Department of Computer Science and Engg.

System Programming And System Administration

Sem-6th

Section-A (notes)

System programming (or **systems programming**) is the activity of programming computer system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly (e.g. word processor), whereas systems programming aims to produce software and software platforms which provide services to other software, are performance constrained, or both (e.g. operating systems, computational science applications, game engines and AAA video games, industrial automation, and software as a service applications

WHAT IS SYSTEM PROGRAMMING?

System programming is the activity of programming system software.

The following attributes characterize systems programming:

- The programmer will make assumptions about the hardware and other properties of the system that the program runs on, and will often exploit those properties, for example by using an algorithm that is known to be efficient when used with specific hardware.
- Usually a low-level programming language or programming language dialect is used that:
    - can operate in resource-constrained environments
    - is very efficient and has little runtime overhead
    - has a small runtime library, or none at all
    - allows for direct and "raw" control over memory access and control flow
    - lets the programmer write parts of the program directly in assembly language
- Often systems programs cannot be run in a debugger. Running the program in a simulated environment can sometimes be used to reduce this problem.

Systems programming is sufficiently different from application programming that programmers tend to specialize in one or the other.

In system programming, often limited programming facilities are available. The use of automatic garbage collection is not common and debugging is sometimes hard to do. The runtime library, if available at all, is usually far less powerful, and does less error checking. Because of those limitations, monitoring and logging are often used; operating systems may have extremely elaborated logging subsystems. Implementing certain parts in operating systems and networking requires systems programming, for example implementing Paging (Virtual Memory) or a device driver for an operating system.

Difference b/t System Programming & Application Programming:

Application programming aims to produce software which provides services to the user (e.g. word processor), whereas Systems programming aims to produce software which provides services to the computer hardware.

Components of Programming system are:-

1 Assemblers
2 Loaders
3 Macros
4 Compilers
5 Linkers

ASSEMBLERS:

At one time, the computer programmer had a basic machine that interpret through hardware certain fundamental instructions. He would program this computer by writing a series of ones and zeros (machine language), place them into the memory of the machine, and press a button, whereupon the computer would start to interpret them as instructions. Programmers find it difficult to write or read programs in machine language, In their hunt for a more convenient language they began to use a mnemonic(symbol) for each machine instruction, which they could subsequently translate into machine language. Such a mnemonic machine language is now called an **assembly language**. Programs known as **assemblers** were written to automate/computerize the translation of assembly language into machine language. The input to an assembler program is called the **source program**; the output is a machine language translation object **program**).
Assembly language is typically used in a system's boot code, (BIOS on IBM compatible PC systems and CP/M), the low-level code that initializes and tests the system hardware prior to booting the OS, and is often stored in ROM. Some compilers translate high-level languages into assembly first before fully compiling, allowing the assembly code to be viewed for debugging and optimization purposes. Relatively low-level languages, such as C, allow the programmer to embed assembly language directly in the source code. Programs using such facilities, such as the Linux kernel, can then construct abstractions using different assembly language on each hardware platform. The system's portable code can then use these processor-specific components through a uniform interface

LOADERS:

Once the assembler produces an object program, that program must be placed into memory and executed. It is the purpose of the loader to assure that object programs are placed in memory in an executable form. The assembler could place the object program directly in a memory and transfer control to it, thereby causing the machine language program to be executed. However this would waste memory by leaving the assembler in memory while the user’s program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time.

To overcome the problem of wasted translation time and wasted memory, system programmers developed another component, called the Loader. Loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary storage device and a loader is placed in memory. The loader places into memory the machine language version of the user’s program & transfers control to it. Since the loader program is much smaller then the assembler, this makes more memory available to the user’s program.

Linker:

In computing, a **linker** or **link editor** is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.

A simpler version that writes its output directly to memory is called the *loader*, though loading is typically considered a separate process

Computer programs typically comprise several parts or modules; these parts/modules need not all be contained within a single object file, and in such cases refer to each other by means of symbols. Typically, an object file can contain three kinds of symbols:

- defined "external" symbols, sometimes called "public" or "entry" symbols, which allow it to be called by other modules,
- undefined "external" symbols, which reference other modules where these symbols are defined, and
- local symbols, used internally within the object file to facilitate relocation.

