрутку стека, то в этом случае нужно использовать инструкцию __leave. Программа, использующая __leave для выхода из блока __try.

```
int main()
{
  __try
  {
    __leave;
    throw_exception_divide_by_zero();
  }
  __finally
  {
    printf("__finally \n");
  }
  system("pause");
  return 0;
}
```

Её ассемблерный код приведен ниже. Из этого кода видно, что на месте вызова __leave, стоит вызов инструкции jmp, которая выполняет раскрутку стека и переводит исполнение программы к маркеру $LN2@main.

Листинг 8: Скомпилированный ассемблерный код

```
argc$ = 48
argv$ = 56
main    PROC
$LN10:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 40                                 ; 00000028H
        jmp     SHORT $LN2@main
        call    int throw_exception_divide_by_zero(void)        ; throw_exception_divide_by_zero
        npad    1
$LN2@main:
$LN8@main:
        lea     rcx, OFFSET FLAT:$SG27154
        call    printf
        xor     eax, eax
$LN4@main:
        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

3.9 Преобразование структурное исключение в исключение языка C

В реализации C++ предусмотрен механизм, который позволяет использовать механизм структурной обработки исключений в механизме обработки исключений, используемом в C++. Для этой цели была разработана функция _set_se_transiator. Эта функция устанавливает в системе функцию, которая называется функцией-транслятором, назначение которой состоит в том, чтобы преобразовывать структурные исключения в исключения C++. Если функция-транслятор установлена, то она вызывается всегда при выбросе структурного исключения. В функции-трансляторе можно использовать инструкцию throw, которая будет выбрасывать исключение нужного типа. Функция-транслятор должна иметь следующий прототип:

```
typedef void (*_se_translator_function) (unsigned int, struct _EXCEPTION_POINTERS*);
```

Прототип описан в заголовочном файле eh.h. Как видно из этого описания — функция-транслятор не возвращает значения и получает два параметра: код исключения и указатель на структуру типа EXCEPTION_POINTERS. Функция _set_se_transiator, которая используется для установки функции-транслятора, также описана в заголовочном файле eh.h и имеет следующий прототип:

```
_se_translator_function _set_se_translator(_se_translator_function se_trans_func);
```

Единственным параметром этой функции является указатель на новую функцию-транслятор, а возвращает функция _set_se_transiator адрес старой функции-транслятора, которая в дальнейшем может быть восстановлена при помощи вызова _set_se_transiator. Если функция-транслятор устанавливается в первый раз, то возвращаемое значение может быть равно NULL. Рассмотрим пример:

```
void se_trans_func(unsigned code, EXCEPTION_POINTERS *)
{
  throw code;
}
```

```
int main(int argc, char **argv)
{
  _set_se_translator(se_trans_func);
  try
  {
    throw_exception_divide_by_zero();
  }
  catch (unsigned code)
  {
    handle_exception(code);
  }
  try
  {
    throw_exception_noncontinuable_exception();
  }
  catch (unsigned code)
  {
    handle_exception(code);
  }
  return 0;
}
```

Данная программа генерирует заданные исключения. Функция-транслятор просто возвращает нам код возникшего исключения. В зависимости от полученного кода обработчик выводит нам соответствующее сообщение. Следует заметить, что для использования функции _set_se_transiator необходимо передать компилятору флаг /EHa, который разрешает работу с C++ исключениями.

В результате выполнения программы:

```
Caught exception with code = 3221225620
EXCEPTION_INT_DIVIDE_BY_ZERO
Caught exception with code = 3221225509
EXCEPTION_NONCONTINUABLE_EXCEPTION
Press any key to continue . . .
```

Рис. 9: Вывод программы

Также в функции-трансляторе можно возвращать более детальную информацию. Например – структуру EXCEPTION_RECORD:

```
void se_trans_func(unsigned code, EXCEPTION_POINTERS *info)
{
  throw *(info->ExceptionRecord);
}
```

