CSE 438-598 Project Report
Topic - Exception Handling

Dipal Saluja
Nikhil Kulkarni
Abstract

With numerous exceptions being generated in a system for page faults, segmentation faults, user programming errors, even a switch from user mode to kernel mode; exception handling has been the epicenter of all event-based systems such as Linux. Exception handling has received enough attention it deserved and hence a robust exception handling mechanism has been one of main reasons for the stability of Linux kernel.

Compilers also play an important role in exception handling in providing support for programming languages. With the Object-oriented languages like C++, Java having such a robust runtime error management mechanism, it makes programmer’s life easy and build better applications with high reliability and fault tolerance.

1. Introduction

An exception is basically an unusual or anomalous condition occurring at runtime disrupting the normal flow of execution. The exception raised can be due to many conditions which include performing illegal operation, page faults, illegal access to memory, intentionally caused to generate a request etc. Before actually exception handling was used, programmers relied on error codes and status flags to check for anomalous conditions and this proved to be a very time consuming and error prone methodology. It becomes very important to address all such unusual conditions occurring in the system in a more efficient and effective way, which is what termed as “exception handling”. It should be noted that just the operating system or the kernel alone doesn't cause exceptions, there are also many user applications which will be performing numerous operations and there can be exceptions raised by these user level programs as well. Compilers include an exception handling mechanism so as to further classify exceptions in programming languages such as C++, java. Though exception handling forms to be an important mechanism his is because the Real Time systems are mostly time critical and running time of exception handlers is unpredictable.

We will be addressing the language support for exceptions and the exceptions at OS level in separate sections.

Section 2 will be addressing the language support for exceptions. We explain how C++ facilitates user space exception handling. We start with explaining the C++ exception handling semantics. Then we explain the C++ exception handling backend. Before getting into the details of the C++ exception handling backend, we will introduce a few concepts like Stack Frames, Stack Unwinding and Landing Pad. To explain the mechanism used by C++ we give a brief introduction to the Itanium C++ ABI. We give a short explanation of the base ABI in Section 2.4 and a detailed explanation of C++ ABI in Section 2.5.

Section 3 will be addressing Operating System support for exceptions where we will be addressing the types of exceptions in an OS. Then we will deal with the exception handling mechanism in Linux and Optimizing strategy using Fast EHM.

2. Exception Handling in C++

Exceptions are a method of responding to unusual conditions (like runtime errors) in the program using functions called “handlers”.

Exceptions are caught by placing a block of code below the exception detection code. A try block is used to contain this block of code. When we encounter an anomalous or an exceptional condition within the try block, an exception is said to be thrown. The exception handler assumes the control from this point. If there are no such errors encountered in try block of code then no exception is thrown, the code follows normal flow of execution without entering the handlers. The keyword throw is used to throw the exception inside the try block code. The keyword catch is used to declare the exception handlers and written right after the try block. Below is an example:

```cpp
do_function()
{
    try{ // block of code being detected for exception
        throw new exception(); // throw exception
    }
    catch(exception &e) // catch the exception of type exception
    { … }  // catch the exception of type exception
}
```

A “throw” basically takes a single parameter and this same argument is passed to the exception handler so as to handle that exception. The “catch” keyword is used to declare an exception handler. The catch block looks like a normal function and accepts at least one parameter. When “throw” statement throws an exception, the parameter which is sent, is checked against the argument in the catch block. The exception is caught only if the parameters match. Multiple exceptions handlers can be written for the same “try” block and in that case each one must have different parameters. When the exception is thrown, only the catch block matching the parameter is executed. C++ provides the option of the using the ellipsis (…) features. When this is specified for a catch block, it can match of exception irrespective of what kind of parameter is sent when the exception is thrown.

Before explaining the C++ exception handling backend, we introduce a few concepts here.

### 2.1 Stack Frame

Stack frames [7] are data structures that store the state information of each subroutine. They are machine and ABI dependent. A stack frame is created for each of the subroutines which were invoked but are not yet terminated. Fig 4 shows a simple stack frame.

![Stack Frame Diagram](image-url)

**Fig1. Stack frame**
2.2 Stack Unwinding
Whenever an exception is thrown and handled, we need a mechanism for deallocating the memory allocated to all the objects constructed at the beginning of try block. To facilitate this, the destructors of all the allocated objects are called by a process called stack unwinding [4]. The destructors are called in the reverse order of their creation.
The `terminate()` function will be called if an exception is thrown during the stack unwinding process.
Stack unwinding is primarily done for two reasons:
- Language defined exceptions support them. Ex: C++
- "forced" unwinding. Ex: caused by `longjmp` or thread termination

Though the interface tries to maintain both of them similar, there is a notable difference:
- Consider the case when an exception is thrown: In this case, the personality routine for each stack frame should know what it has to do i.e. either handle the exception or ignore it.
- In the case of forced unwinding, an external agent is responsible for handling how the stack unwinding takes place. For example, in case of “longjump”, the external agent decides when the unwinding is to be stopped.