For most compilers, each object file is the result of compiling one input source code file. When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along.

Linkers can take objects from a collection called a *library*. Some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default.

The linker also takes care of arranging the objects in a program's address space. This may involve *relocating* code that assumes a specific base address to another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads and stores.

The executable output by the linker may need another relocation pass when it is finally loaded into memory (just before execution). This pass is usually omitted on hardware offering virtual memory: every program is put into its own address space, so there is no conflict even if all programs load at the same base address. This pass may also be omitted if the executable is a position independent executable.

In computer science, a linker is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program.

Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program. The program as a whole refers to these separately-compiled object files using symbols. The linker combines these separate files into a single, unified program; resolving the symbolic references as it goes along.

Dynamic linking is a similar process, available on many operating systems, which postpones the resolution of some symbols until the program is executed. When the program is run, these dynamic link libraries are loaded as well. Dynamic linking does not require a linker.
The linker bundled with most Linux systems is called ld; see our **ld** documentation page for more information.

Macro:

In computer science is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to a replacement output sequence (also often a sequence of characters) according to a defined procedure. The mapping process that instantiates (transforms) a macro use into a specific sequence is known as *macro expansion*. A facility for writing macros may be provided as part of a software application or as a part of a programming language. In the former case, macros are used to make tasks using the application less repetitive. In the latter case, they are a tool that allows a programmer to enable code reuse or even to design domain-specific languages.

Macros are used to make a sequence of computing instructions available to the programmer as a single program statement, making the programming task less tedious and less error-prone.[1][2] (Thus, they are called "macros" because a "big" block of code can be expanded from a "small" sequence of characters.) Macros often allow positional or keyword parameters that dictate what the conditional assembler program generates and have been used to create entire programs or program suites according to such variables as operating system, platform or other factors. The term derives from "macro instruction", and such expansions were originally used in generating assembly language code

In reference to computers, a **macro** (which stands for "macroinstruction") is a programmable pattern which translates a certain sequence of input into a preset sequence of output. **Macros** can be used to make certain tasks less repetitive by representing a complicated sequence of keystrokes, mouse movements, commands, or other types of input.
In computer programming, **macros** are a tool which allow a developer to re-use code. For instance, in the C programming language, this is an example of a simple **macro** definition which incorporates arguments:

```
#define square(x) ((x) * (x))
```
After being defined like this, our macro can be used in the code body to find the square of a number. When the code is preprocessed before compilation, the macro will be expanded each time it occurs. For instance, using our macro like this:

```
int num = square(5);
```

is the same as writing:

```
int num = ((5) * (5));
```

... which will declare an integer-type variable named **num**, and set its value to **25**.
Note that a **macro** is not the same as a function: functions require special instructions and computational overhead to safely pass arguments and return values; a macro is a way to repeat frequently-used lines of code. In some simple cases, using a macro instead of a function can improve performance by requiring fewer instructions and system resources to execute.

Compiler:

A **compiler** is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. The most common reason for converting source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. More generally, compilers are a specific type of translator.

A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a source-to-source compiler or transpiler. A language rewriter is usually a program that translates the form of expressions without a change of language. The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser. A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (syntax-directed translation), code generation, and code optimization. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest significant effort to ensure compiler correctness

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

Interpreter:

In computer science, an ***interpreter*** is a computer program that directly executes, i.e. *performs*, instructions written in a programming or scripting language, without previously compiling them into a machine language program. An interpreter generally uses one of the following strategies for program execution:

1. parse the source code and perform its behavior directly.
2. translate source code into some efficient intermediate representation and immediately execute this.
3. explicitly execute stored precompiled code[1] made by a compiler which is part of the interpreter system.

Early versions of Lisp programming language and Dartmouth BASIC would be examples of the first type. Perl, Python, MATLAB, and Ruby are examples of the second, while UCSD Pascal is an example of the third type. Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and/or compiler (for JIT systems). Some systems, such as Smalltalk and contemporary versions of BASIC and Java may also combine two and three.[2] Interpreters of various types have also been constructed for many languages traditionally associated with compilation, such as Algol, Fortran, Cobol and C/C++.

