Subprogram Sequence Control

Subprogram sequence control is related to concept:

How one subprogram *invokes* another and called subprogram returns to the first.

Simple Call-Return Subprograms

- **Program is composed of single main program.**
- **During execution It calls various subprograms which may call other subprograms and so on to any depth**
- **Each subprogram returns the control to the program/subprogram after execution**
- **The execution of calling program is temporarily stopped during execution of the subprogram**
- **After the subprogram is completed, execution of the calling program resumes at the point immediately following the call**

Copy Rule

The effect of the call statement is the same as would be if the call statement is replaced by body of subprogram (with suitable substitution of parameters)

We use subprograms to avoid writing the same structure in program again and again.

Subprogram Sequence Control

Simple Call-Return Subprograms

The following assumptions are made for simple call return structure

i) Subprogram can not be recursive

ii) Explicit call statements are required

iii) Subprograms must execute completely at call

iv) Immediate transfer of control at point of call or return

v) Single execution sequence for each subprogram

Implementation

1. There is a distinction between a subprogram definition and subprogram activation.

Subprogram definition – The written program which is translated into a template.

Subprogram activation – Created each time a subprogram is called using the template created from the definition

2. An activation is implemented as two parts

   Code Segment – contains executable code and constants

   Activation record – contains local data, parameters & other data items

3. The code segment is invariant during execution. It is created by translator and stored statically in memory. They are never modified. Each activation uses the same code segment.

4. A new activation record is created each time the subprogram is called and is destroyed when the subprogram returns. The contents keep on changing while subprogram is executing

Subprogram Sequence Control

Two system-defined pointer variables keep track of the point at which program is being executed.

Current Instruction Pointer (CIP)

The pointer which points to the instruction in the code segment that is currently being executed (or just about to be) by the hardware or software interpreter.

Current Environment Pointer (CEP)

Each activation record contains its set of local variables. The activation record represents the "referencing environment" of the subprogram.

The pointer to current activation record is Current Execution Pointer.

Execution of Program

First an activation for the main program is created and CEP is assigned to it. CIP is assigned to a pointer to the first instruction of the code segment for the subprogram.

When a subprogram is called, new assignments are set to the CIP and CEP for the first instruction of the code segment of the subprogram and the activation of the subprogram.

To return correctly from the subprogram, values of CEP and CIP are stored before calling the subprogram. When return instruction is reached, it terminates the activation of subprogram, the old values of CEP and CIP that were saved at the time of subprogram call are retrieved and reinstated.

Recursive Subprograms

Recursive Subprograms

Recursion is a powerful technique for simplifying the design of algorithms.

Recursive subprogram is one that calls itself (directly or indirectly) repeatedly having two properties

a) It has a terminating condition or base criteria for which it doesn't call itself

b) Every time it calls itself, it brings closer to the terminating condition

In Recursive subprogram calls A subprogram may call any other subprogram including A itself, a subprogram B that calls A or so on.

The only difference between a recursive call and an ordinary call is that the recursive call creates a second activation of the subprogram during the lifetime of the first activation.

If execution of program results in chain such that 'k' recursive calls of subprogram occur before any return is made. Thus 'k+1' activation of subprogram exist before the return from k^{th} recursive call.

Both CIP and CEP are used to implement recursive subprogram.

Exception and Exception Handlers

Type of Bugs -

Logic Errors – Errors in program logic due to poor understanding of the problem and solution procedure.

Syntax Errors – Errors arise due to poor understanding of the language.

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing.

eg. Divide by zero, access to an array out of bounds, running out of memory or disk space

When a program encounters an exceptional condition, it should be Identified and dealt with effectively.

Exception and Exception Handlers

Exception Handling –

It is a mechanism to detect and report an 'exceptional circumstance" so that appropriate action can be taken. It involves the following tasks.

- **Find the problem (Hit the exception)**
- **Inform that an error has occurred (Throw the exception)**
- **Receive the error information (catch the expression)**
- **Take corrective action (Handle the exception)**

```
main()
{  int x, y;
    cout << "Enter values of x and y";
    cin >>x>>y;
   try {
          if (x != 0)
             cout "y/x is ="<<y/x;
          else
             throw(x);
      }
  catch (int i)  {
     cout << "Divide by zero exception caught";
    }
}
```

Exception and Exception Handlers

try – Block contains sequence of statements which may generate exception.

throw – When an exception is detected, it is thrown using throw statement

catch – It's a block that catches the exception thrown by throw statement and handles it appropriately.

catch block immediately follows the try block.

The same exception may be thrown multiple times in the try block.

There may be many different exceptions thrown from the same try block.

There can be multiple catch blocks following the same try block handling different exceptions thrown.

The same block can handle all possible types of exceptions.

```
catch(…)
{
        // Statements for processing all exceptions
}
```

Exception and Exception Handlers

```
procedure sub1()
  divide_zero exception;
  wrong_array_sub exception;
 ---------------- other exceptions
begin
   -----------
   if x = 0 then raise divide_zero;
   -----------
exception
    when divide_zero =>
         ---------------- handler for divide-zero
    when array_sub =>
        ----------------- handler for array sub
end;
```

Exception and Exception Handlers

Propagating an Exception –

If an handler for an exception is not defined at the place where an exception occurs then it is propagated so it could be handled in the calling subprogram. If not handled there it is propagated further.

If no subprogram/program provides a handler, the entire program is terminated and standard language-defined handler is invoked.

After an exception is handled –

What to do after exception is handled?

Where the control should be transferred?

Should it be transferred at point where exception was raised?

Should control return to statement in subprogram containing handler after it was propagated?