2.3 Landing Pad
Landing Pad [4] is the user code that is responsible for handling the exception or catching the exception. It is also responsible for cleanup after the exception. Control is transferred to Landing Pad code with the help of personality routine.

2.4 Itanium Base ABI for Exception Handling
We introduce the Unwind Library interface in this section. This interface is used to build the C++ ABI exception handling facilities [4].

2.4.1 Unwind Process
The unwind process[4] is a two phase process. It is started by the runtime when an exception is raised. The two phases are as follows:
- Search Phase: The search phase repeatedly invokes the personality routine for each stack frame. It traverses all the stack frames till either it finds the handler routine in a frame or a handler is not found in any of the frames. If the search phase does not find any handler, it calls `terminate()` instead of beginning with phase 2.
- Cleanup Phase: If an exception handler is found in the Search Phase, the cleanup phase is started. Again, like the search phase, the cleanup phase invokes the personality routine repeatedly for each stack frame, but uses the `_UA_CLEANUP_PHASE` flag. This is done till it reaches the stack frame that contains the exception handler routine. Then, the landing pad code gets the control.

2.4.2 Data Structure
The following data structure is being used by the Itanium Architecture.

Exception Header: The exception being thrown needs to be represented and identified by the unwind interface. To serve this purpose, we have a pointer to an exception header object[4]. The
header defined as follows is understood by the unwind interface. The exception object (language and implementation dependent) is prefixed by this header.

```c
typedef void (*_Unwind_Exception_Cleanup_Fn)
    (_Unwind_Reason_Code reason,
     struct _Unwind_Exception *exc);
```

```c
struct _Unwind_Exception {
    uint64    exception_class; //set by user code
    _Unwind_Exception_Cleanup_Fn    exception_cleanup; //set by user code
    uint64     private_1; //set by runtime
    uint64     private_2; //set by runtime
};
```

An `Unwind_Exception` is double-word aligned.

### 2.4.3 Routines

The Itanium Base ABI provides the following routines[4] for exception handling and stack unwinding:

- **Unwind_RaiseException**: This function raises an exception.
- **Unwind_Resume**: This function resumes unwinding the stack after cleanup has been performed by the landing pad code.
- **Unwind_DeleteException**: This function is used to delete the given exception object.
- **Unwind_GetGR**: This function is used to get the 64-bit value in the required general register.
- **Unwind_SetGR**: This function is used to set the 64-bit value in the required register.
- **Unwind_GetIP**: This function is used to fetch the 64-bit value in the instruction pointer (IP).
- **Unwind_SetIP**: This function is used to set the instruction pointer’s (IP) value.
- **Unwind_GetRegionStart**: This function is used to get the address of the code fragment which is described by the current unwind descriptor block.
- **Unwind_GetLanguageSpecificData**: The pointer to the data area which is language-specific for the current stack frame is returned by this function.
- **Unwind_ForcedUnwind**: This function raises an exception for forced unwinding.

### 2.4.4 Personality Routine[4]

#### 2.4.4.1 The Personality Routine

```c
_Unwind_Reason_Code (*__personality_routine)
    ( int version,
      _Unwind_Action actions,
      uint64exceptionClass,
      struct _Unwind_Exception *exceptionObject,
      struct _Unwind_Context *context);
```

We need some interface between the system unwind library and the various language specific semantics for exception handling. This purpose is served by the personality routines in the C++ library. This function has no specified name.
The actions performed by the personality routine are decided based on the value of the ‘actions’ parameter. The actions that can be performed by the personality routine are described below:

_UA_SEARCH_PHASE
   When this bit is set, the personality routine checks the stack frame if it has the exception handler code it is looking for.

_UA_CLEANUP_PHASE
   When this bit is set, the personality routine performs cleanup for the current stack frame.

_UA_HANDLER_FRAME
   If this bit is set, it acts as an indicator to the cleanup phase that this is the frame that contains the landing pad code.

_UA_FORCE_UNWIND
   When this bit is set, it signifies that no one is allowed to handle the exception.

