1 Exception Handling

- What should a program do if it tries to pop an element off an empty stack, or divides by 0, or indexes outside an array, or produces an arithmetic error, such as overflow?

- In C, many procedures will return a status code. In most cases programmers will “forget” to check this status flag.

- Modern languages have built-in exception handling mechanisms. When an exception is raised (or thrown) it must be handled or the program will terminate.

- Exceptions can be raised implicitly by the run-time system (overflow, array bounds errors, etc), or explicitly by the programmer.

2 Exception Handling...

- When an exception is raised, the run-time system has to look for the corresponding handler, the piece of code that should be executed for the particular exception.

- The right handler cannot be determined statically (at compile-time). Rather, we have to do a dynamic (run-time) lookup when the exception is raised.

- In most languages, you start looking in the current block (or procedure). If it contains no appropriate handler, you return from the current routine and re-raise the exception in the caller. This continues until a handler is found or until we get to the main program (in which case the program terminates with an error).

Exceptions in PL/I
3 So, what happens afterwards?

- What happens after an exception handler has been found and executed?

  **resumption model** Go back to where the exception was raised and re-execute the statement (PL/I).

  **termination model** Return from the procedure (or unit) containing the handler (Ada).

4 Exceptions in PL/I

- Exceptions are declared like this:

  ```plaintext
  ON condition
  statement
  ```

  This statement is not actually executed. It’s just “remembered” until later, should an error occur.

- For example:

  ```plaintext
  ON OVERFLOW
  GOTO error;
  ...
  error:
  PRINT "something bad happened"
  ```

5 Exceptions in PL/I...

- Consider this example:

  ```plaintext
  ...
  ON OVERFLOW
  PRINT "use smaller numbers, please!"
  ...
  A := A*1000000000000;
  ```

- Where does execution continue after the exception handler has executed? *After* the location where the exception was thrown.
6 Exceptions in Modula-3

- Exceptions are declared like this:

```modula3
INTERFACE M;
  EXCEPTION Error(TEXT);
  PROCEDURE P () RAISES {Error};
END M;
```

- Exceptions can take parameters. In this case, the parameter to `Error` is a string. Presumably, the programmer will return the kind of error in this string.

- The declaration of `P` states that it can only raise one exception, `Error`.

- If there is no `RAISES` clause, the procedure is expected to raise no exceptions.

7 Exceptions in Modula-3...

- `S_1` and `S_2` can raise exceptions implicitly, or the programmer can raise an exception explicitly using `RAISE`.

- When the `Error`-exception is raised, the `EXCEPT`-block is searched and the code for the `Error` exception is executed.

```modula3
PROCEDURE P () RAISES {Error};
BEGIN
  TRY
    S_1; RAISE Error("Help!"); S_2;
  EXCEPT
    Error (V) => Write(V); |
    Problem (V) => Write("No Probs!"); |
    ELSE Write("Unhandled Exception!");
  END;
END P;
```

8 Exceptions in Modula-3...

- An unhandled exception is re-raised in the next dynamically enclosing `TRY`-block. If no matching handler is found the program is terminated.

```modula3
MODULE M;
BEGIN
  TRY
    TRY S_1; EXCEPT
      Problem (V) => Write(V);
    END;
    EXCEPT
      Error (V) => Write(V); |
      ELSE Write("Unhandled Exception!");
    END;
  END
END M;
```
9 Exceptions in Modula-3...

- An unhandled exception is re-raised in the calling procedure. Exception handlers can explicitly re-raise an exception, or raise another exception.

```plaintext
MODULE M;
PROCEDURE P ();
BEGIN
  TRY S1; EXCEPT
    Problem (V)=>RAISE Error("OK")
  END;
END P;
BEGIN
  TRY P(); EXCEPT
    Error (V) => Write(V); |
    Problem (V) => Write(V);
  END;
END M;
```
Exceptions in Java

10 Exceptions as classes

- In Java, exceptions are classes. They are declared like this:

  ```java
class StupidError extends Exception {}
```

- An exception is thrown by creating a new exception object, and then calling `throw`:

  ```java
  throw new StupidError("Forgot to buy milk!")
  ```

11 Exception hierarchy

- Java defines a hierarchy of exceptions. Programmers can construct their own by extending one of these classes:

  ```java
class Throwable {}
class Exception extends Throwable {}
class InterruptedException extends Exception {}
class RuntimeException extends Exception {}
class ArithmeticException extends RuntimeException {}
class NullPointerException extends RuntimeException {}
class ClassCastException extends RuntimeException {}
class Error extends Throwable {}
class ThreadDeath extends Error {}
```

12 Catching an exception

- Exceptions are caught like this:

  ```java
  try {
    throw new Exception("They were out of milk!")
  } catch (StupidError e) {
    ...
  } catch (Exception e) {
    ...
  } catch (Throwable e) {
    ...
  }
  ```

- Each `catch`-clause is considered in turn until one is found that is a subclass of the exception object thrown.
13 The finally-clause

- The finally-clause is executed regardless of whether an exception is thrown or not. This can be used to close files, etc:

```java
try {
    throw new Exception("They were out of milk!");
} catch (StupidError e) {
    getACow();
} finally {
    eatCornFlakes();
}
```
Implementing exceptions

14 Implementation

- We want 0-overhead exception handling. This means that – unless an exception is raised – there should be no cost associated with the exception handling mechanism.
- We allow raising and handling an exception to be quite slow.
- When an exception is raised we need to be able to
  1. in the current procedure find the exception handler (if any) that encloses the statement that raised the exception, and
  2. rewind the stack (pop activation records) until a procedure with an exception handler is found.

