9

Runtime Administration

1. Define runtime environment. What are the issues in runtime environment?

Ans: The source language definition contains various abstractions such as names, data types, scopes, bindings, operators, parameters, procedures, and flow of control constructs. These abstractions must be implemented by the compiler. To implement these abstractions on target machine, compiler needs to cooperate with the operating system and other system software. For the successful execution of the program, compiler needs to create and manage a **runtime environment**, which broadly describes all the runtime settings for the execution of programs.

In case of compilation of a program, the runtime environment is indirectly controlled by generating the code to maintain it. However, in case of interpretation of the program, the runtime environment is directly maintained by the data structures of the interpreter.

The runtime environment deals with several issues which are as follows:

- The allocation and layout of storage locations for the objects used in the source program.
- The mechanisms for accessing the variables used by the target program.
- The linkages among procedures.
- The parameter passing mechanism.
- The interface to input/output devices, operating systems and other programs.

2. What are the important elements in runtime environment?

Ans: The important elements that constitute a runtime environment for a program are as follows:

- **Memory organization:** During execution, a program requires certain amount of memory for storing the local and global variables, source code, certain data structures, and so on. The way memory is organized for storing these elements is an important characteristic of runtime environment. Different programming languages support different memory organization schemes. For example, `C++` supports the use of pointers and dynamic memory with the help of `new()` and `delete()` functions, whereas FORTRAN 77 does not support pointers and usage of dynamic memory.
- **Activation records:** The execution of a procedure in a program is known as the **activation** of the procedure. The activation of procedures or functions is managed with the help of a contiguous

block of memory known as **activation record**. Activation record can be created statically or dynamically. Statically, a single activation record can be constructed, which is common for any number of activations. Dynamically, number of activation records can be constructed, one for each activation. The activation record contains the memory for all the local variables of the procedure, depending on the way by which activation record is created, the target code has to be generated accordingly to access the local variables.

- **Procedure calling and return sequence:** Whenever a procedure is invoked or called, certain sequence of operations need to be performed, which include evaluation of function arguments, storing it at a specified memory location, transferring the control to the called procedure, etc. This sequence of operations is known as **calling sequence** and **procedure calling**. Similarly, when the activated procedure terminates, some other operations need to be performed such as fetching the return value from a specified memory location, transferring the control back to the calling procedure, etc. This sequence of operations is known as **return sequence**. The calling sequence and return sequence differ from one language to another, and in some cases even from one compiler to another for the same language.
- **Parameter passing:** The functions used in a program may accept one or more parameters. The values of these parameters may or may not be modified inside the function definition. Moreover, the modified values may or may not be reflected in the calling procedure depending on the language used. In some languages like PASCAL and C++, some rules are specified which determine whether the modified value should be reflected in the calling procedure. In certain languages like FORTRAN77 the modified values are always reflected in the calling procedure. There are several techniques by which parameters can be passed to functions. Depending on the parameter passing technique used, the target code has to be generated.

3. Give subdivision of runtime memory.

Or

What is storage organization?

Or

Explain stack allocation and heap allocation?

Or

What is dynamic allocation? Explain the techniques used for dynamic allocation (stack and heap allocation).

Ans: The target program (already compiled) is executed in the runtime environment within its own logical address space known as **runtime memory**, which has a storage location for each program value. The compiler, operating system, and the target machine share the organization and management of this logical address. The runtime representation of the target program in the logical address space comprises data and program areas as shown in Figure 9.1. These areas consist of the following information:

- The generated target code
- Data objects
- Information to keep track of procedure activations.

Since the size of the target code is fixed at compile time, it can be placed in a statically determined area named *Code* (see Figure 9.1), which is usually placed in the low end of memory. Similarly, the memory occupied by some program data objects such as global constants can also be determined at

compile time. Hence, the compiler can place them in another statically determined area of memory named *Static*. The main reason behind the static allocation of as many data objects as possible is that the compiler could compile the addresses of these objects into the target code. For example, all the data objects in FORTRAN are statically allocated.

The other two areas, namely, *Stack* and *Heap* are used to maximize the utilization of space of runtime. The size of these areas is not fixed, that is, as the program executes their size can change. Hence, these areas are dynamic in nature.

