CSc 553

Principles of Compilation

18 : Exceptions

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Exception Handling I

- 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.

Exception Handling II

- 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).

Exception Handling III

- 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).

Exceptions in Modula-3 I

- Exceptions are declared like this:

```
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.

Exceptions in Modula-3 II

- 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.

```
PROCEDURE P () RAISES {Error};
BEGIN
  TRY
    S1; RAISE Error("Help!"); S2;
  EXCEPT
    Error (V) => Write(V); |
    Problem (V) => Write("No Probs!"); |
    ELSE Write("Unhandled Exception!");
  END;
END P;
```

Exceptions in Modula-3 III

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

```
MODULE M;
BEGIN
   TRY
      TRY S1; EXCEPT
         Problem (V)=>Write(V);
      END;
   EXCEPT
      Error (V) => Write(V); |
      ELSE Write("Unhandled Exception!");
   END;
END M;
```

Exceptions in Modula-3 IV

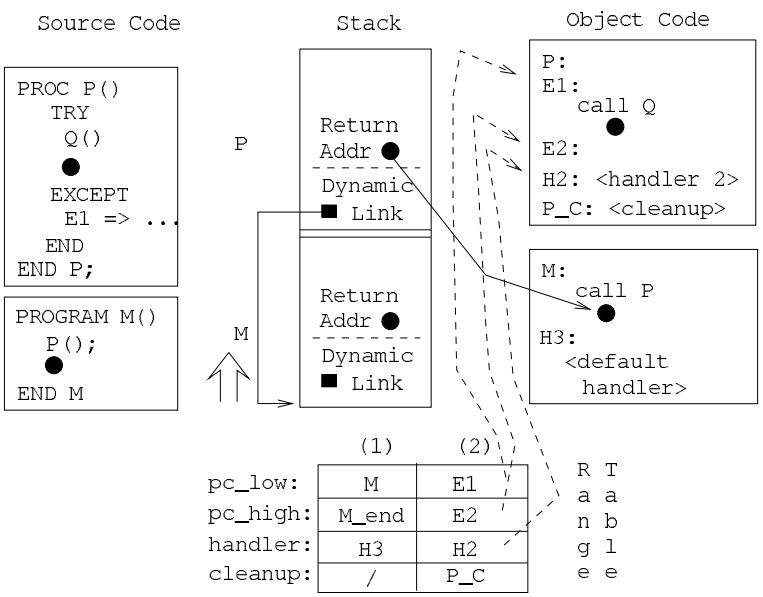
- An unhandled exception is re-raised in the calling procedure.
- Exception handlers can explicitly re-raise an exception, or raise another exception.

```
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;
```

The Range Table

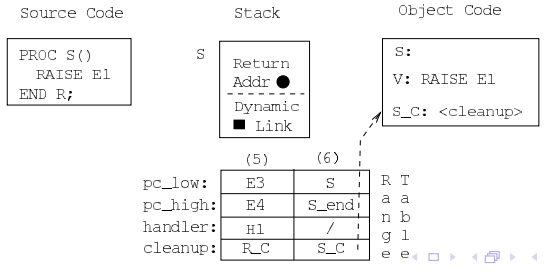
- We want 0-overhead exception handling. This means that – unless an exception is raised – there should be no cost associated with the exception handling mechanism.
- We allow raising and handling an exception to be quite slow.
- When an exception is raised we need to be able to
  1. in the current procedure 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.

- 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.



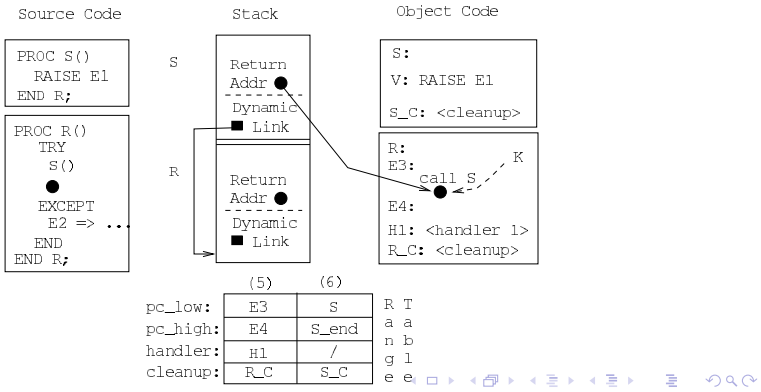
Unwinding the Stack I

- 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 `pc_low<=V<=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.



Unwinding the Stack II

- 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!



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;                        IF e = E1 THEN
EXCEPT                         H1
  E1 => H1 |                 ELSIF e = E2 THEN
  E2 => H2 |                   H2
END;                         ELSE
                               RAISE e
                             ENDIF;
```

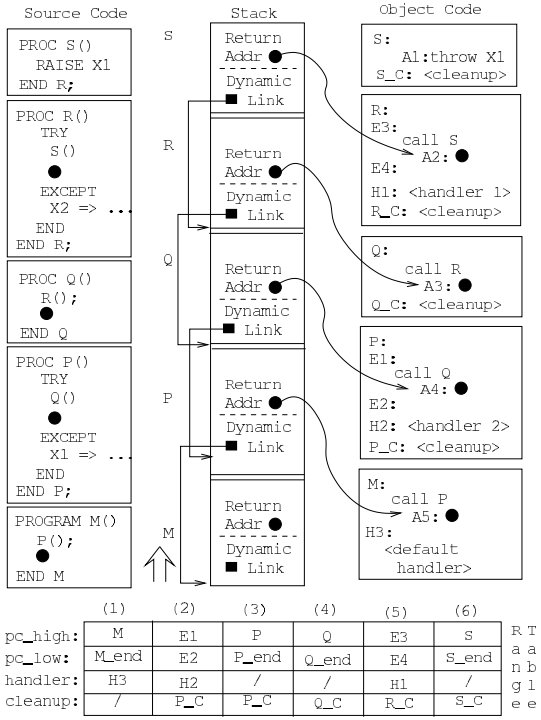
The Algorithm

```
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;
```

Example I (a)

Explanation of the 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.



Example I (b)

Explanation of run-time actions:

- A1∈[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, A2.
- Since A2∈[E3,E4] in Range Table entry (5), and (5).handler==H1≠NIL, we GOTO H1. This handler doesn't handle exception X1, so it will simply re-raise X1.
- 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, A4.
- Since A4∈[E1,E2] in Range Table entry (2), and (2).handler=H2, we GOTO H2. This handler catches X1. ⇒ Done.

Readings and References

- Further reading:
  1. Drew, Gough, Lederman, *Implementing Zero Overhead Exception Handling*, http://www.dstc.qut.edu.au/~gough/zeroex.ps.
  2. Drew, Gough, *Exception handling: Expecting the Unexpected*, Computer Language, Vol 32, No 8, pp. 69–87, 1994.

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.