SYSTEM SOFTWARE

- COMPILER

Compiler: A compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language; see Fig. 1 . 1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

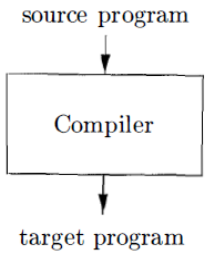


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs as shown in Fig. 1.2.

Figure 1.2: Running the target program

Interpreter: An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

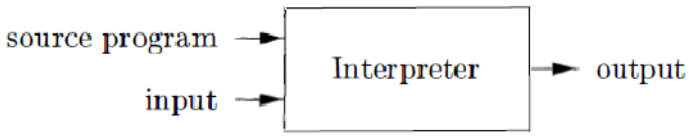


Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Preprocessor, Assembler, Loader, Linker: In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1 .5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as

output and is easier to debug. The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.

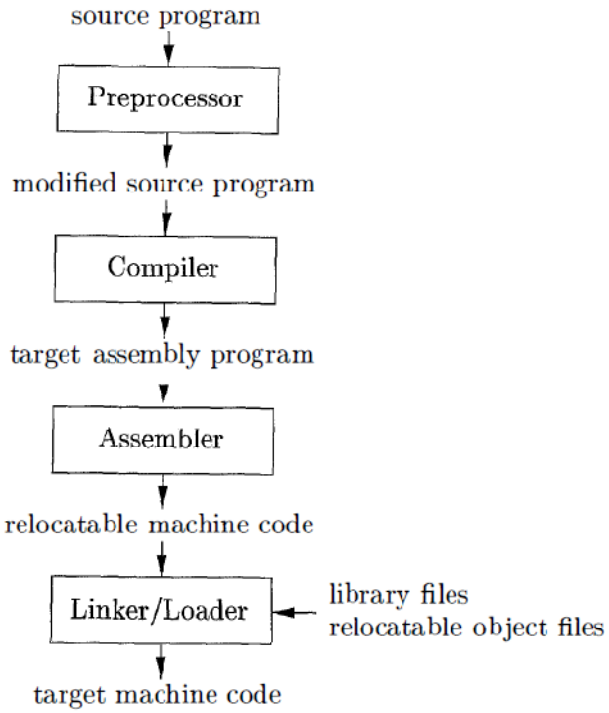


Figure 1.5: A language-processing system

Structure of a Compiler: There are two parts of the compiler: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part. The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1 .6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1 .6 may be missing.