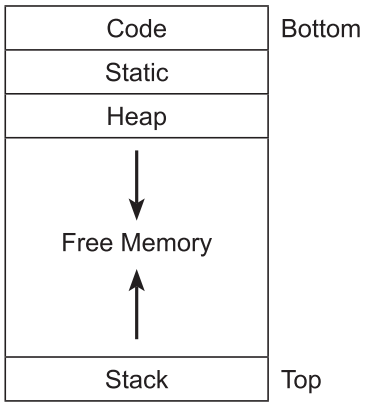

Figure 9.1 Subdivision of Runtime Memory

Stack allocation: The stack (also known as **control stack** or **runtime stack**) is used to store activation records that are generated during procedure calls. Whenever a procedure is invoked, the activation record corresponds to that procedure is pushed onto the stack and all local items of the procedure are stored in the activation record. When the execution of procedure is completed, the corresponding activation record is popped from the stack and the values of locals are deleted.

The stack is used to manage and allocate storage for the active procedure such that

- On the occurrence of a procedure call, the execution of the calling procedure is interrupted, and the activation record for the called procedure is constructed. This activation record stores the information about the status of the machine.
- On receiving control from the procedure call, the values in the relevant registers are restored and the suspended activation of the calling procedure is resumed, and then the program counter is updated to the point immediately after the call. The stack area of runtime storage is used to store all this information.
- Some data objects which are contained in this activation and their relevant information are also stored in the stack.
- The size of the stack is not fixed. It can be increased or decreased according to the requirement during program execution.

Runtime stack is used in C and Pascal.

Heap allocation: The main limitation of stack area is that it is not possible to retain the values of non-local variables even after the activation record. This is because of last-in-first-out nature of stack allocation. To retain the values of such local variables, heap allocation is used. The heap allocates a contiguous memory locations as and when required for storing the activation records and other data elements. When the activation ends, the memory is deallocated, and this free space can be further used by the heap manager. The heap management can be made efficient by creating a linked list of free blocks. Whenever some memory is deallocated, the free block is appended in the linked list, and when memory needs to be allocated, the most suitable (best-fit) memory block is used for allocation. The heap manager dynamically allocates the memory, which results into a runtime overhead of taking care of defragmentation and garbage collection. The garbage collection enables the runtime system to automatically detect unused data elements and reuse their storage.

4. Explain static allocation. What are its limitations?

Ans: An allocation is said to be static if all data objects are stored at compile time. It has the following properties:

- The binding of names is performed during compilation and no runtime support package is required.
- The binding remains same at runtime as well as compile time.
- Each time a procedure is invoked, the names are bounded to the same storage locations. The values of local variables remain unchanged before and after the transfer of controls.
- The storage requirement is determined by the type of a name.

Limitations of static allocation are given as follows:

- The information like size of data objects and constraints on their memory position needs to be present during compilation.
- Static allocation does not support any dynamic data structures, because there is no mechanism to support run-time storage allocation.
- Since all the activations of a given procedure use the same bindings for local names, recursion is not possible in static allocation.

5. Explain in brief about control stack.

Ans: A stack representing procedure calls, return, and flow of control is called a **control stack** or **runtime stack**. Control stack manages and keeps track of the activations that are currently in progress. When the activation begins, the corresponding activation node is pushed onto the stack and popped out when the activation ends. The control stack can be nested as the procedure calls or activations nest in time such that if `p` calls `q`, then the activation of `q` is nested within the activation of `p`.

6. Define activation tree.

Ans: During the execution of program, activation of the procedures can be represented by a tree known as **activation tree**. It is used to depict the flow of control between the activations. Activations are represented by the nodes in activation tree where each node corresponds to one activation, and the root node represents the activation of the `main` procedure that initiates the program execution. Figure 9.2 shows that the `main()` activates two procedures P_1 & P_2. The activations of procedures P_1 & P_2 are represented in the order in which they are called, that is, from left to right. It is important to note that the left child node must finish its execution before the activation of right node can begin. The activation of P_2 further activates two procedures P_3 and P_4. The flow of control between the activations can be depicted by performing a depth first traversal of the activation tree. We start with the root of the tree. Each node is visited before its child nodes are visited and the child nodes are visited from left to right. When all the child nodes of a particular node have been visited, we can say that the procedure activation corresponding to a node is completed.

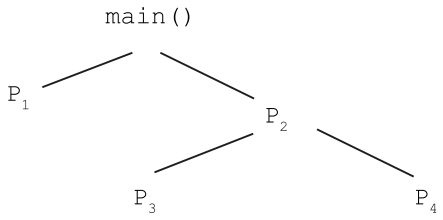

Figure 9.2 Activation Tree

7. Discuss in detail about activation records.

Ans: The **activation record** is a block of memory on the control stack used to manage information for every single execution of a procedure. Each activation has its own activation record with the root of activation tree at the bottom. The path from one activation to another in the activation tree is determined by the corresponding sequence of activation records on the control stack.

Different languages have different activation record contents. In FORTRAN, the activation records are stored in the static data area while in C and Pascal, the activation records are stored in stack area. The contents of activation records are shown in Figure 9.3.

The various fields of activation record are as follows:

- **Temporaries:** The temporaries are used to store intermediate results that are generated during the evaluation of an expression and cannot be held in registers.
- **Local data:** This field contains local data like local variables, which are local to the execution of a procedure stored in the activation record.
- **Saved machine status:** This field contains the information regarding the state of a machine just before the procedure is called. This information consists of the machine register contents and the return address (program counter value).

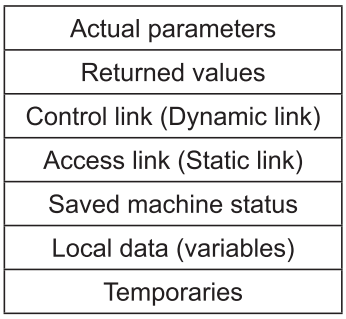

Figure 9.3 Activation Record Model

- **Access link:** It is an optional field and also called **static link** field. It is a link to non-local data in some other activation record which is needed by the called procedure.
- **Control link:** It is also an optional field and is called **dynamic link** field. It points to the activation record of the calling procedure.
- **Returned value:** It is also an optional field. It is not necessary that all the procedures return a value, but if the procedure does, then for better efficiency this field stores the return value of the called procedure.
- **Actual parameters:** This field contains the information about actual parameters which are used by the calling procedure.

8. Explain register allocation.

Ans: On the target machine, registers are the fastest for the computation as fetching the data from the registers is easy and efficient as compared to fetching it from the memory. So the instructions that involve the use of registers are much smaller and faster than those using memory operands. The main problem with the usage of the register is that the system has limited number of registers which are not enough to hold all the variables. Thus, an efficient utilization of registers is very important to generate a good code. The problem of using registers is divided into two sub problems:

- **Register allocation:** It includes selecting a set of variables to be stored within the registers during program execution.
- **Register assignment:** It includes selecting a specific register to store a variable.

The purpose of register allocation is to map a large number of variables into a few numbers of registers, which may result in sharing of single register by several variables. However, since two variables in use cannot be kept in the same register at the same time, therefore, the variables that cannot be assigned to any register must be kept in the memory.

Register allocation can be done either in intermediate language or in machine language. If register allocation is done during intermediate language, then the same register allocator can be used for several target machines. Machine languages, on the other hand, initially use symbolic names for registers, and register allocation turns these symbolic names into register numbers.

It is really difficult to find an optimal assignment of registers. Mathematically, the problem to find a suitable assignment of registers can be considered as a NP-Complete problem. Sometimes, the target machine's operating system or hardware uses certain register-usage convention to be observed, which makes the assignment of registers more difficult.

For example, in case of integer division and integer multiplication, some machines use even/odd register pairs to store operands and the results. The general form of a multiplication instruction is as follows:

```
M a,b
```

Here, operand a is a multiplicand, and is in the odd register of an even/odd register pair and b is the multiplier, and it can be stored anywhere. After multiplication, the entire even/odd register pair is occupied by the product.

The division instruction is written as

```
D a,b
```

Here, dividend a is stored in the even register of an even/odd register pair and the divisor b can be stored anywhere. After division, remainder is stored in the even register and quotient is stored in the odd register.

Now, consider the two three-address code sequences given in Figure 9.4(a) and (b).

These three-address code sequences are almost same; the only difference is the operator in the second statement. The assembly code sequences for these three-address code sequences are given in Figure 9.5(a) and (b).

Here, L, ST, S, M, and D stand for load, store, subtract, multiply, and divide respectively. R_0 and R_1 are machine registers and SRDA stands for Shift-Right-Double-Arithmetic. SRDA R_0, 32 shifts the dividend into R_1 and clears R_0 to make all bits equal to its sign bit.

```
x = a - b        x = a - b
x = x * c        x = x - c
 x = x/d          x = x/d
   (a)              (b)
```

Figure 9.4 Two Three-address Code Sequences

```
L   R1, a        L    R0, a
S   R1, b        S    R0, b
M   R0, c        S    R0, c
D   R0, d        SRDA R0, 32
ST  R1, x        D    R0, d
                 ST   R1, x
   (a)              (b)
```

Figure 9.5 Assembly Code (Machine Code) Sequences

9. Explain the various parameter passing mechanisms of a high-level language.

Or

What are the various ways to pass parameters in a function?

Ans: When one procedure calls another, the communication between the procedures occurs through non-local names and through parameters of the called procedure. All the programming languages have two types of parameters, namely, *actual parameters* and *formal parameters*. The **actual parameters** are those parameters which are used in the call of a procedure; however, **formal parameters** are those which are used during the procedure definition. There are various parameter passing methods but most of the recent programming languages use *call by value* or *call by reference* or both. However, some older programming languages also use another method *call by name*.

- **Call by value:** It is the simplest and most commonly used method of parameter passing. The actual parameters are evaluated (if expression) or copied (if variable) and then their r-values are passed to the called procedure. **r-value** refers to the value contained in the storage. The values of actual parameters are placed in the locations which belong to the corresponding formal parameters of the called procedure. Since the formal and actual parameters are stored in different memory locations, and formal parameters are local to the called procedure, the changes made in the values of formal parameters are not reflected in the actual parameters. The languages C, C++, Java, and many more use call by value method for passing parameters to the procedures.
- **Call by reference:** In call by reference method, parameters are passed by reference (also known as **call by address** or **call by location**). The caller passes a pointer to the called procedure, which points to the storage address of each actual parameter. If the actual parameter is a name or an expression having an l-value, then the l-value itself is passed (here, l-value represents the address of the actual parameter). However, if the actual parameter is an expression like a + b or 2, having

no l-value, then that expression is calculated in a new location, and the address of that new location is passed. Thus, the changes made in the calling procedure are reflected in the called procedure.

- **Call by name:** It is a traditional approach and was used in early programming languages, such as ALGOL 60. In this approach, the procedure is considered as a macro, and the body of the procedure is substituted for the call in the caller and the formals are literally substituted by the actual parameters. This literal substitution is called **macro expansion** or **in-line expansion**. The names of the calling procedure are kept distinct from the local names of the called procedure. That is, each local name of the called procedure is systematically renamed into a distinct new name before the macro expansion is done. If necessary, the actual parameters are surrounded by parentheses to maintain their integrity.

10. What is the output of this program, if compiler uses following parameter passing methods?

(1) Call by value (2) Call by reference (3) Call by name

The program is given as:

```
void main (void)
{
    int a, b;
    void A(int, int, int);
    a = 2, b = 3;
    A(a + b, a, a);
    printf ("%d", a);
}
void A (int x, int y, int z)
{
    y = y + 1;
    z = z + x;
}
```

Ans: Call by value: In call by value, the actual values are passed. The values `a = 2` and `b = 3` are passed to the function `A` as follows:

```
A (2 + 3, 2, 2);
```

The value of `a` is printed as `2`, because the updated value is not reflected in `main()`.

Call by reference: In call by reference, both formal parameters `y` and `z` have the same reference that is, `a`. Thus, in function `A` the following values are passed.

```
x = 5
y = 2
z = 2
```

After the execution of `y = y + 1`, the value of `y` becomes

```
y = 2 + 1 = 3
```

Since `y` and `z` are referring to the same memory location, `z` also becomes `3`. Now after the execution of statement `z = z + x`, the value of `z` becomes

```
z = 3 + 5 = 8
```

When control returns to `main()`, the value of `a` will now become `8`. Hence, output will be `8`.

Call by name: In this method, the procedure is treated as macro. So, after the execution of the function

```
x = 5
y = y + 1 = 2 + 1 = 3
z = z + x = 2 + 5 = 7
```

When control returns to `main()`, the value of `a` becomes `7`. Hence, output will be `7`.

Multiple-Choice Questions

1. What are the issues that the runtime environment deals with?
   (a) The linkages among procedures
   (b) The parameter passing mechanism
   (c) Both (a) and (b)
   (d) None of these
2. The elements of runtime environment include _____.
   (a) Memory organization
   (b) Activation records
   (c) Procedure calling, return sequences, and parameter passing
   (d) All of these
3. Which of the following area in the memory is used to store activation records that are generated during procedure calls?
   (a) Heap
   (b) Runtime stack
   (c) Both (a) and (b)
   (d) None of these
4. _____ are used to depict the flow of control between the activations of procedures.
   (a) Binary trees
   (b) Data flow diagrams
   (c) Activation trees
   (d) Transition diagram
5. The _____ is a block of memory on the control stack used to manage information for every single execution of a procedure.
   (a) Procedure control block
   (b) Activation record
   (c) Activation tree
   (d) None of these
6. _____ is the process of selecting a set of variables that will reside in CPU registers.
   (a) Register allocation
   (b) Register assignment
   (c) Instruction selection
   (d) Variable selection

7. Which of the following is the parameter passing mechanism of a high-level language?
   (a) Call by value
   (b) Call by reference
   (c) Both (a) and (b)
   (d) None of these

Answers

1. (c) 2. (d) 3. (b) 4. (c) 5. (b) 6. (a) 7. (c)