```
int main()
{
  _set_se_translator(se_trans_func);
  try
  {
    throw_exception_divide_by_zero();
  }
  catch (EXCEPTION_RECORD record)
  {
    handle_exception(record.ExceptionCode);
  }
  try
  {
    throw_exception_noncontinuable_exception();
  }
  catch (EXCEPTION_RECORD record)
  {
    handle_exception(record.ExceptionCode);
  }
  return 0;
}
```

В результате выполнения программы:

```
Caught exception with code = 3221225620
EXCEPTION_INT_DIVIDE_BY_ZERO
Caught exception with code = 3221225509
EXCEPTION_NONCONTINUABLE_EXCEPTION
Press any key to continue . . .
```

Рис. 10: Вывод программы

3.10 Использование __try – __finally блока

В Windows существует еще один способ обработки исключений - код, при исполнении которого возможен выбрасывание исключения, как и в случае с фреймовой обработкой исключений, заключается в блок __try. Но только теперь за блоком __try следует код, который заключается в блок __finally. Система гарантирует, что при любой передаче управления из блока __try, независимо от того, произошло или нет исключение внутри этого блока, предварительно управление будет передано блоку __finally. Такой способ обработки исключений называется финальная обработка исключений. Структурно финальная обработка исключений выглядит следующим образом:

```
    __try
    {
        // guarded code
    }
    __finally
    {
        // finally code
    }
```

Финализированная обработка исключений используется для того, чтобы при любом исходе исполнения блока __try освободить ресурсы, которые были захвачены внутри этого блока. Такими ресурсами могут быть память, файлы, критические секции и т.д.

```
int main()
{
  __try
  {
    __try
    {
      __try
      {
        throw_exception_divide_by_zero();
      }
      __finally
      {
        std::cout << "First finally" << std::endl;
      }
    }
    __except (filter(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO))
    {
      std::cout << "Caught exception: EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
    }
    __try
    {
      __try
      {
         throw_exception_noncontinuable_exception();
      }
      __finally
      {
        std::cout << "Second finally" << std::endl;
      }
    }
    __except (filter(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION))
    {
      std::cout << "Caught exception: EXCEPTION_NONCONTINUABLE_EXCEPTION" << std::endl;
    }
  }
  __finally
  {
    std::cout << "Main Finally" << std::endl;
  }
  return 0;
}
```

По выводу программы можно понять, что код из блока __finally вызывается не только в случае ошибки, но и если блок завершился корректно.

В результате выполнения программы:

```
First finally
Caught exception: EXCEPTION_INT_DIVIDE_BY_ZERO
Second finally
Caught exception: EXCEPTION_NONCONTINUABLE_EXCEPTION
Main Finally
Press any key to continue . . .
```

Рис. 11: Вывод программы

Управление из блока __try может быть передано одним из следующих способов:

1. нормальное завершение блока
2. выход из блока при помощи управляющей инструкции __leave
3. выход из блока при помощи одной из управляющих инструкций return
4. break, continue или goto C++
5. передача управления обработчику исключения

В первых двух случаях считается, что блок __try завершился нормально, а во вторых двух случаях — аварийно. Для того чтобы определить, как завершился блок __try, используется функция AbnormaiTermination, которая имеет следующий прототип:

```
BOOL AbnormaiTermination (VOID);
```

В случае если блок __try завершился аварийно, эта функция возвращает ненулевое значение, а в противном случае — значение false.

```
int main(int argc, char **argv) {
  __try
  {
    __try
    {
      __try
      {
        throw_exception_divide_by_zero();
      }
      __finally
      {
        std::cout << "First finally" << std::endl;
        if (AbnormalTermination())
        {
          std::cout << "Try teminated (first)" << std::endl;
        }
      }
    }
    __except (filter(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO))
    {
      std::cout << "Caught exception: EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
    }
    __try
    {
      __try
      {
        throw_exception_noncontinuable_exception();
      }
      __finally
      {
        std::cout << "Second finally" << std::endl;
        if (AbnormalTermination())
        {
          std::cout << "Try teminated (second)" << std::endl;
        }
      }
    }
    __except (filter(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION))
    {
      std::cout << "Caught exception: EXCEPTION_NONCONTINUABLE_EXCEPTION" << std::endl;
    }
  }
```

```
43      __finally
44      {
45        std::cout << "Main finally" << std::endl;
46        if (AbnormalTermination())
47        {
48          std::cout << "Try teminated (Main)" << std::endl;
49        }
50      }
51      system("pause");
52      return 0;
53  }
```

Из вывода программы видно, что в случае корректного завершения блока __try, функция AbnormalTermination возвращает false.

