ISAST Transactions on Computers and Intelligent Systems, No. 2, Vol. 1, 2009
H. Rashidi and Z. Rashidi: A Simple Technique for Implementation of Coroutines in Programming Languages

A Simple Technique for Implementation of Coroutines in Programming Languages

Hassan Rashidi*, Zaynab Rashidi**

** Department of Mathematics, Statistics and Computer Science, Allameh Tabatabaee University, Tehran, Iran
(Email: hrashi@essex.ac.ir; hrashi@atu.ac.ir)
** Department of Computer Engineering, Alzahra University, Tehran, Iran, Email:ZaynabRashidi@gmail.com

ABSTRACT
In this paper, a simple technique for implementation of Coroutines in programming languages is presented. A coroutine is a subprogram that can return control to the caller before completion of execution. There is no direct method for implementation of Coroutines in some programming languages such as C/C++. The technique presented in this paper is based on using a ***static integer variable*** in a ***case*** statement that controls each resumption point in execution time. This technique is applicable to many other programming languages.

KEYWORDS
Programming, Programming languages, Coroutines, Simulation, Distributed Processing.

1. Introduction

Computing has evolved from a solitary activity on a single machine into a federation of cooperating activities that often span the globe. Each activity can be performed by a single function/routine on a single machine. There are a width variety of functions/routines. One of these functions is a coroutine. A coroutine is a subprogram that can return control to the caller before completion of execution[1][2][3]. Coroutines are not currently a common control structure in programming languages outside of discrete simulation languages. However, they provide a control structure in many algorithms that is more natural than the ordinary subprogram hierarchy. Coroutines are commonly used in Parallel Processing, Distributed processing and Simulation.

The question is that how we can implement Coroutines in imperative languages. In some programming languages, there is no direct method for programmers to use a subprogram as Coroutines. In this paper, a simple technique for implementation of Coroutines in C/C++ programming languages is presented.

The remainder of this article is organized as follows. In the next section we give an overview of the relevant work on the matter. In Section 3, a detailed description of the technique is given. The strength and weakness of the technique is presented in a comparative with other solutions. In Section 4. Finally, the experimental results of running the approaches are presented in Section 5.

2. Related Work

When a coroutine receives control from another subprogram, it executes partially and then is suspended when it returns control. At a later point, the calling program may resume execution of the coroutine from the point at which execution was previously suspended.

Figure 1 shows a simple control transfer between two Coroutines A and B. In both Coroutines, note that S_i is a sequence of statements. When A calls subprogram B as a coroutine, B executes awhile and returns control to A, via a ***Resume*** A. just as any ordinary subprogram would do. When A again passes control to B via a ***Resume*** B, B again executes a while and returns control to A, just as ordinary subprogram.

Implementation of Coroutines can be discussed in two levels: Programming level and inside of programming languages itself. There are not important techniques for implementation of Coroutines in programming level. The most common method is that coroutine structure may be readily simulated in many languages using the ***goto*** statement and a resume point variable specifying the label of the statement at which execution is to resume[1].

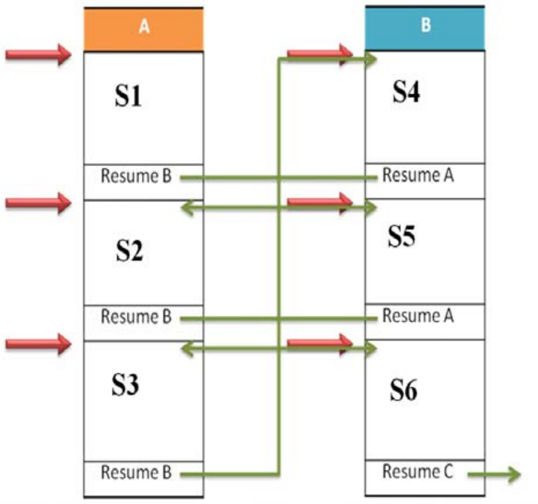

Figure 1: Control transfer between Coroutines

For implementation of Coroutines inside programming languages, more information is needed. In this level suppose the simple call-return structure [1]. Pratt et al (2001) presents a method for implementation of Coroutines inside programming languages. Their implementation based on the simple call- return, but they make differences in handling the CIP (Current Instruction Pointer). Each coroutine has IP location in its activation record, used to provide the value for CIP on resuming the execution. A single activation record is allocated storage statically at the beginning of execution as an extension of the code segment for a coroutine. A single location, which is called the resume point, is reserved in the activation record to save the old IP (Instruction Pointer) of the CIP when a ***resume*** instruction transfers control to another coroutine.

3. The Technique and its details

Our technique is simple and easy to implement by programmers and programming languages. In a short sentence, a coroutine is simply a ***case statement*** with a function return as the last statement of each ***case item***. A single ***static integer variable*** is used to determine an entry point of each execution. A static variable is a variable that its value is held after returning the control to the caller subprogram . The static local variables inside each subprogram are maintained at the end part of subprogram's code segment.