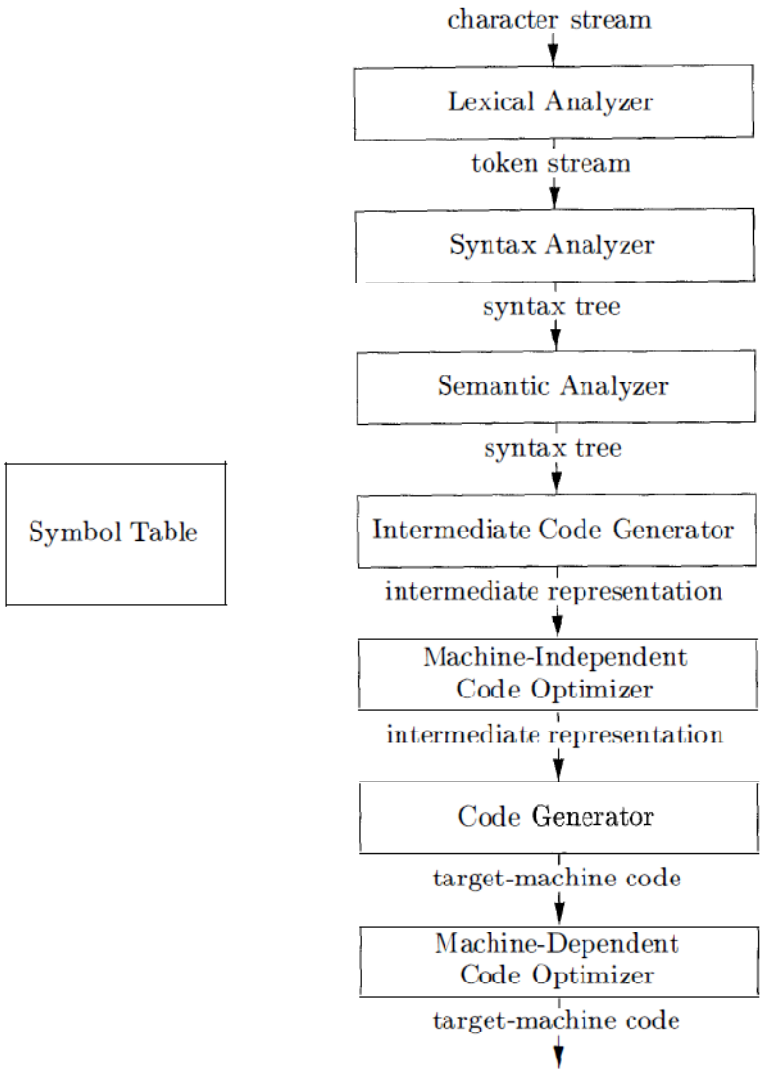


Figure 1.6: Phases of a compiler

Lexical Analysis: The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

{token- name, attribute-value}

that it passes on to the subsequent phase, syntax analysis . In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60 (1.1)

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. **position** is a lexeme that would be mapped into a token {id, 1}, where **id** is an abstract symbol standing for identifier and 1 points to the symbol table entry for **position**. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. **The assignment symbol =** is a lexeme that is mapped into the token {=}. Since this token needs no attribute-value, we have omitted the second component.

3**. initial** is a lexeme that is mapped into the token {id, 2}, where 2 points to the symbol-table entry for **initial**.

4. + is a lexeme that is mapped into the token {+}.

5. **rate** is a lexeme that is mapped into the token {id, 3} , where 3 points to the symbol-table entry for **rate**.

6. * is a lexeme that is mapped into the token {*}.

7. **60** is a lexeme that is mapped into the token {60}.

Blanks separating the lexemes would be discarded by the lexical analyzer. Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

{id, l} { = } {id, 2} {+} {id, 3} { * } {60} (1.2)

In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.

Syntax Analysis: The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1 .2) is shown as the output of the syntactic analyzer in Fig. 1.7.

This tree shows the order in which the operations in the assignment position = initial + rate * 60 are to be performed. The tree has an interior node labeled * with (id, 3) as its left child and the integer 60 as its right child. The node (id, 3) represents the identifier rate. The node labeled * makes it explicit that we must first multiply the value of rate by 60. The node labeled + indicates that we must add the result of this multiplication to the value of initial. The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier posit ion. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

Semantic Analysis: The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

Intermediate Code Generation: In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence:

```
t l = inttofloat (60)
t2 = id3 * t l
t3 = id2 + t2                    (1.3)
id1 = t3
```

Code Optimization: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform (1 .3) into the shorter sequence

```
t1 = id3 * 60.0
id1 = id2 + t 1                                    (1 .4)
```

Code Generation: The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the careful assignment of registers to hold variables.