15 The Range Table

- We build a *RangeTable* at compile-time. It has one entry for each procedure and for each TRY-block.
- Each entry holds four addresses: pc_high, pc_low, handler and cleanup.
- [pc_low..pc_high] is the range of addresses for which handler is the exception handler.

16 The Range Table...

17 Unwinding the Stack (Locate)

- Let procedure S raise exception E at code address V. We search the range table to find an entry which covers V, i.e. for which \( \text{pc_low} \leq V \leq \text{pc_high} \).
- Entry (6) covers all of procedure S (for S to S_end), and hence V. There’s no exception handler for this range. We just execute S’s cleanup code, S_C.
- S_C will restore saved registers, etc, and deallocate the activation record.
18  Unwinding the Stack (Locate)...

Source Code  Stack  Object Code

PROC S()
RAISE E1
END R;

V: RAISE E1
END R;

PROC S()
R
a
n
g
e
T
b
a
l
h
E3
E4
H1
R_C
S_C
S_end/
S_C: <cleanup>
S:
Stack
Return
Addr •
Dynamic
■ Link
((5) (6)
pc_low: E3
pc_high: E4
handler: H1
cleanup: R_C
S_C
S: V: RAISE E1
S_C: <cleanup>

19  Unwinding the Stack (Unwind)

• Since S didn’t have a handler, we must unwind the stack until one is found.
• S’s return address is K, which is covered by entry (5) in the range table. Entry (5) has a handler
defined (at address H1). Run it!

20  Unwinding the Stack (Unwind)...

Source Code  Stack  Object Code

PROC S()
RAISE E1
END R;

PROC R()
TRY
S()
•
EXCEPT
E2 => ...
END
END R;

V: RAISE E1
S_C: <cleanup>
S:
R:
E3:
K
E4:
H1:
R_C:
S_C:
S:
V: RAISE E1
S_C: <cleanup>
R:
E3:
K
E4:
H1:
R_C:
S_C:
S:
V: RAISE E1
S_C: <cleanup>

21  The Exception Handler

• The exception handler itself can be translated as a sequential search.
• If the TRY-EXCEPT-block has no ELSE part, the default action will be to re-raise the exception.

TRY
S1;
S1;
RAISE e;
RAISE e;
S2;
S2;
EXCEPT
IF e = E1 THEN H1
ELSIF e = E2 THEN H2
END;
ELSE
E1 => H1 |
E2 => H2 |
RAISE e
END;
ENDIF;
22 The Algorithm

LOOP
  D := The first procedure descriptor (Range Table entry) such that D.pc_low <= PC <= D.pc_high;
  IF D.handler = the default handler THEN
    abort and coredump
  ELSIF D.handler ≠ NIL THEN GOTO D.handler;
  ELSE
    Execute the cleanup routine D.cleanup;
    PC := Return address stored in the current frame;
    SP := SP of previous frame;
    FP := FP of previous frame;
  END;
END;

23 Example — Explanation of source code

- Consider the example on the next slide.
- The main program calls procedure P(). There is a <default handler> defined for the program at address H3.
- Procedure P() calls Q(). Exception X1 is caught by the handler at address H2.
- Q() calls R().
- R() calls S(). Exception X2 is caught by the handler at address H1.
- S() throws exception X1 at address A1.

24 Example
25 Example — Explanation of Actions

• $A_1 \in [S, S_{end}]$, in Range Table entry (6). (6) has no handler, so we execute its cleanup routine ($S, C$) and update PC to the return address, $A_2$.

• Since $A_2 \in [E_3, E_4]$ in Range Table entry (5), and (5).$handler=H_1 \neq NIL$, we GOTO $H_1$. This handler doesn’t handle exception $X_1$, so it will simply re-raise $X_1$.

• $Q()$ has no handler, so we execute its cleanup routine ($Q, C$) and propagate the exception to $P()$. I.e. We update PC to the return address stored in Q’s frame, $A_4$.

• Since $A_4 \in [E_1, E_2]$ in Range Table entry (2), and (2).$handler=H_2$, we GOTO $H_2$. This handler catches $X_1$. ⇒ Done.
Exceptions in C

26 setjmp/longjmp

- In C, setjmp/longjmp can be used to implement exceptional control flow:

  ```c
  if (!setjmp(buffer)) {
    /* setjmp returned 0. Protected code. */
    ...
    longjmp(buffer);
    ...
  } else {
    /* setjmp returned 1. Handler code. */
  }
  ```

27 setjmp/longjmp...

- The first time setjmp returns 0 and execution continues as normal. When longjmp is called it appears as if setjmp has returned for the second time, this time returning 1. The state is now the same as it was when setjmp was first called.

- setjmp’s buffer argument stores the program’s current state, in particular register values.

- Unlike a “real” exception handler, the stack is not rewound nicely. Rather, all stack frames are thrown away. This can lead to problems if not all register values have been saved back in memory. Variables that may be thus affected should be declared volatile, i.e. they will always be returned to memory after operated on.

28 Readings and References

- Read Scott: pp. 441–452


29 Summary

- The algorithm we’ve shown has no overhead (not even one instruction), unless an exception is thrown.

- The major problem that we need to solve is finding the procedure descriptor for a particular stack frame.

- An alternative implementation would be to store a pointer in each frame to the appropriate descriptor. The extra space is negligible, but it would cost 1-2 extra instructions per procedure call.