While interpretation and compilation are the two main means by which programming languages are implemented, they are not mutually exclusive, as most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" signify that the canonical implementation of that language is an interpreter or a compiler, respectively. A high level language is ideally an abstraction independent of particular implementations

Difference Between Interpreter and Compiler

We generally write a computer program using a high-level language. A high-level language is one which is understandable by us humans. It contains words and phrases from the English (or other) language. But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the machine code. A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished my compilers and interpreters. Hence, a compiler or an interpreter is a program that converts program written in high-level language into machine code understood by the computer.

The difference between an interpreter and a compiler is given below:

Interpreter	**Compiler**
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

A compiler can broadly be divided into two phases based on the way they compile.

Analysis Phase

Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

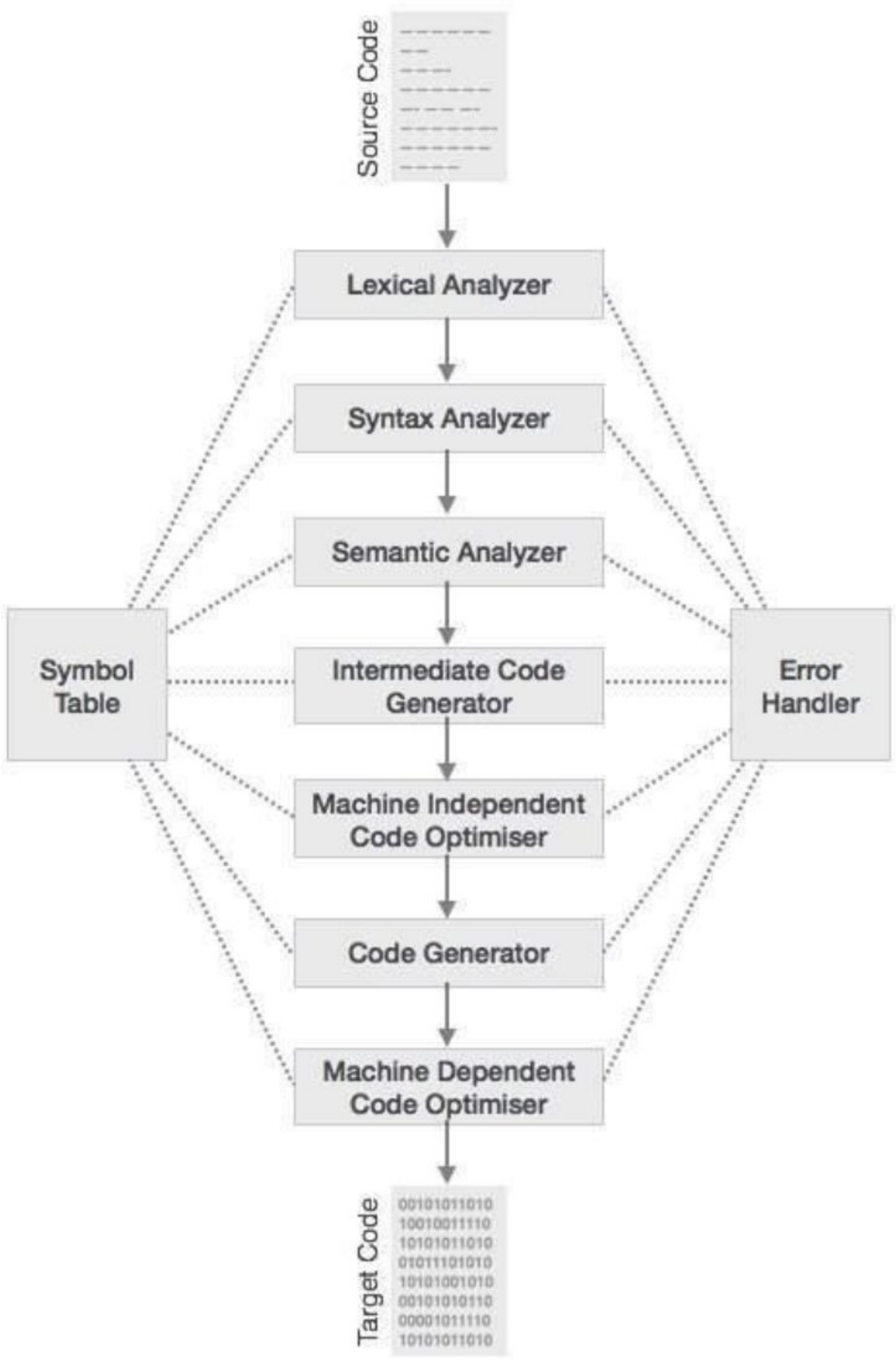
Synthesis Phase

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Tokens:

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by

|tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intvalue;

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Software Tools:

A **programming tool** or **software development tool** is a computer program that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs, that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object. The ability to use a variety of tools productively is one hallmark of a skilled software engineer.

The most basic tools are a source code editor and a compiler or interpreter, which are used ubiquitously and continuously. Other tools are used more or less depending on the language, development methodology, and individual engineer, and are often used for a discrete task, like a debugger or profiler. Tools may be discrete programs, executed separately – often from the command line – or may be parts of a single large program, called an integrated development environment (IDE). In many cases, particularly for simpler use, simple ad hoc techniques are used instead of a tool, such as print debugging instead of using a debugger, manual timing (of overall program or section of code) instead of a profiler, or tracking bugs in a text file or spreadsheet instead of a bug tracking system.

Text editor:

A **text editor** is a type of program used for editing plain text files. Such programs are sometimes known as "**notepad**" software, following the Microsoft Notepad. Text editors are provided with operating systems and software development packages, and can be used to change configuration files, documentation files and programming language source code

There are important differences between plain text files created by a text editor and document files created by word processors such as Microsoft Word or WordPerfect.