2.4.4.2 Transferring Control to a Landing Pad
When the personality routine identifies the frame which contains the landing pad code (marked in the search phase), it sets up the registers with appropriate values and then returns _URC_INSTALL_CONTEXT.
Let us look in more detail how it is done. Here is some background information on .eh_frame [6] here.

2.4.4.3 .eh_frame
We need some description as to how the stack unwinding should take place. To facilitate that, gcc produces tables while generating code that handles exceptions. .eh_frame section in the assembly code generated contains these tables. Some key points are:

- Multiple records make up each .eh_frame.
- The two kinds of records possible in each frame are, firstly CIE (Common Information Entry) and secondly, an FDE (Frame Description Entry). One CIE is generally used per object file. A list of FDEs comprises a CIE and each single function corresponds to a FDE.
- Both the CIE and FDE are used together to describe how stack unwinding is done.
- Each executing instruction is covered by one FDE.

We can get the pointer to the personality routine from CIE and pointer to the LSDA can be obtained from FDE.

2.5 ItaniumC++ ABI for Exception Handling
In this section we give a detailed explanation of the C++ ABI for exception handling[4]. It is build on the Base ABI for exception handling which we have explained in section 2.4

2.5.1 Data Structures
The following data structures are used by the C++ ABI for facilitating exception handling.

2.5.1.1 C++ Exception Objects
C++ exception object primarily includes:

- Header: This forms to be a wrapper for the unwind object header.
- C++ exception object being thrown

The header structure is as shown below:
struct __cxa_exception {
    std::type_info * exceptionType;
    void (*exceptionDestructor) (void *);
    unexpected_handler unexpectedHandler;
    terminate_handler terminateHandler;
    __cxa_exception * nextException;
    int handlerCount;
    int handlerSwitchValue;
    const char * actionRecord;
    const char * languageSpecificData;
    void * catchTemp;
    void * adjustedPtr;
    _Unwind_Exception unwindHeader;
};

The fields in the exception object are as follows:

- The type of the exception being thrown is contained in the `exceptionType` field.
- An exception has a destructor and a pointer to this destructor has to be stored in the `exceptionDestructor` field. If the destructor is not specified, then the value contained in the `exceptionDestructor` is NULL.
- The `unexpectedHandler` field contains pointer to unexpected handler.
- The `terminateHandler` contains pointer to the terminate handler.
- A linked list of exceptions can be created using the `nextException` field.
- Multiple handlers can catch an exception object and this count of multiple handlers is being stored in the field `handlerCount`.
- When there are multiple languages present or a same language might have multiple runtimes, `unwindHeader` field ensures that the exception is operating as designed.

### 2.5.1.2 Caught Exception Stack

Each C++ program thread can access `struct __cxa_eh_globals` object. The class is as shown below:

```c
struct __cxa_eh_globals {
    __cxa_exception * caughtExceptions; /* contains all the active exceptions in a stack. All these are linked by the nextException field in the exception header */
    unsigned int uncaughtExceptions; /* contains the number of uncaught exceptions*/
};
```

Every thread contains this information and it might be different for each thread. Any of the below APIs can be used to get the `__cxa_eh_globals` for the current thread:

- `__cxa_eh_globals* __cxa_get_globals(void)` : A pointer to the `__cxa_eh_globals` structure for the current thread is returned. It might also perform some initialization if required.
- `__cxa_eh_globals * __cxa_get_globals_fast(void)` : This API is also similar to `__cxa_get_globals(void)` but it assumes that the current thread has made at-least one call to `__cxa_get_globals` earlier.
2.5.2 C++ ABI: Throwing an Exception

2.5.2.1 Allocating the Exception Object
- The storage allocated to the Exception Object must remain consistent while stack is being unwound. This is because the handler uses the storage allocated to the exception object and must be thread-safe. Hence it is allocated in the heap.
- __cxa_allocate_exception runtime library routine is used to allocate memory to the Exception Object. Below is the function prototype:
  ```c
  void *__cxa_allocate_exception(size_t thrown_size);
  ```

2.5.2.2 Throwing the Exception object
__cxa_throw() routine is used to throw an exception.

```c
void __cxa_throw (void *thrown_exception, std::type_info *tinfo, void (*dest) (void *));
```
- This routine never returns.
- Arguments: The first argument is a pointer to the exception object being thrown. The second argument is a pointer to std::type_info and the third argument is a pointer to the destructor which will be called for destroying the object.