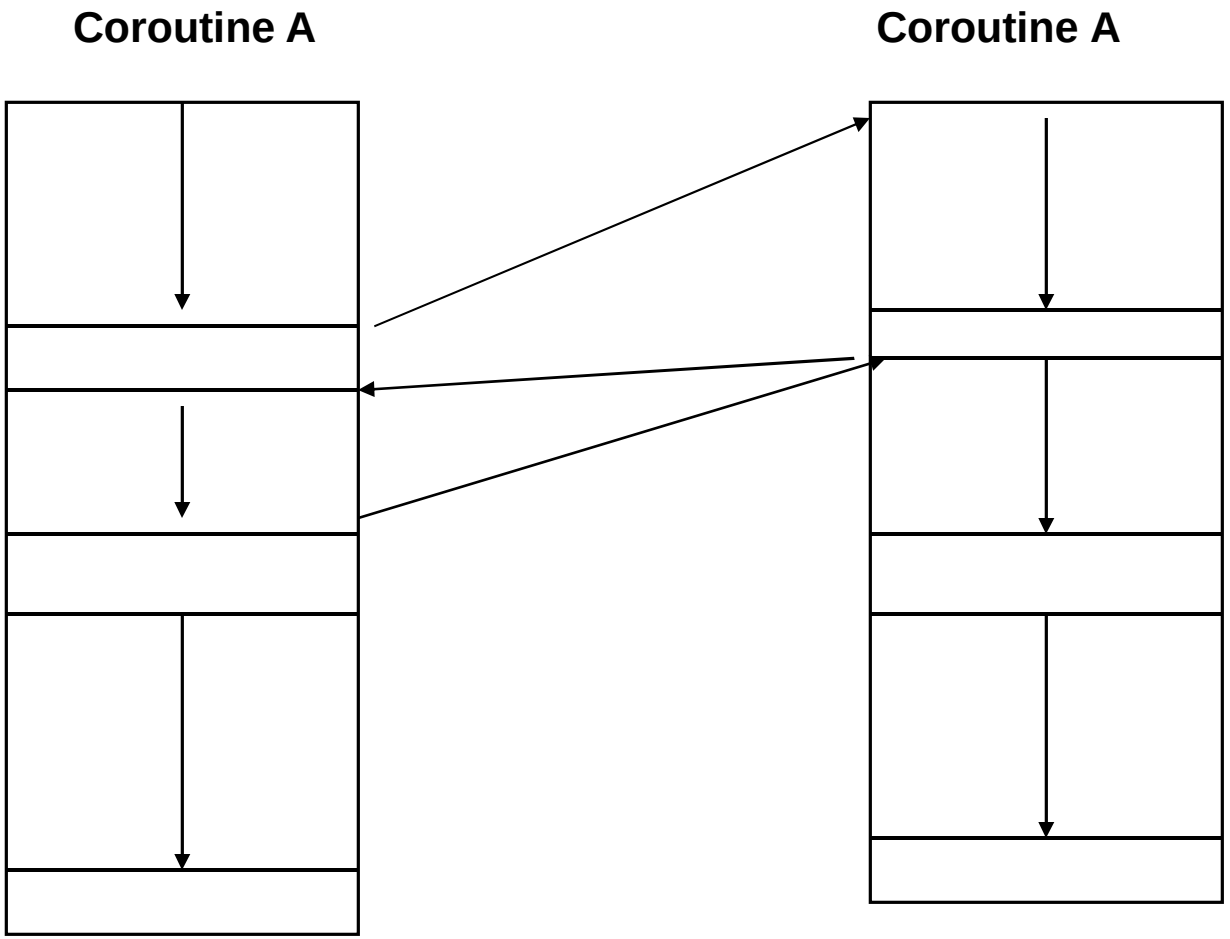
Should subprogram containing the handler be terminated normally and control transferred to calling subprogram? – ADA

Depends on language to language

COROUTINES

COROUTINES –

Coroutines are subprogram components that generalize subroutines to allow multiple entry points and suspending and resuming of execution at certain locations.



COROUTINES

Comparison with Subroutines

1. The lifespan of subroutines is dictated by last in, first out (the last subroutine called is the first to return); lifespan of coroutines is dictated by their use and need,
2. The start of the subroutine is the only point entry. There might be multiple entries in coroutines.
3. The subroutine has to complete execution before it returns the control. Coroutines may suspend execution and return control to caller.

Example: Let there be a consumer-producer relationship where one routine creates items and adds to the queue and the other removes from the queue and uses them.

```
var q := new queue
coroutine produce
  loop
     while q is not full
          create some new items
          add item to q
          yield to consume
```

```
coroutine consume
  loop
     while q is not empty
          remove some items from q
          use the items
     yield to produce
```

COROUTINES

Implementation of Coroutine

Only one activation of each coroutine exists at a time.

A single location, called *resume point* is reserved in the activation record to save the old ip value of CIP when a resume instruction transfer control to another subroutine.

Execution of resume B in coroutine A will involve the following steps:

- **The current value of CIP is saved in the resume point location of activation record for A.**

The ip value in the resume point location is fetched from B's activation record and assigned to CIP so that subprogram B resume at proper location

SCHEDULED SUBPROGRAMS

Subprogram Scheduling

Normally execution of subprogram is assumed to be initiated immediately upon its call

Subprogram scheduling relaxes the above condition.

Scheduling Techniques:

1. Schedule subprogram to be executed before or after other subprograms.

call B after A

2.Schedule subprogram to be executed when given Boolean expression is true

call X when Y = 7 and Z > 0

3. Schedule subprograms on basis of a simulated time scale.

call B at time = Currenttime + 50

4. Schedule subprograms according to a priority designation

call B with priority 5

Languages : GPSS, SIMULA