Figure 2 demonstrates a sample of the technique used for implementation of the coroutine that is shown in Figure 1. At the first call, the ***static variable*** A_Control is set to 1 to determine the first entry point. After doing the statements for sequence S1 and before returning the control to the caller subprogram, this variable is set to 2. Therefore, for the second entry point, it does the statements of sequence S2. In summary, each time the coroutine is called, set the variable to an appropriate value to set the appropriate entry point.

Note that in the technique, each coroutine can have some formal parameter as inputs. Moreover, the coroutine can return a single value to the caller subprogram.

```
type  A_Coroutine(…………)
{
          static int A_Control =1;
 typ          e ret_val;
          switch(A_Contro)
          {
             case 1:
                    Statements for S1;
                    A_Control = 2;
                    return(ret_value);
             case 2:
                    Statements for S2;
                    A_Control = 3;
                    return(ret_value);
            case 3:
                    Statements for S3;
                    A_Control = 1;
                    return(ret_value);
          }
}
```

Figure 2: A sample of the technique used

The Figure 3 shows a part of the code segment generated by the programming languages like C/C++ compiler for the program of Figure 2. These languages store and update the value of ***static variable*** at the last location of the code segment. The first instruction fetches value t of the static variable A_Control. The second instruction jumps to an appropriate point of the code depending on the value t. Therefore, the program jumps either to L1 or L2 or L3 depending on t=1, 2, 3 respectively. At each entry point of the code, the program does the instructions for the sequence S_t, t=1,2,3. Then it sets the A_Control to a specific value to determine the next entry point in the next call.

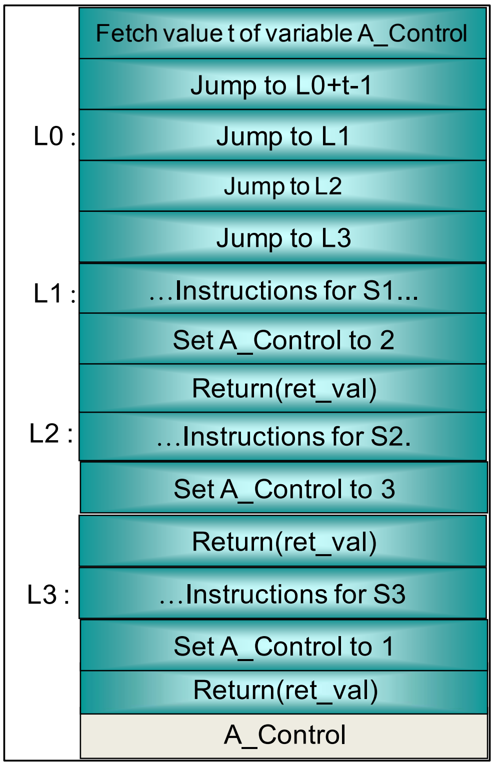

Figure 3: The Code generated by the compiler

The Control structure of the main function for the example in Figure 1 is depicted by Figure 4. The first statements in main dedicated for declarations and initialization. After that, the calling functions make a sequence control as Figure 1.

```
type  main(............)
{
        Declarations and initialization;.
        Call A_Coroutine(...);
        Call B_Coroutine(...);
        Call A_Coroutine(...);
        Call B_Coroutine(...);
        Call A_Coroutine(...);
        Call B_Coroutine(...);
        Call C_Coroutine(...);
}
```

Figure 4: The Control structure of main

4. The Comparative discussion

In this section, a comparative discussion over the techniques is presented. It has some advantages over the techniques in the literature. Firstly, the technique is usable in both programming level and inside programming languages itself. Secondly, the technique presented in this paper can be implemented in any languages. For some languages, for example Pascal and Fortran, that do not support ***static variable*** inside a subprogram, a global variable can be used.

5. Summary and Conclusion

In this paper, a simple technique for implementation of Coroutines in programming languages is presented. A coroutine is simply a ***case statement*** with a function return as the last statement of each ***case item***. We can have a ***static/global variable*** and then set to the appropriate case entry. Each time the coroutine is called, set the ***case*** variable to point to the next option.

Control structures in which subprograms may be invoked either as coroutine or ordinary subprograms and in which Coroutines may be recursive (i.e. may have multiple simultaneous activation) require more complex implementation.

REFERENCES:

[1]. Pratt T. and Zelkowitz M., "Programming Languages, Design and Implementation", 4th Edition Prentice Hall, 2001.

[2]. Sebesta R. W., "Concepts of Programming Languages", 6th Edition, Addison Wesley, 2004.

[3]. Tucker and Noonan, Programming Languages. McGraw Hill, 2002.