Unit 8: File System

8.5. Win32 Structured Exception Handling

Win32 Structured Exception Handling

- Robust mechanism to respond to unexpected events
  - Addressing exceptions
  - Arithmetic faults
  - System errors
- Exit from anywhere in a code block
  - Perform programmer-specified processing
  - Free resources / perform cleanup functions
- Simplify program logics
- Detect external signals: console control handler

Exceptions and their Handlers

SEH is supported:

- Through Win32 functions,
- Language support by compiler: _try, _except
- Run-time support: filter_expression

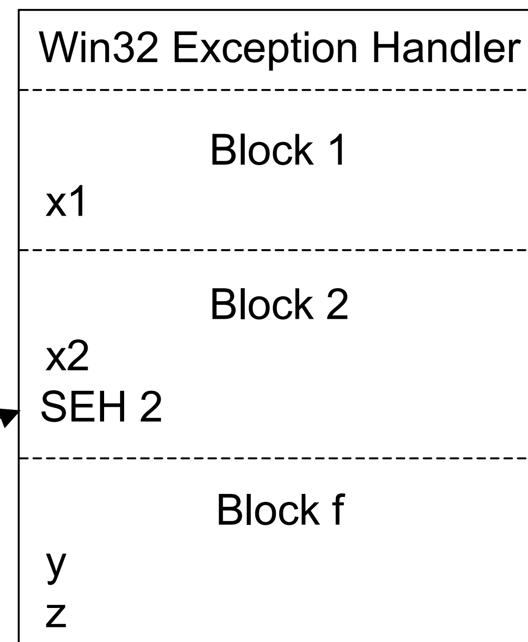
Try and Except Blocks

```
__try {
        /* Block of monitored code */
}
__except (filter_expression) {
        /* Exception handling block */
}
```

SEH, Blocks, and Functions

```
{ DWORD x1; /* Block 1 */
                ...
  __try { /* Block 2 */
     DWORD x2;
                ...
     x2 = f(x1);
                ...
  }
  __except () {
        /* SEH 2 */
  }
}
DWORD f (DWORD y) {
    /* Block f */
   DWORD y;
   z = y / (y – 1);
   return z / y;
}
```

Exception occurs
Execute this SEH



Filter Expressions and Their Values

- Filter_expression is evaluated immediately after exception occurs
  - Usually a literal constant or a filter function which performs analysis
- Expression should return:
  1. EXCEPTION_EXECUTE_HANDLER – system executes except block
  2. EXCEPTION_CONTINUE_SEARCH – system ignores exception handler and searches for an exception handler in enclosing block until it finds a handler
  3. EXCEPTION_CONTINUE_EXECUTION – system returns control to point where exception occurred.
     Another exception might be raised immediately.

Example: delete a temporary file

```
GetTempFileName( TempFile, ...);
while (...) __try {
    hFile = CreateFile( TempfFile, ... );
    ...
    i = *p; /* p == NULL; exception occurs */
    ...
}
__except( EXCEPTION_EXECUTE_HANDLER ) {
    CloseHandle( hFile );
    DeleteFile( TempFile );
}
/*  control passes here in case of either normal loop termination
    or an exception */
```

Exception Codes

- Exception code must be obtained immediately
  - Can be called only from within filter_expression
- Categories of exception codes:
  - Program violations
    - EXCEPTION_ACCESS_VIOLATION, EXCEPTION_DATATYPE_MISALIGNMENT, EXCEPTION_NONCONTINUABLE_EXECUTION
  - Exceptions raised by memory allocation functions
    - HeapAlloc(), HeapCreate() with HEAP_GENERATE_EXCEPTIONS flag set
  - User-defined exception created via RaiseException()
  - Variety of arithmetic exceptions:
    - EXCEPTION_INT_DIVIDE_BY_ZERO, EXCEPTION_FLT_OVERFLOW
  - Exceptions used by debugger:
    - EXCEPTION_BREAKPOINT, EXCEPTION_SINGLE_STEP

Get Exception Codes via:

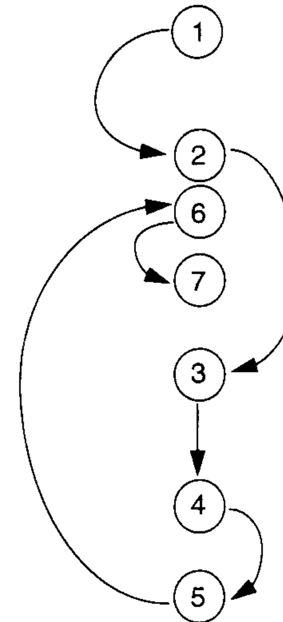
```
DWORD                       GetExceptionCode( VOID );
LPEXCEPTION_POINTERS        GetExceptionInformation( VOID );
```

```
typedef struct _EXCEPTION_POINTERS {
        PEXCEPTION_RECORS ExceptionRecord;
        PCONTEXT ContextRecord
} EXCEPTION_POINTERS;
```

- Part of this information is processor-specific (ContextRecord – see winnt.h)
- Data members of ExceptionRecord contain:
  - Virtual memory address, ExceptionAddress
  - Parameter array

Exception Handling Sequence

```
_try {
        . . .
        i = j / 0;
        . . .
}
_except (Filter (GetExceptionCode ())) {
        . . .
}
        . . .
DWORD Filter (DWORD ExCode)
{
        switch (ExCode) {
        . . .
        case EXCEPTION_INT_DIVIDE_BY_ZERO:
        . . .
        return EXCEPTION_EXECUTE_HANDLER;
        case . . .
        }
}
```



Floating-Point Exceptions

```
DWORD _controlfp( DWORD new, DWORD mask );

Actual mask: (currrent_mask & ~mask) | (new & mask)
```

Normally disabled – turn mask bit off to enable exception

- Specific exceptions for:
  - Underflow, overflow, div-by-zero, inexact result, de-normalized op, invalid op

```
/* Save old control mask. */
FPOld = _controlfp (0, 0);
                    /* Enable floating-point exceptions. */
FPNew = FPOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT
             | EM_ZERODIVIDE | EM_DENORMAL | EM_INVALID);
                    /* Set new control mask.
                       MCW_EM combines all six exceptions */
_controlfp (FPNew, MCW_EM);
```

User-Generated Exceptions

```
VOID RaiseException( DWORD dwExceptionCode,
          DWORD dwExceptionFlags,
          DWORD cArguments,
          LPDWORD lpArguments );
```

- dwExceptionCode is the user-defined code
- Bit 28 is reserved for system; Error code is encoded in bits 27-0
- Bits 31-30 encode:
  - 0 – success
  - 1 – informational
  - 2 – warning
  - 3 – error

Exceptions cannot be raised in another process – Use console control handlers + GenerateConsoleCtrlEvent() instead

- dwExceptionFlags == EXCEPTION_NONCONTINUABLE: no continuation after filter expression allowed
- lpArguments points to array of size cArguments – to be passed to filter_expression (accessible via GetExceptionInformation())

Comparison to UNIX signals

- UNIX signal model is different from SEH.
  Signals can be missed or ignored; flow is different.
- UNIX signals are largely supported through C library;
  console control handlers can be used in place of signals
- Signal-to-exception correspondence:
  - SIGILL – EXCEPTION_PRIV_INSTRUCTION
  - SIGSEGV – EXCEPTION_ACCESS_VIOLATION
  - SIGFPE – seven distinct FP exceptions, such as EXCEPTION_FLT_DIVIDE_BY_ZERO
  - SIGUSR1, SIGUSR2 – user-defined exceptions
- Win NT will not generate SIGILL, SIGSEGV, SIGTERM
  Win32 does not support SIGINT
- Kill() – comparable to GenerateConsoleCtrlEvent()
- Kill (pid, SIGKILL) – comparable to TerminateProcess(pHandle);

Termination Handler

```
__finally {
        /* block – cleanup */
}
```

- Termination handler is executed whenever control flow leaves __try-block
  - Reaches the end of the block
  - Execution of: return, break, goto, continue, longjmp, __leave
  - Exception
- Termination handlers are not executed on thread/process termination (!)
- Abnormal termination: everything, except __leave statement or reaching end of __try-block; check with:

```
BOOL AbnormalTermination (VOID)
```

Console Control Handlers

- Detect log-off, ctrl-c, ctrl-break

```
BOOL SetConsoleCtrlHandler(
        PHANDLER_ROUTINE HandlerRoutine,
        BOOL Add);

BOOL HandlerRoutine( DWORD dwCtrlType );
```

- Possible control types:
  1. CTRL_C_EVENT
  2. CTRL_CLOSE_EVENT (console window is being closed)
  3. CTRL_BREAK_EVENT
  4. CTRL_LOGOFF_EVENT (user is logging off)
  5. CTRL_SHUTDOWN_EVENT (system is shutting down)
- Next handler in list is called or process is terminated if signal handler returns false