- A plain text file uses a character encoding such as UTF-8 or ASCII to represent numbers, letters, and symbols. The only non-printing characters in the file that can be used to format the text are newline, tab, and formfeed. Plain text files are often displayed using a monospace font so horizontal alignment and columnar formatting is sometimes done using space characters.
- Word processor documents are generally stored in a binary format to allow for localization and formatted text, such as boldface, italics and multiple fonts, and to be structured into columns and tables.
- Although they are often viewed with formatting, documents using markup languages are stored in plain text files that contain a combination of human-readable text and markup tags. For example, web pages are plain text with HTML tags to achieve formatting when

rendered by a web browser. Many web pages also contain embedded JavaScript that is interpreted by the browser.

Word processors were developed to allow formatting of text for presentation on a printed page, while text produced by text editors is generally used for other purposes, such as input data for a computer program.

Program generator:

Software program that enables an individual to create a program of their own with less effort and programming knowledge. With a program generator a user may only be required to specify the steps or rules required for his or her program and not need to write any code or very little code. Some great examples of a program generator are Adventure Maker, Alice, Stagecast Creator, and YoYo Games

APPLICATIONS:
Beginner-friendly mouse operation.

Complicated environmental configuration not required.

Just start up and start creating sample software immediately. Since the module library provided even defines MCU peripheral circuits, you can operate your MCU as soon as software combination is complete.
The Sample Application Program Generator & Organizer system is C-compliant.
The combination of supported program modules enables highly-flexible software creation.
Since the source file of the created software is output, it can also be used as a base for development.
As control of the Sample Application Program Generator & Organizer system utilizes only the comment line, ROM/RAM cannot be increased.
User programs can be registered in a library at the function level using the GUI..
The program module, like sample software, is provided free of charge.

Debug Monitors:

- A debug monitor, is a tool that helps to find and reduce them number of bugs and defects in a computer program or any electrical device within or attached to the computer in order to make it act the way it should.
  For example, when an armored car drives up to a bank and the guards have to transfer money from the truck to the bank, there are special guards that stand watch to make sure no one tries to rob them thus making the transaction go smoothly. Those guards could be the debug monitors in the computer industry.
  If the debugging monitor locates a bug or defect in any of the equipment, it will first try to reproduce the problem which will allow a programmer to view each string (in a program) that was within the bug or defect range and try to fix it.