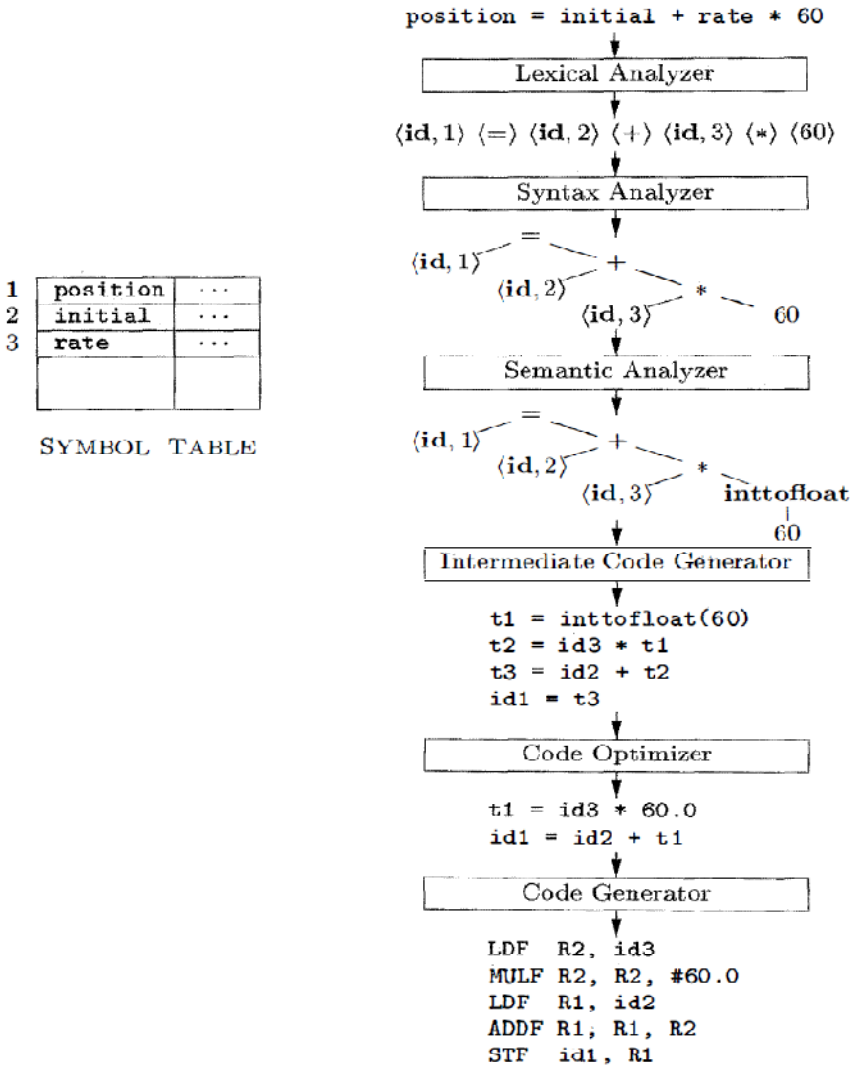


Figure 1.7: Translation of an assignment statement

Differences between Compiler and Interpreter:

No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input.
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

Preprocessor: Preprocessor performs some preprocessing before a program is compiled. Some possible actions are:

- Inclusion of other files in the file being compiled.
- Definition of symbolic constants and macros.
- Conditional compilation of program code or code segment.
- Conditional execution of preprocessor directives.

Macro: Macro is a single instruction that expands automatically into a set of instructions to perform a particular task**.**

Difference between function and Macro: A macro is a preprocessor directive. This means that before your program starts compiling it will go through and process all your macros. Macros are useful because

- macro can make program easier to read
- Since macro calls are replaced by its body no stack is used to store the return address like functions. So no memory space is required for stack.
- Since no return addresses are needed to store in the stack, push and pop operations are not required for macro. Hence macro executes faster than function.
- They can shorten long or complicated expressions that are used a lot.

Disadvantages:

- Expand the size of the executable code. Suppose in the program that there a Macro which used 50 times that means it will consume memory 50 times but in function, if a function is called 50 times it will take single memory every time because every time it de-allocate that memory.

- Can flood your name space if not careful. For example, if you have too many preprocessor macros you can accidentally use their names in your code, which can be very confusing to debug.

Loader: Loader is utility program which takes object code as input prepares it for execution and loads the executable code into the memory. Thus loader is actually responsible for initiating the execution process.

Functions of Loader: The loader is responsible for the activities such as allocation, linking relocation and loading

1) It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.

2) It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.

3) There are some address dependent locations in the program; such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.