```
First finally
Try teminated (first)
Caught exception: EXCEPTION_INT_DIVIDE_BY_ZERO
Second finally
Try teminated (second)
Caught exception: EXCEPTION_DATATYPE_MISALIGNMENT
Main finally
Press any key to continue . . .
```

Рис. 12: Вывод программы

4 Вывод

В ходе работы были изучены структурные исключения SEH. Механизм структурной обработки исключений в Windows немного отличается от механизма обработки исключений, принятого в языке программирования C++. Дело в том, что механизм структурной обработки исключений был разработан раньше, чем принят стандарт языка C++. Кроме того, в отличие от языка программирования C++ механизм структурной обработки исключений ориентирован не только на обработку программных исключений, но и на обработку аппаратных исключений. В SEH исключение рассматривается как ошибка, происшедшая при выполнении программы. В языке программирования C++ используется более абстрактный подход и исключение рассматривается как объект произвольного типа, который может выбросить программа, используя оператор throw. В свою очередь обработчик исключения catch может рассматриваться как функция с одним параметром, которая выполняется только в том случае, если тип ее параметра соответствует типу выброшенного исключения.

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.

OllyDbg – бесплатный проприетарный отладчик уровня ассемблера для операционных систем Windows, предназначенный для анализа и модификации откомпилированных исполняемых файлов и библиотек.

OllyDbg выгодно отличается от классических отладчиков (таких, как SoftICE) интуитивно понятным интерфейсом, подсветкой специфических структур кода, простотой в установке и запуске.

WinDBG – это многоцелевой отладчик для операционной системы Windows. Его можно использовать для отладки приложений пользовательского режима, драйверов устройств и самой операционной системы в режиме ядра. Как и более известный отладчик Visual Studio, он имеет графический пользовательский интерфейс. Все функциональности, которые представлены в данном инструменте довольно удобные для использования.

Proccess Monitor — утилита, которая имеет возможности программы мониторинга обращений к реестру Regmon и программы мониторинга обращений к файловой системе Filemon, и дополнительно, позволяет получать более подробную информацию о взаимодействии процессов, использовании ресурсов, сетевой активности и операциях ввода-вывода.

Process Monitor выполняется наблюдение в реальном времени для следующих классов событий:

- Файловая система: создание(открытие)/закрытие/чтение/запись/удаление элементов файловой системы: файлов, каталогов, атрибутов, содержимого.
- Реестр: создание/чтение/запись/перечисление/удаление элементов реестра: ветвей, ключей, значений.
- Сеть: установка соединения, передача данных, закрытие соединения. Информация об источнике/-приемнике TCP/UDP трафика. Общая информация о протоколах, пакетах. Сами передаваемые данные не записываются.
- Процесс/поток: Создание процесса, создание потока внутри процесса, завершение потока/процесса. детальная информация о процессе (путь, командная строка, ID пользователя/сессии), запуск/завершение, загрузка образов (библиотеки/драйвера), стек выполнения.
- Профилирование: Специальный класс событий, записываемых с целью отслеживания количества процессорного времени, затрачиваемого каждым процессом. Использование памяти процесса.

Для своей работы, Process Monitor устанавливает в системе собственный драйвер PROCMON20.SYS , с помощью которого выполняется перехват контролируемых монитором системных функций и сбор данных подлежащих мониторингу. Наблюдение выполняется для следующих классов операций - обращения к файловой системе (file system), обращение к реестру (Registry), работа с сетью (Network), и активность процессов (Process).