Storage Organization

The executing target program runs in its own logical address space in which each program value has a location.

STORAGE ORGANIZATION:

The executing target program runs in its own logical address space in which each program value has a location. The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

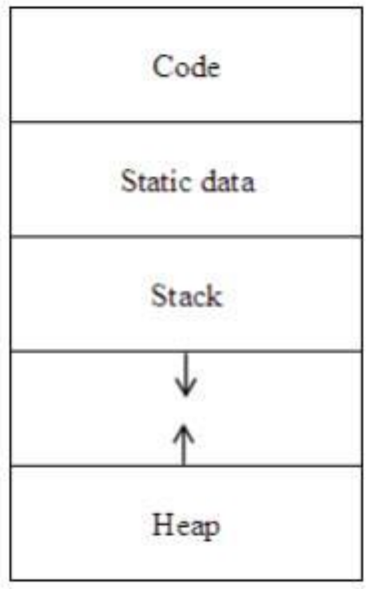


Fig. 2.9 Typical subdivision of run-time memory into code and data areas

form a machine word. Multibyte objects are bytes and given the address of first byte. Run-time storage comes in blocks, where a byte is the smallest unit of memory. Four bytes

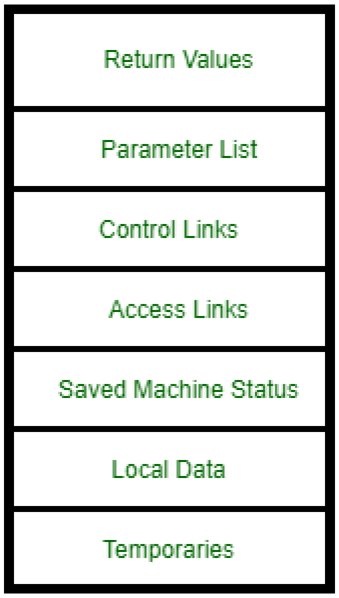
The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
This unused space due to alignment considerations is referred to as padding.

The size of some program objects may be known at run time and may be placed in an area called static.
The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation Record :

An activation record is a contiguous block of storage that manages information required by a single execution of a procedure. When you enter a procedure, you allocate an activation record, and when you exit that procedure, you de-allocate it. Basically, it stores the status of the current activation function. So, whenever a function call occurs, then a new activation record is created and it will be pushed onto the top of the stack. It will remain in stack till the execution of that function. So, once the procedure is completed and it is returned to the calling function, this activation function will be popped out of the stack.
If a procedure is called, an activation record is pushed into the stack, and it is popped when the control returns to the calling function.
Activation Record includes some fields which are –
Return values, parameter list, control links, access links, saved machine status, local data, and temporaries.



Activation Record

Activation Trees

An activation record comprises all the crucial information required to call a procedure. It stores immediate values and temporary values of expression. An activation tree is a structure used to represent the function calls.

Do you know what is an activation tree in compiler design? If not, then don't worry. In this article, we will discuss about activation tree in compiler design and activation records, and we will also discuss an example to understand the concept properly. Moving forward, let's discuss the activation tree in compiler design.

Activation Trees

An activation tree tells the flow of execution of a program and represents it in a graphical manner where each node depicts a particular procedure.
A compiler uses stack-based memory management to manage, allocate, and deallocate memory for executing a program. It uses some units of user-defined action, such as procedures, methods, or functions for managing their runtime memory or a part of memory on the stack.

How is an activation tree created?

Whenever a compiler calls a procedure, it allocates a space for it by pushing it into the stack. After the termination of the procedure, it is popped off the stack. This enables sharing of space by procedure calls. Thus the use of stack makes code compilation efficient. The compiler allocates space to the activation record on the stack’s top.
The execution of a procedure is known as **activation.** An activation record comprises all the crucial information required to call a procedure. It stores immediate values and temporary values of expression.
Each procedure’s activation record is pushed in a stack in a predictable order. A procedure performs a specific task, and whenever it is called an instance of the procedure, an activation is created.
The local variables can be accessed relatively at the code’s beginning and the non-local variable’s address. A program has a sequential flow of control. Control is passed to the part of the procedure called for execution. After the procedure is completed, the control returns to the caller.
Procedure activates a tree where the main program is the **root** and a new node is created whenever a procedure is called. The **children** of the tree

represent the procedures from the particular node A procedure is considered recursive if a new activation starts before the completion of the previous procedure. Therefore an activation tree tells the flow of execution of a program and represents the control flow of a program in a graphical manner where each node depicts a particular procedure.

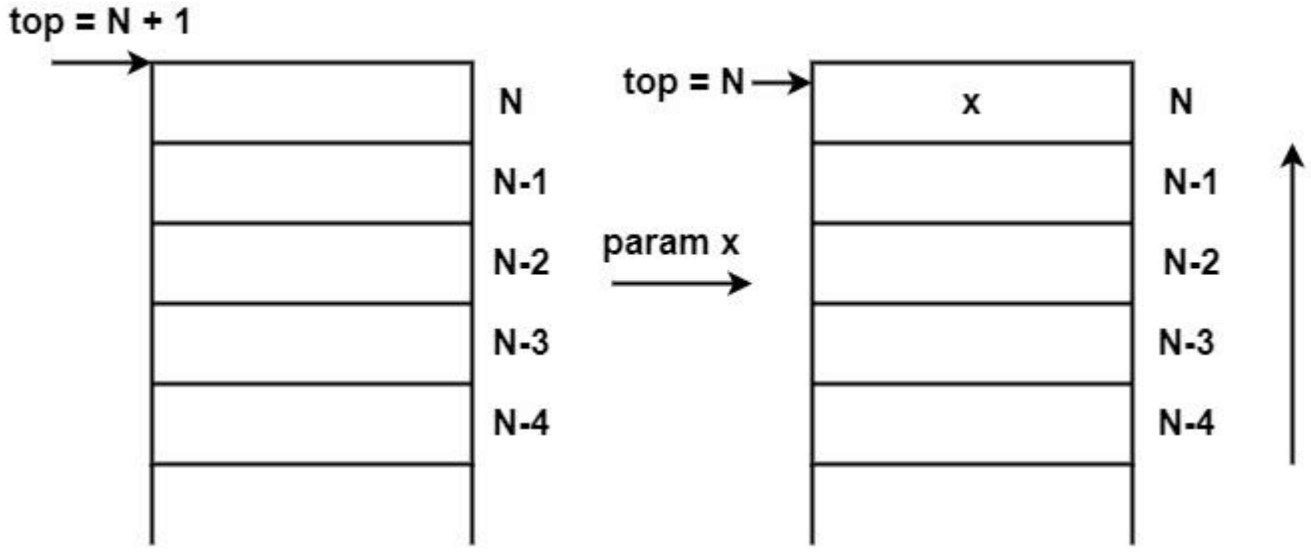
Stack Allocation

It is the simplest allocation scheme in which allocation of data objects is done at compile time because the size of every data item can be determined by the compiler. The main function of static allocation is to bind data items to a particular memory location. The static memory allocation procedure consists of determining the size of the instruction and data space.

Recursive Subprogram and Arrays of adjustable length are not permitted in a language. In static allocation, the compiler can decide the amount of storage needed by each data object. Thus, it becomes easy for a compiler to find the address of these data in the activation record. FORTRAN uses this kind of storage allocation strategy.

The binding of the name with the amount of storage allocated does not change at runtime. Therefore, the name of this allocation is static allocation. In static allocation, the compiler can decide the amount of storage needed by each data object. Therefore, it becomes easy for a compiler to find the address of these data in the activation record.

- **Passing Parameter to Procedure (param x)**− When actual parameter x is passed to the procedure, it will be pushed into the stack, i.e., push (x).



∴ param(x) refers to push (x) which refers to decrementing of the top pointer from N + 1 to N and x will be pushed onto the stack.

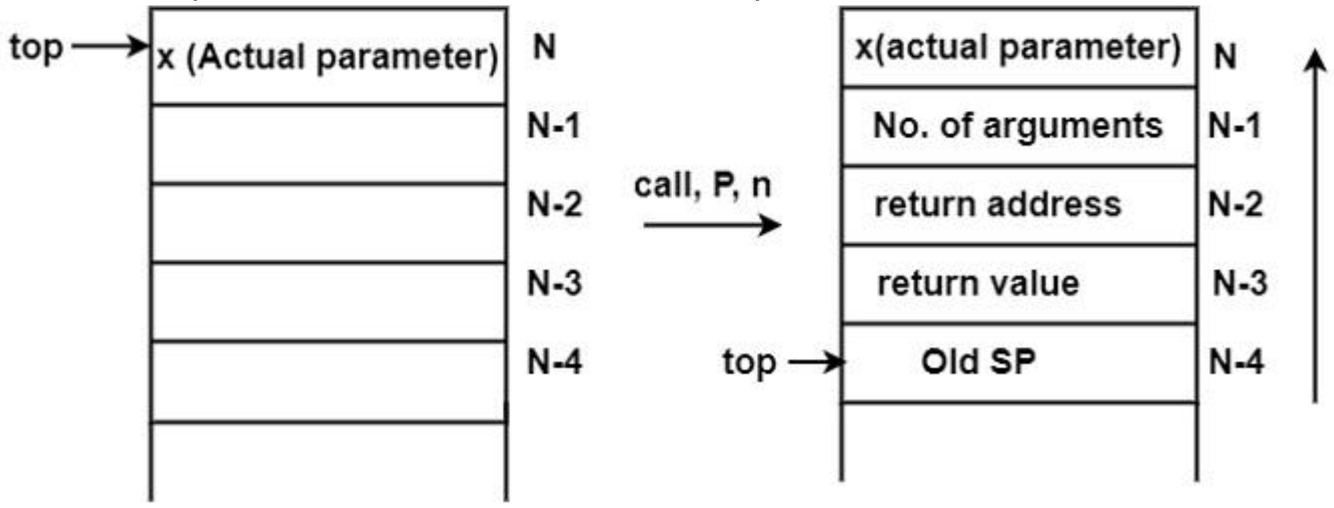
∴ The following statements will be executed.