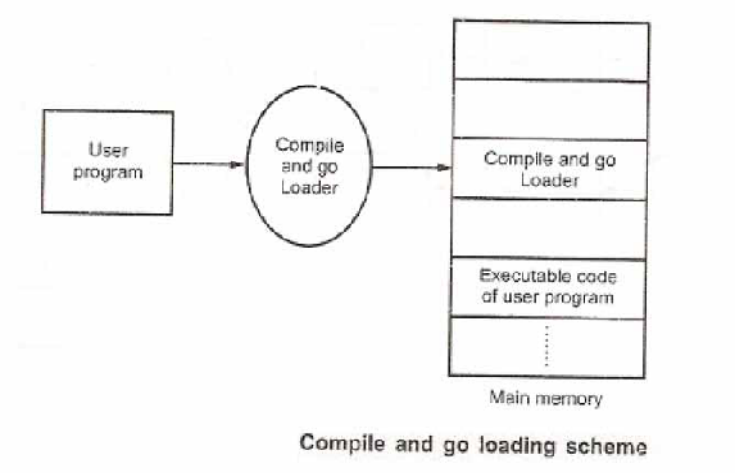
4) Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.

Different types of Loader: Based on the various functionalities of loader, there are various types of loaders:

1. **"compile and go" loader:** In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. This loading scheme is also called as "assemble and go".

   Advantages:
   • This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.

   Disadvantages:
   • In this scheme some portion of memory is occupied by assembler which is simply wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.
   • There is no production of object (.obj) file, the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled and executed each time, which then becomes a time consuming activity.
   • It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.
   • For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the "compile and go" loader cannot handle such programs.
   • The execution time will be more in this scheme as every time program is assembled and then executed.



Compile and go loading scheme

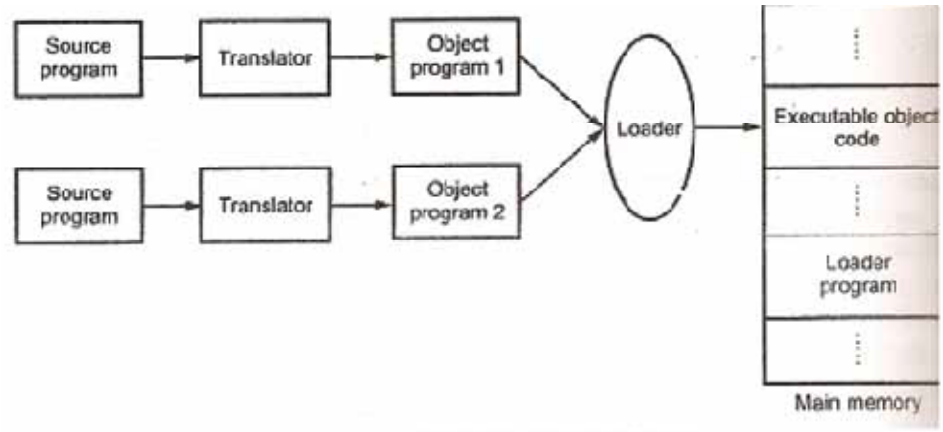
2. **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

Advantages:

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.

• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.

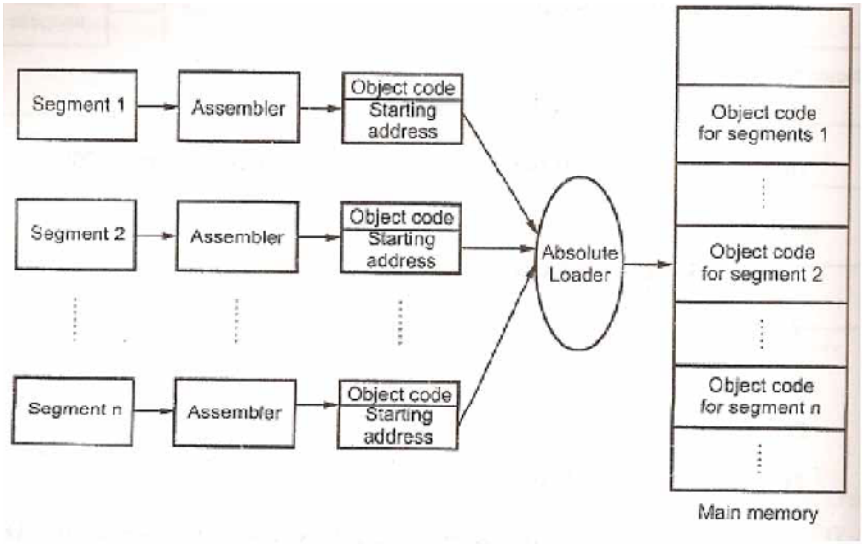
• It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.