PROGRAMMING ENVIRONMENT:

The programming environment is the set of processes and programming tools used to create the program or software product. An integrated development environment is one in which the processes and tools are coordinated to provide developers an convenient view of the development process (or at least the processes of writing code,testing it, and packaging it for use). An example of an IDE product is Microsoft's Visual Studio .NET. And Oracle JDeveloper 10g and Eclipse (an IBM product) for Java development. An integrated development environment (IDE) is a programming environment that has been packaged as an application program, typically consisting of a code editor, a compiler, a debugger, and a graphical user interface (GUI) builder. The IDE may be a standalone application or may be included as part of one or more existing and compatible applications. IDEs provide a user-friendly framework for many modern programming languages, such as Visual Basic, Java etc. IDEs for developing HTML applications are among the most commonly used. For example, many people designing Web sites today use an IDE (such as H S HomeSite, DreamWeaver, or FrontPage) for Web site development that automates many of the tasks involved.

Incremental compiler:

An **incremental compiler** is a kind of incremental computation applied to the field of compilation. Quite naturally, whereas ordinary compilers make so called **clean build**, that is, (re)build all program modules, incremental compiler recompiles only those portions of a program that have been modified.
Incremental compilers for imperative language compiling

- The PECAN Programming Environment Generator was an incremental compiler, developed by Steven P. Reiss in the early 1980s.[3][4]
- GNU Compiler Collection has branched off[5] its development with the IncrementalCompiler project, concentrating in providing C/C++ with a fast incremental compiler
- The Eclipse platform has a Java incremental compiler included as a part of the *Java Development Tools* project[6]
- The IBM VisualAge C++ compiler 4.0 is an incremental compiler for C++
- Embarcadero Delphi, previously Borland Delphi
- The .NET Compiler Platform (codename Roslyn) is an open source incremental compiler for C# and Visual Basic .NET, and is the default compiler from Visual Studio 2015 onwards

Incremental compilers in interactive programming environments and runtime systems

- Poplog (its core language POP-11 and its predecessor POP-2)
- Some versions of Lisp:
    - Steel Bank Common Lisp
    - Carnegie Mellon University Common Lisp
    - Scieneer Common Lisp
    - GNU CLISP

    - Franz Allegro Common Lisp
- Some versions of Scheme:
    - Ikarus
- Most versions of Prolog:
    - SWI-Prolog, Yap Prolog, XSB, Ciao
- Versions of ML:
    - Standard ML of New Jersey (Bell Labs' headquarters resides in New Jersey)
    - Poplog ML
- Forth
    - Kerridge KCML
- Ceylon of Red-Hat

Assembler:

Convert mnemonic operation codes to their machine language equivalents
Convert symbolic operands to their equivalent machine addresses
Build the machine instructions in the proper format
Convert the data constants to internal machine representations

Two Pass Assembler

Pass 1

Assign addresses to all statements in the program
Save the values assigned to all labels for use in Pass 2
Perform some processing of assembler directives

Pass 2

Assemble instructions
Generate data values defined by BYTE, WORD
Perform processing of assembler directives not done in

General loader scheme:

In general loader scheme the object code is generated first by the translator using source program. Then the object modules are used by the loader to convert it into executable form and load it into the assigned memory location. The loader program occupies main memory

Compile and go system

In computer programming, a **compile and go system**, **compile, load, and go system**, **assemble and go system**, or **load and go system**[1][2][3] is a programming language processor in which the compilation, assembly, or link steps are not separated from program execution. The intermediate forms of the program are generally kept in primary memory, and not saved to the file system.

Examples of compile-and-go systems are WATFOR, PL/C, and Dartmouth BASIC.[3]

An example of a load-and-go system is the OS/360 loader, which performed many of the functions of the Linkage Editor, but placed the linked program in memory rather than creating an executable on disk.

Compile and go systems differ from interpreters, which either directly execute source code or execute an intermediate representation.

Absolute Loader:

An input program is translated by the assembler from a source program into an object program (on tape), which is loaded into memory by the **loader** for execution. There are several advantages to this **scheme** (in addition to the fact that it is necessary.