The __cxa_throw routine performs the following operations:
- Get the exception header which is stored in the __cxa_exception for the thrown exception object. We can get the exception header using the below statement:
  ```c
  __cxa_exception *header = ((__cxa_exception *) thrown_exception - 1);
  ```
- Save into the __cxa_exception header, the unexpected and the terminate handlers.
- The thread info i.e. tinfo and the destination i.e. dest argument must be saved in the __cxa_exception header.
- Now the exception_class field contained in the unwind header must be set and then the uncaught_exception flag must be incremented.
- _Unwind_RaiseException present in the system unwind library is then supposed to be called with argument as a pointer to the thrown exception.
- The stack unwinding process begins with the call to __Unwind_RaiseException.

2.5.3 Catching an Exception

2.5.3.1 The Personality Routine
Control to a landing pad is transferred by the personality routine via the unwind library, enabling the latter to do any final cleanup.

Now we see how the personality routine decides whether it has to do a cleanup or catch an exception. This decision is made based on the LSDA. Every language has its own personality function, but all of them do almost the same thing, exclude all the aspects that are irrelevant for the language (e.g., the C language personality routine runs only some of the cleanups irrespective of the exception handlers) [5].

The .gcc_except_table section contains the LSDA with the personality function in the .text section. _Unwind_GetLanguageSpecificData is used to pass a pointer of LSDA to the personality function. The call-site table is present just after the header of LSDA. The call-site table is ordered based on the start address field. There are four fields in each entry in the call-site table (Fig 2). Every entry in the call-site table specifies a sequence of instructions which are given as below:

1. The landing pad base byte offset forms to be the start of the instructions for the current call site. The header encoding is used to encode this.
2. The current call-site instruction byte length. Again the header encoding is used to encode this.
3. The landing pad pointer for this instruction sequence. If there is no pointer then this is 0.
4. An unsigned LEB128 type which specifies the action to take. In the action table it appears to be byte offset+1 and there is no action when the value is zero.

```
.LLSDAT966:
  .byte 0ff //Leds. Language Specific Data Area.
  .byte 0x3 //dw落到pe_omit
  .uleb128 .LLSDAT966-.LLSDAT966 //length of call site table

.LLSDAT966:     //call site table
  .byte 0x1
  .uleb128 .LLSDAC566-.LLSDAC566 //length

.LLSDAC566:
  .uleb128 .LEHE9-.LFB966 //The start of the instructions for the current call site, a byte offset from the
  .uleb128 .LEHE9-.LEHE9 //The length of the instructions for the current call site, in bytes. This is
  .uleb128 .LE-.LFB966 //A pointer to the landing pad for this sequence of instructions, or 0 if there
  .uleb128 0x1 //The action to take, an unsigned LEB128. This is 1 plus a byte offset into the
  .uleb128 .LEHE1-.LFB966 //action table. The value zero means that there is no action.
```

Fig 2. An example of call site table

The following steps are taken by the personality function depending on the entry in the call-site table:

- The call-site table contains no entry for the current PC: When the personality doesn’t find any entry in the call-site table, there will be no exception information. In this case, C++ calls the std::terminate.
- The call-site table contains no entry for the current PC and landing pad is zero: In this case the personality function has nothing to do as there are neither any exceptions to be caught nor any destructors that need to be run. This can be assumed to be a normal case and the unwinder will continue its normal flow.
- The personality function finds that the action record is zero: In this case there are some destructors that need to be run but exceptions to be caught. The personality function will do the operations for running the destructors.
- In none of the above 3 cases, an offset is found in the action table with each entry being a pair of signed LEB128 values. The former value is of type filter and the latter value is a byte offset. If the byte offset is a 0, there are no more actions taken.

2.5.3.2 Exception Handlers

In this section we explain the basic structure and steps taken by an exception handler

- On entering an exception handler, a handler should call:
  ```
  void *__cxa_get_exception_ptr ( void*exceptionObject );
  ```
An adjusted pointer to the exception object is returned by this routine. (The personality routine computes the adjusted pointer in the phase 1 and saves in the exception object.)

- Once the catch parameter is initialized, a handler is supposed to call:
  ```c
  void *__cxa_begin_catch ( void *exceptionObject );
  ```

  The __cxa_begin_catch routine does the following:
  - Firstly the handler count of the exception is incremented.
  - Then the exception is placed on the currently-caught exceptions stack if it’s a new one and this links the exception to the previous stack top.
  - The uncaught_exception count is then decremented.
  - Finally the adjusted pointer to the exception object is returned.

- If the __cxa_begin_catch exits for some reason, the handler must call __cxa_end_catch():

  The __cxa_end_catch routine does the following operations:
  - Firstly the most recently caught exception is located and then the handler count is decremented.
  - If the handler count becomes 0, then the exception is removed from the caught exception stack and destroys the exception.