General loader scheme

3. **Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is; specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a

change in the starting address of immediate next modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.



Process of absolute loading

Thus the absolute loader is simple to implement in this scheme

1) Allocation is done by either programmer or assembler
2) Linking is done by the programmer or assembler
3) Resolution is done by assembler
4) Simply loading is done by the loader

As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

Advantages:

1. It is simple to implement
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the address resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient.

Disadvantages:

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management. If at all any modification is done the some segments, the starting addresses of immediate next segments may get changed, the programmer has to take care of this issue and he needs to update the corresponding starting addresses on any modification in the source.

4. Direct Linking Loaders: The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative. If at all the address is relative then it is the assembler who informs the

loader about the relative addresses. The assembler should give the following information to the loader

1) The length of the object code segment
2) The list of all the symbols which are not defined in the current segment but can be used in the current segment.
3) The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The USE table holds the information such as name of the symbol, address, address relativity.
The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table. The definition table holds the information such assymbol, address.

Advantages
1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.
2. The system can be dynamically reconfigured.

Disadvantages The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory.

Bootstrap Loader: As we turn on the computer there is nothing meaningful in the main memory (RAM). A small program is written and stored in the ROM. This program initially loads the operating system from secondary storage to main memory. The operating system then takes the overall control. This program which is responsible for booting up the system is called bootstrap loader. This is the program which must be executed first when the system is first powered on. If the program starts from the location x then to execute this program the program counter of this machine should be loaded with the value x. Thus the task of setting the initial value of the program counter is to be done by machine hardware. The bootstrap loader is a very small program which is to be fitted in the ROM. The task of bootstrap loader is to load the necessary portion of the operating system in the main memory .The initial address at which the bootstrap loader is to be loaded is generally the lowest (may be at 0th location) or the highest location.

Concept of Linking: The execution of program can be done with the help of following steps

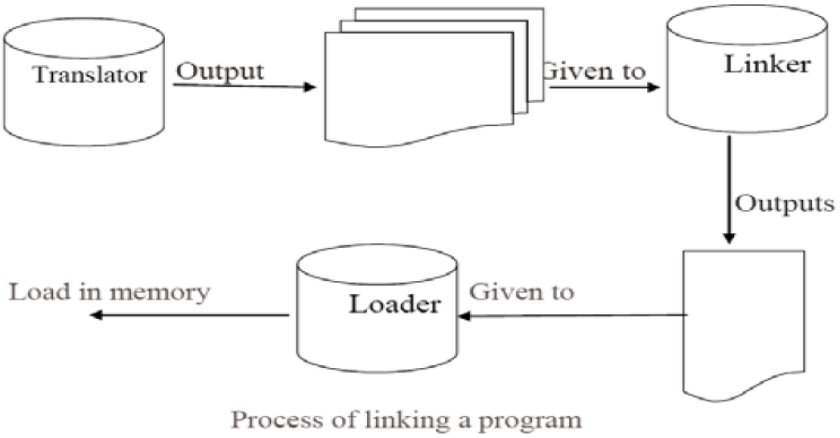
1. Translation of the program (done by assembler or compiler)
2. Linking of the program with all other programs which are needed for execution. This also involves preparation of a program called load module.
3. Loading of the load module prepared by linker to some specified memory location.

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary program. The "binary program also contains some necessary information about allocation and relocation. The loader then loads this program into memory for execution purpose.

Various tasks of Linker:

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.
2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.

3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.



Process of linking a program