An absolute loader is the simplest of loaders. Its function is simply to take the output of the assembler and load it into memory. The output of the assembler can be stored on any machine-readable form of storage, but most commonly it is stored on punched cards or magnetic tape, disk, or drum.

Absolute load format:

The input to an absolute loader is generally not quite in machine language. Machine language is really defined only in terms of a program in memory. The input is in *loader format*, a form of representation for a program which is so close to machine language that the loader need do nothing more than store the program in the appropriate memory locations.

How does the loader know which addresses the instructions (and data) should be loaded into? The assembler is told by ORIG statements written by the programmer, and this information must be passed on to the loader by the assembler. Thus the input to the loader must consist of two pieces of information: the instruction or data to be loaded, and the address where it should be loaded.

In addition, the loader will need to know when to stop loading, so we need some kind of an end-of-file marker at the end of the loader input. When the loader stops, it should transfer control to the newly loaded program. This means that a starting address is also needed.

The absolute loader is the simplest and quickest of the two. The loader loads the file into memory at the location specified by the beginning portion (header) of the file, then passes control to the program. If the memory space specified by the header is currently in use, execution cannot proceed, and the user must wait until the requested memory becomes free.

Rellocating Loader:

The relocating loader will load the program anywhere in memory, altering the various addresses as required to ensure correct referencing. The decision as to where in memory the program is placed is done by the Operating System, not the programs header file. This is obviously more efficient, but introduces a slight overhead in terms of a small delay whilst all the relative offsets are calculated. The relocating loader can only relocate code that has been produced by a linker capable of producing relative code. A loader is unnecessary for interpreted languages, as the executable code is built up into the memory of the computer.

DIRECT LINKAGE 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. It is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program– loading process can be extremely time consuming. Though smaller than the assembler, the loader absorbs a considerable amount of space– Dividing the loading process into two separate programs a binder and a module loader can solve these problems.

BINDERS:

In file management, a **binder** is a software utility that combines two or more files into a single file. ... When the user clicks on the host file, the embedded files are automatically decompressed and, if they contain an application (that is, if the package includes an executable file), the application is run.

One of the most popular binders was Microsoft's "Pack and Go" feature in PowerPoint. It allowed users to save their slide shows (.ppt files), graphics (.gif, .jpg, or .bmp files), and sound (.mid, .wav or .au files) in one file under one name. Unfortunately, a binder file can also be used to hide malicious code such a Trojan horse, so many software vendors, including Microsoft, have discontinued its use.

OVERLAYS:

Overlay (**programming**) In a general computing sense, overlaying means "the process of transferring a block of **program** code or other data into internal memory, replacing what is already stored". Overlaying is a **programming** method that allows programs to be larger than the computer's main memory. Constructing an overlay program involves manually dividing a program into self-contained object code blocks called **overlays** laid out in a tree structure. *Sibling* segments, those at the same depth level, share the same memory, called *overlay region* or *destination region*. An overlay manager, either part of the operating system or part of the overlay program, loads the required overlay from external memory into its destination region when it is needed. Often linkers provide support for overlays.

most business applications are intended to run on platforms with virtual memory. A developer on such a platform can design a program as if the memory constraint does not exist unless the program's working set exceeds the available physical memory. Most importantly, the architect can focus on the problem being solved without the added design difficulty of forcing the processing into steps constrained by the overlay size. Thus, the designer can use higher-level programming languages that do not allow the programmer much control over size (e.g. Java, C++, Smalltalk).

Still, overlays remain useful in embedded systems.[4] Some low-cost processors used in embedded systems do not provide a memory management unit (MMU). In addition many embedded systems are real-time systems and overlays provide more determinate response-time than paging. For example, the Space Shuttle *Primary Avionics System Software (PASS)* uses programmed overlays.

In the home computer era overlays were popular because the operating system and many of the computer systems it ran on lacked virtual memory and had very little RAM by current standards — the original IBM PC had between 16K and 64K depending on configuration. Overlays were a popular technique in Commodore BASIC to load graphics screens. In order to detect when an overlay was already loaded, a flag variable could be used