- top = top – 1
- *top = x or 0[top] = x

Here 0[top] means 0 offsets from the top, i.e., 0 distance away from the top, i.e., the top itself.

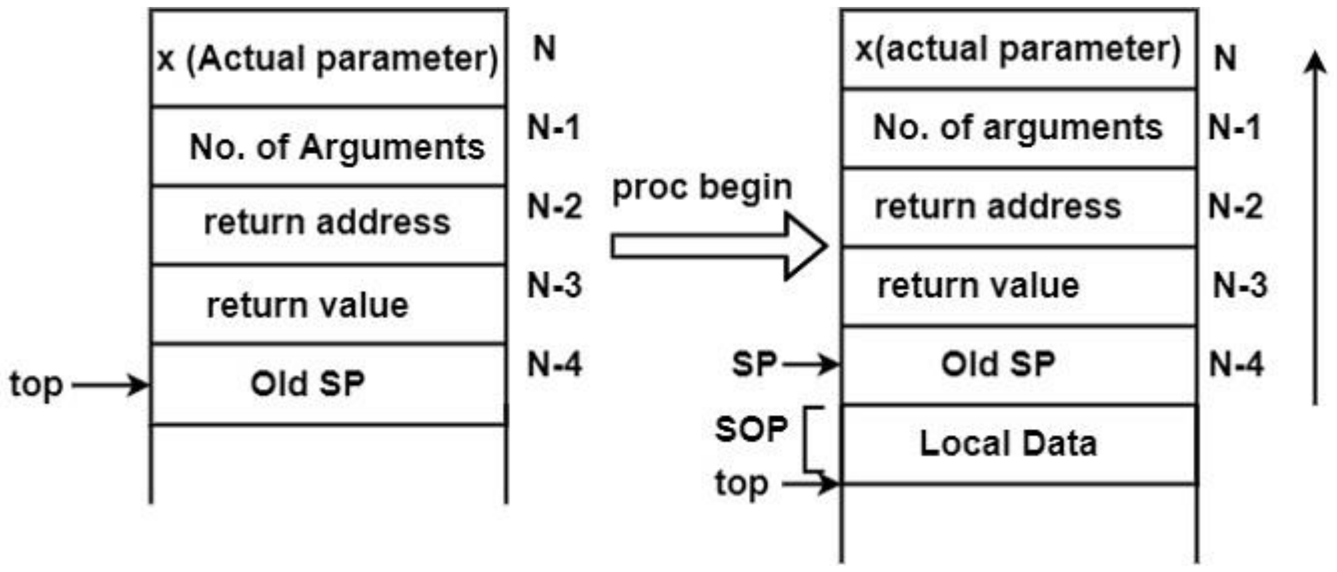
∴ *top = x will assign value of top = x.

- **Calling Procedure (call P, n)**− Execution of this statement will insert all entries of activation record of P, i.e., the number of arguments, return address, space for the return value, old stack pointer onto the stack.



∴ call P, n will lead to the execution of the following statements.

- **push(n)**− Push the number of arguments.
- **push(l_1)**− l1is the label of the return address.
- **push()**− Keep empty space for the return value to be filled.
- **push(Sp)**− Store old stack pointer.
- **goto l_2**− l_2 is the label of the first statement of procedure P.
- **First Statement of procedure (procbegin)**− It assigns the value of stack pointer to old SP. Top pointer will point to the top of the activation record. Local data, i.e., the size of Procedure P is added to the stack.



If SOP= Size of procedure or memory is taken by local data of procedure.