### 2.5.3.3 Finishing and Destroying the Exception

- An exception is said to be handled:
  1. Right after the parameter of the respective catch clause is initialized or on entering the catch clause.
  2. On entering unexpected handler or terminate handler caused by a throw.

- An exception is said to be finished when the catch clause completes execution or the unexpected handler exits.

### 3. Operating System support for Exception Handling

In this section we explain the support provided by the Linux Operation system for handling exceptions.

#### 3.1 Types of Exceptions:

In the Linux kernel, an exception can be classified into following types[3]:

- **Processor detected exceptions:** These are generated when CPU detects an anomalous exception.
  - These can be further into 3 types
    1. **Faults:** An exception is a fault if the instruction which caused the anomalous condition can be corrected and the same instruction is executed again. Ex: Page Fault
    2. **Traps:** An exception is a trap if the instruction which caused the anomalous condition is not to be executed again once the exception is handled. There is no loss of continuity of control flow in case of both traps and faults. Ex: breakpoint
    3. **Aborts:** This is a severe error and in this case the process has to be terminated. Ex: inconsistent values in system tables.
ii. *Programmed exceptions*: These are programmer generated exceptions usually caused due to bugs in programs like check on address bound or overflow instructions.

The x86 platform has around 20 exceptions (depending on the processor model this may vary). Some of which are as shown below:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
</tr>
<tr>
<td>1</td>
<td>Debug</td>
</tr>
<tr>
<td>2</td>
<td>NMI</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>5</td>
<td>Bounds check</td>
</tr>
<tr>
<td>12</td>
<td>Stack segment fault</td>
</tr>
<tr>
<td>14</td>
<td>Page Fault</td>
</tr>
</tbody>
</table>

3.2 Exception handling mechanism in Linux:
The current exception handling mechanism in Linux is quite complex and is mainly a general purpose handling mechanism to handle both the synchronous exceptions and the asynchronous exceptions which include the interrupts. The diagram below shows the exception handling mechanism in Linux[1,3].

![Fig 3. Exception handling mechanism in Linux[1]](image)

When the user process raises an exception, the hardware saves the execution state of the user process into the registers and invokes the general-purpose kernel exception handler which initializes the kernel stack. Then the specific type of exception is decoded into vector and the appropriate kernel exception handler is invoked. The exception handler converts hardware exception code into Unix/Linux-specific signal. This signal is then posted to the process that caused the exception by setting a per-process flag bit. The kernel then proceeds to deliver the signal to the process. While delivering the signal, the kernel also copies the process state to user
accessible area. The kernel also creates a trampoline code at runtime such that the return from kernel exception handler reaches the trampoline code. The trampoline code invokes the user's signal handler, which, after execution returns to the trampoline code again. The trampoline code returns back to the kernel and finally the kernel then restores the user program's state and the user program resumes its execution.

3.3 Optimization using Fast EHM:
The exception handling mechanism shown above is no doubt a good strategy and that it handles both the synchronous and asynchronous exceptions using this same mechanism. In the current EHM, we see that every time an exception occurs, complete process state is being copied to the user accessible memory and this is a time consuming operation. The kernel doesn’t need to copy all the process state information into the user accessible area since the exception is being dispatched to the process executing currently, also handling some of the exceptions might not need so much of state information.

A Fast Exception Handling Mechanism[1] can be used to optimize such exceptions thereby gaining performance. For Fast EHM to be used, the user process should register the exceptions that can be handled this way and specify the callbacks for the respective signals. Further the user process must specify a user accessible memory where the kernel can store the current PC and the condition register on exception. It is the task of the kernel to ensure that this is stored on the pinned memory.

When an exception occurs,
- The kernel handler checks if this is a user-mode enabled exception. If yes, Fast EHM can be used else the standard EHM is used to handle the exception.
- In case of the user-mode enabled exception, the kernel handler saves the PC and contents of few registers in the user accessible area specified by the process i.e. partial state is saved. When the kernel handler returns with a signal to the user process, user’s signal handler is invoked using the contents of the condition register saved on the pinned memory.

![Fig 4. Fast Exception Handling Mechanism[1]](#)

- After executing the user's signal handler, there is no need to switch to kernel mode again because the PC was also saved in the pinned memory. Thus we can again continue with execution of the user process.
There are a couple of constraints in using the Fast EHM:

1. Fast EHM can be used only if the exception is to be delivered to the currently executing process since we are not saving the complete process state information.
2. Not all exceptions can be handled this way. Examples of those that can be handled this way are data unaligned accesses, memory protection faults, etc.

References:


