Tratarea excepțiilor

CURS NR. 8/2

SEH – Structured Exception Handling

SEH este un mecanism de tratare a excepțiilor (și tratarea terminării)

- Permite aplicațiilor să răspundă la evenimente asincrone neașteptate
  - Ex. erori aritmetice, erori de sistem, excepții de adresare, etc.
- Oriunde ar fi fluxul de execuție în aplicație, la recepționarea unei excepții se va efectua automat procesarea specificată de programator înainte de terminare
- Permite eliberarea resurselor înainte de terminarea threadului sau procesului
  - Terminare din cauze controlate sau în urma apariției unei excepții
- Poate fi adăugat cu ușurință la codul existent
  - Simplifică logica programului

Excepțiile pot fi inițiate de software sau hardware, în mod kernel sau mod utilizator

- indică de regulă o eroare fatală, fără recuperare – threadul sau chiar întregul proces trebuie să se termine
- Pot fi excepții din care programul se poate recupera și își poate continua execuția
- Exemple de excepții: dereferențierea unui pointer NULL, împărțirea la zero, acces la zona de memorie la care nu are drepturi etc.

Fără SEH excepțiile neașteptate ar duce la terminarea imediată a programului fără a fi efectuate procesările normale de terminare – eliberarea resurselor, ștergerea fișierelor temporare, etc.

Tratarea excepțiilor: blocuri `try - except`

SEH permite specificarea unui bloc de cod care efectuează tratarea excepției – handler-ul excepției
- permite realizarea procesărilor dorite de programator, înainte de terminare

Primul pas
- Alegerea blocului de cod care trebuie monitorizat
- Scrierea codului de tratare a excepțiilor – handlerul excepției

  - Blocuri candidate
    - Pot apare erori detectabile (incluzând erori la apeluri sistem) din care se dorește recuperarea din eroare decât terminarea programului
    - Există posibilitatea dereferențierii unor pointeri care posibil nu au fost inițializați
    - La gestiunea tablourilor se poate accesa elemente în afara limitelor
    - Codul efectuează aritmetică cu valori flotante și pot apare împărțiri la zero, rezultate imprecise sau overflow
    - Codul apelează funcții care pot genera excepții din cauza argumentelor incorecte sau din alte motive

- Blocul monitorizat este inclus într-un cadru try - except

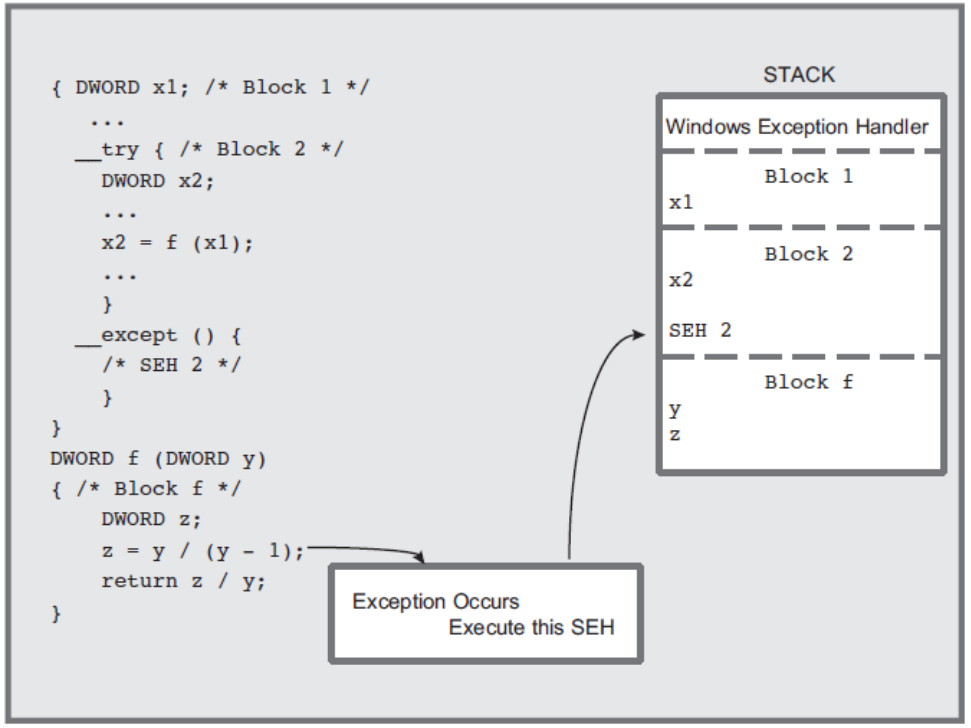
```
__try {
    /* Blocul monitorizat */
}

__except (expresie_de_filtrare) {
    /* Blocul de tratare a exceptiei */
}
```

Bloc de cod din aplicația normală
- Dacă apare o excepție în acest bloc SO transferă controlul blocului de tratare a excepției

Expresia de filtrare este evaluată de sistem imediat după apariția excepției. Acesta determină acțiunile care urmează.

EXCEPTION_EXECUTE_HANDLER (controlul se transferă handler-ului de excepție)
EXCEPTION_CONTINUE_SEARCH (sistemul continuă să caute alte handler-uri de tratare)
EXCEPTION_CONTINUE_EXECUTION (sistemul oprește căutarea pentru un bloc de tratare și returnează controlul în punctul în care s-a generat excepția. Dacă excepția este necontinuabilă, se generează **EXCEPTION_NONCONTINUABLE_EXCEPTION**)

Tratarea excepțiilor: blocuri try - except



SEH, Blocks, and Functions

Blocul monitorizat poate fi un bloc de cod în care se apelează altă funcție

- Excepția poate apare în codul funcției apelate
  - Handler-ul de tratare a excepției este regăsit pe stivă

Exemplu: verificare dacă apare împărțirea la zero

```
BOOL SafeDiv(INT32 dividend, INT32 divisor, INT32 *pResult) {
   __try
   {
      *pResult = dividend / divisor;
   }
   __except (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
   {
      return FALSE;
   }
   return TRUE;
}
```

Codul excepției

Expresia de filtrare poate obține codul excepției imediat după apariția acesteia

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

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

Alternativa – apelabilă doar din expresia de filtrare

- Informații adiționale

- Expresia de filtrare poate trimite codul ca și parametru funcției de filtrare
  - Funcția de filtrarea nu poate apela funcția `GetExceptionCode` (restricție impusă de compilator)

```
__except (MyFilter(GetExceptionCode())) {
/* ..................*/
}
```

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

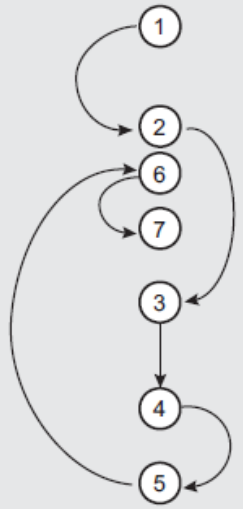


Figure 4–2 Exception Handling Sequence

Codul excepției

Codurile returnate de `GetExceptionCode` pot fi diverse (vezi lista),
și pot fi clasificate în mai multe categorii

- Abateri în execuția normală a programului
  - **EXCEPTION_ACCESS_VIOLATION** - tentativă de accesare a unei zone de memorie la care nu are drepturi de acces
  - **EXCEPTION_DATATYPE_MISALIGNMENT**
  - **EXCEPTION_NONCONTINUABLE_EXCEPTION** - expresia de filtrare indică continuare **EXCEPTION_CONTINUE_EXECUTION** dar continuarea nu este posibilă după această excepție
- Excepții generate de funcțiile de alocare a memoriei – HeapAlloc și HeapCreate – dacă folosesc flagul **HEAP_GENERATE_EXCEPTIONS** și codul poate fi
  - **EXCEPTION_ACCESS_VIOLATION** sau **STATUS_NO_MEMORY**
- Cod definit de utilizator - excepții generate de funcția RaiseException
- Coduri generate de excepții aritmetice
  - Exemple: **EXCEPTION_INT_DIVIDE_BY_ZERO** și **EXCEPTION_FLT_OVERFLOW**
- Excepții generate de debugger, cum ar fi **EXCEPTION_BREAKPOINT** sau **EXCEPTION_SINGLE_STEP**

Excepții generate de utilizator

Programul poate genera excepții – astfel programul poate detecta erori și le poate trata ca și excepții

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

Codul definit de aplicație. Bitul 28 este rezervat. Codul de eroare trebuie codificat în biții 0-27

Flaguri: 0 (excepție continuabilă), EXCEPTION_NONCONTINUABLE (excepție necontinuabilă)

NULL sau argumentele de trimis expresiei de filtrare
Și numărul argumentelor (al treilea parametru)

Exemplu

```
VOID ReportException (LPCTSTR userMessage, DWORD exceptionCode)

/* Report as a non-fatal error.
Print the system error message only if the message is non-null. */
{
   if (lstrlen (userMessage) > 0)
      ReportError (userMessage, 0, TRUE);

   /* If fatal, raise an exception. */
   if (exceptionCode != 0)
      RaiseException ((0x0FFFFFFF & exceptionCode) | 0xE0000000, 0, 0, NULL);

   return;
}
```

Tratarea terminării: blocuri try - finally

Handler-ul de terminare este similar cu handler-ul de excepție

- Însă este garantată execuția când threadul părăsește blocul de cod atât ca urmare a **terminării normale** cât și datorită apariției unei **excepții**
  - Excepție la apelul `ExitProcess`, `ExitThread`, `TerminateProcess`, `TerminateThread`
  - Nu poate diagnostica excepțiile – deci nu există expresie de filtrare
- Permite eliberarea resurselor, închiderea handle-urilor, etc.
  - Programul poate apela return din mijlocul unui bloc de cod, iar handlerul de terminare se ocupă de rânduirea resurselor
    - Nu este necesară includerea codului pentru procesarea terminării imediat înainte de return – logica programului este mai clară

```
__try {
   /* Blocul monitorizat */
}
__finally  {
   /* Handler de terminare */
}
```

Combinarea blocurilor finally și except

- Un bloc __try poate avea un singur bloc finally sau except – nu ambele!!

```
__try {
/* Blocul monitorizat */
}

__except (expresie_de_filtrare) {
/* Blocul de tratare a exceptiei */
}

__finally  {
/* Handler de terminare */
}
```

Soluție: imbricarea blocurilor

- Tehnică des folosită

```
DWORD FilterFunction() {
   printf("1 ");                    // printed first
   return EXCEPTION_EXECUTE_HANDLER;
}

INT main(VOID) {
   __try {
      __try {
         RaiseException(
                  1,        // exception code
                  0,        // continuable exception
                  0, NULL); // no arguments
      }
      __finally {
         printf("2 ");     // this is printed second
      }
   }
   __except (FilterFunction()) {
   printf("3\n");                // this is printed last
   }
}
```

Exemplu: tratarea terminării

```
DWORD Funcenstein1()
{
   DWORD dwTemp;
   // 1. Do any processing here.
   ...
   __try {
      // 2. Request permission to access
      // protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);
      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;
   }
   __finally {
   // 3. Allow others to use protected data.
   ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // 4. Continue processing.
   return(dwTemp);
}
```

- Numerotarea comentariilor indică și ordinea în care se execută secțiunile de cod
- Exemplul efectuează
  - o așteptare simplă după un semafor,
  - modifică valoarea datei partajate,
  - salvează valoarea nouă într-o variabilă locală
  - eliberează semaforul
  - returnează valoarea nouă funcției apelante
  - Se asigură că semaforul este eliberat de fiecare dată la părăsirea blocului de instrucțiuni

Exemplu: ieșire prematură

```
DWORD Funcenstein2() {
   DWORD dwTemp;
   // 1. Do any processing here.
   ...
   __try {
      // 2. Request permission to access
      // protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);
      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;

      // Return the new value.
      return(dwTemp);
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // Continue processing--this code
   // will never execute in this version.
   dwTemp = 9;
   return(dwTemp);
}
```

- Adăugarea unei instrucțiuni `return` în blocul de cod protejat
  - Returnarea valorii noi a variabilei partajate
- Problema: nu se eliberează semaforul !! (fără finally)
  - Threadurile care așteaptă după acest semafor sunt înfometate
- Soluție: prin handler-ul de terminare se asigură că nu se revine în mod prematur din funcție
  - Înainte de revenire se execută codul handler-ului de terminare
    - → întotdeauna se eliberează semaforul înainte de revenire din funcție
- Codul de după handler-ul de terminare nu se execută niciodată
  - Ca urmare a instrucțiunii return în blocul de cod protejat
  - Deci se returnează valoarea 5 și nu valoarea 9

Local unwind
apare când sistemul execută blocul finally din cauza unei ieșiri premature din blocul try

Sistemul trebuie să genereze cod adițional – efect negativ asupra performanței programului → SĂ EVITĂM IEȘIREA PREMATURĂ !!