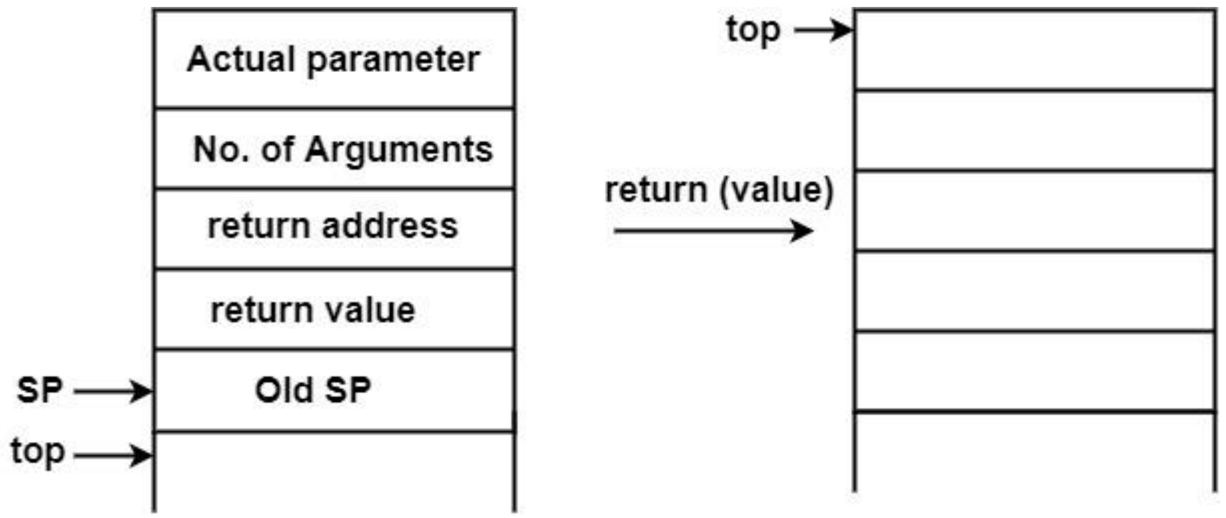
∴ Following statements will be executed −

- $\boldsymbol{SP} = \boldsymbol{top}$− Stack Pointer SP will point to old SP which was earlier stored at the top.
- $\boldsymbol{top} = \boldsymbol{SP} + \boldsymbol{SOP}$− Size of Procedure (SOP) is added to SP to give top position.

Now top pointer will point to the top of the activation record.

- **Return Statement (return value)**− When the procedure returns a value, the returned value will be stored at an empty stack which is kept free for it above the stack pointer (SP) position.

When the procedure returns, the activation record of the procedure will be deleted or popped from the memory location.



When the procedure P returns the value, set the top pointer to the value it had before P was called.

Set pointer P to the value of old SP, i.e., SP of the procedure which had called P.

The top pointer will point to extra storage of activation record of procedure which had called P.

Following statements will be executed on returning a value −

- $\mathbf{1}[\mathbf{SP}] = \mathbf{value}$− Since the return value is 1 location always SP pointer. The returned value will be stored at offset 1 from SP.
- $\mathbf{top} = \mathbf{SP} + \mathbf{2}$− top points to the return address.
- $\mathbf{SP} =* \mathbf{SP}$− Set pointer SP to the value of old SP.
- $\mathbf{l} =* \mathbf{top}$− the value of the l will contain the return address.
- $\mathbf{top} = \mathbf{top} + \mathbf{1}$− top points to the number of arguments.
- $\mathbf{top} = \mathbf{top} + \mathbf{1} +* \mathbf{top}$− Currently, top points to the number of arguments, addition with 1 will move the top pointer to an actual parameter, addition with *top, i.e., the number of arguments will move the top pointer to the position it was earlier before P was called.

Symbol Table

Symbol table is an important data structure used in a compiler.

Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

The symbol table used for following purposes:

2.5M

201

Hello Java Program for Beginners

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

1. <symbol name, type, attribute>

For example, suppose a variable store the information about the following variable declaration:

1. **static int** salary

then, it stores an entry in the following format:

1. <salary, **int**, **static**>

The clause attribute contains the entries related to the name.

Implementation

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

A symbol table can be implemented in one of the following techniques:

- Linear (sorted or unsorted) list
- Hash table
- Binary search tree

Symbol table are mostly implemented as hash table.

Operations

The symbol table provides the following operations:

Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.
- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

1. **int** x;

Should be processed by the compiler as:

1. insert (x, **int**)

lookup()

In the symbol table, lookup() operation is used to search a name. It is used to determine:

- The existence of symbol in the table.
- The declaration of the symbol before it is used.
- Check whether the name is used in the scope.
- Initialization of the symbol.
- Checking whether the name is declared multiple times.

The basic format of lookup() function is as follows:

1. lookup (symbol)

This format is varies according to the programming language