Exemplu: salt în afara blocului try

```
DWORD Funcenstein3() {
   DWORD dwTemp;
   // 1. Do any processing here.
   ...
   __try {
      // 2. Request permission to access
      // protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);
      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;
      // Try to jump over the finally block.

      goto ReturnValue;
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }
   dwTemp = 9;
   // 4. Continue processing.
   ReturnValue:
   return(dwTemp);
}
```

- Adăugarea unei instrucțiuni de salt `goto` în blocul de cod protejat
  - Salt în afara blocului try
- Se generează local unwind
  - Se execută mai întâi blocul handler-ului de terminare
- Codul de la eticheta ReturnValue se execută
  - Deoarece nu apare instrucțiune de ieșire în blocul try sau finally
  - Deci se returnează valoarea 5

Exemplu: necesitatea tratării terminării

```
DWORD Funcfurter1() {
   DWORD dwTemp;
   // 1. Do any processing here.
   ...
   __try {
      // 2. Request permission to access
      // protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);
      dwTemp = Funcinator(g_dwProtectedData);
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }
   // 4. Continue processing.
   return(dwTemp);
}
```

- **Dacă** funcția apelată în blocul try, `Funcinator`, conține instrucțiuni care provoacă eroare de acces invalid la memorie
  - Fără SEH → ”Application has stopped working” → se termină aplicația
    - Semaforul nu este eliberat
  - Cu SEH → blocul finally garantează eliberarea semaforului chiar dacă funcția apelată provoacă accesul invalid
  - Finally poate capta instrucțiuni simple gen `break`, `continue` și combinații `setjmp` și `longjmp`

Exemple - utilizarea __leave

```
DWORD Funcarama4() {
   // IMPORTANT: Initialize all variables to assume failure.
   HANDLE hFile = INVALID_HANDLE_VALUE;
   PVOID pvBuf = NULL;
   // Assume that the function will not execute successfully.
   BOOL bFunctionOk = FALSE;
   __try {
      DWORD dwNumBytesRead;
      BOOL bOk;
      hFile = CreateFile(TEXT("SOMEDATA.DAT"), GENERIC_READ,
                         FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
      if (hFile == INVALID_HANDLE_VALUE) {
         __leave;
      }
      pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
      if (pvBuf == NULL) {
         __leave;
      }
      bOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
      if (!bOk || (dwNumBytesRead == 0)) {
         __leave;
      }
      // Do some calculation on the data.
       ...
      // Indicate that the entire function executed successfully.
      bFunctionOk = TRUE;
   }
   __finally {
      // Clean up all the resources.
      if (pvBuf != NULL)
          VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
      if (hFile != INVALID_HANDLE_VALUE)
         CloseHandle(hFile);
   }
   // Continue processing.
   return(bFunctionOk);
}
```

Rolul cuvântului cheie `__leave` este de a face un salt la finalul blocului try

- La acolada de terminare a blocului `try`
- Fluxul de control trece natural la execuția blocului `finally`
  - Fără alte încărcări adiționale
  - Este necesară o variabilă booleană adițională
    - Arată dacă s-a terminat cu succes sau cu eșec blocul `try`
    - Indică în blocul `finally` ce resurse au fost alocate și trebuie eliberate
  - Alternativă: flaguri pentru alocarea cu succes a resurselor

Materiale de studiu

- Johnson HART, Windows System Programming, 4th edition, Addison Wesley, 2010
  - Capitolul 4
  - Exemplele incluse sunt din cartea de mai sus – vezi arhiva de pe moodle: WSP4_Examples.zip