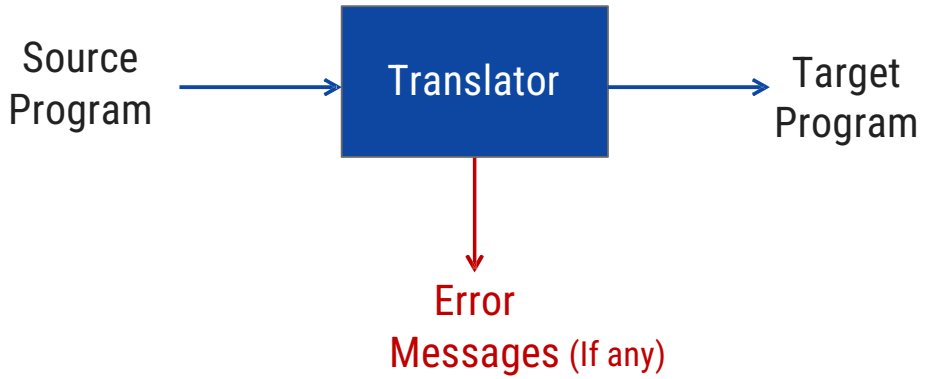
Compiler Design

Unit-1 Introduction

Introduction to translators- Assembler, Compiler, Interpreter, Difference between Compiler and Interpreter, Linker, Loader , one pass compiler, multi pass compiler, cross compiler , The components of Compiler, Stages of Compiler: Front end, Back end, Qualities of Good Compiler

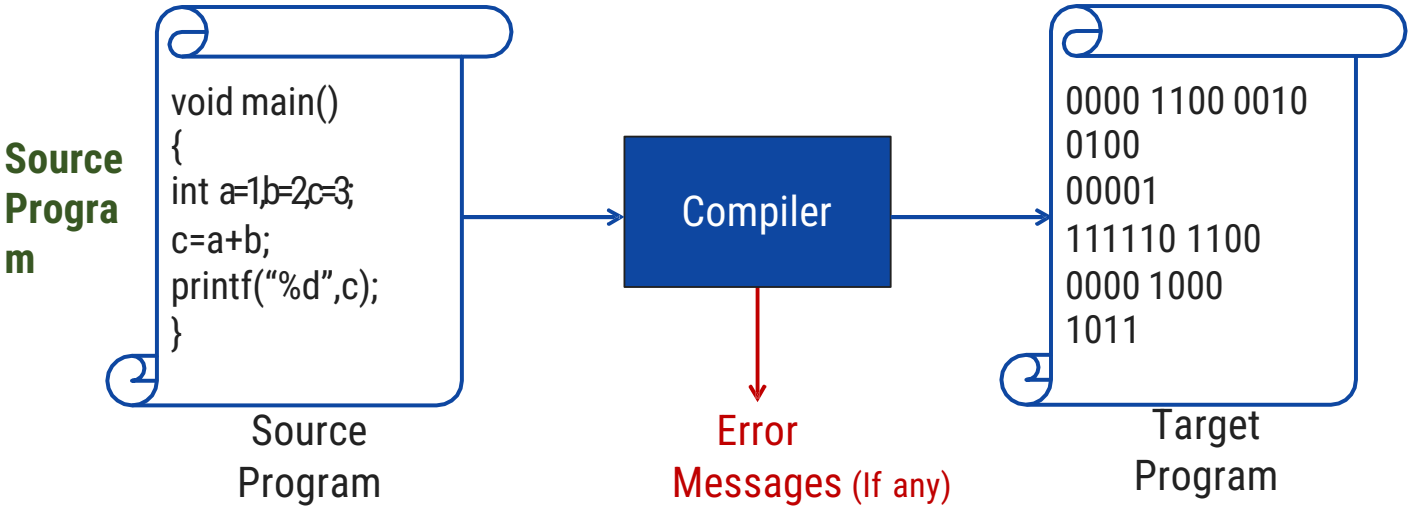
Translator

- A translator is a program that **takes one form of program as input** and **converts it into another form.**
- Types of translators are:
  1. Compiler
  2. Interpreter
  3. Assembler



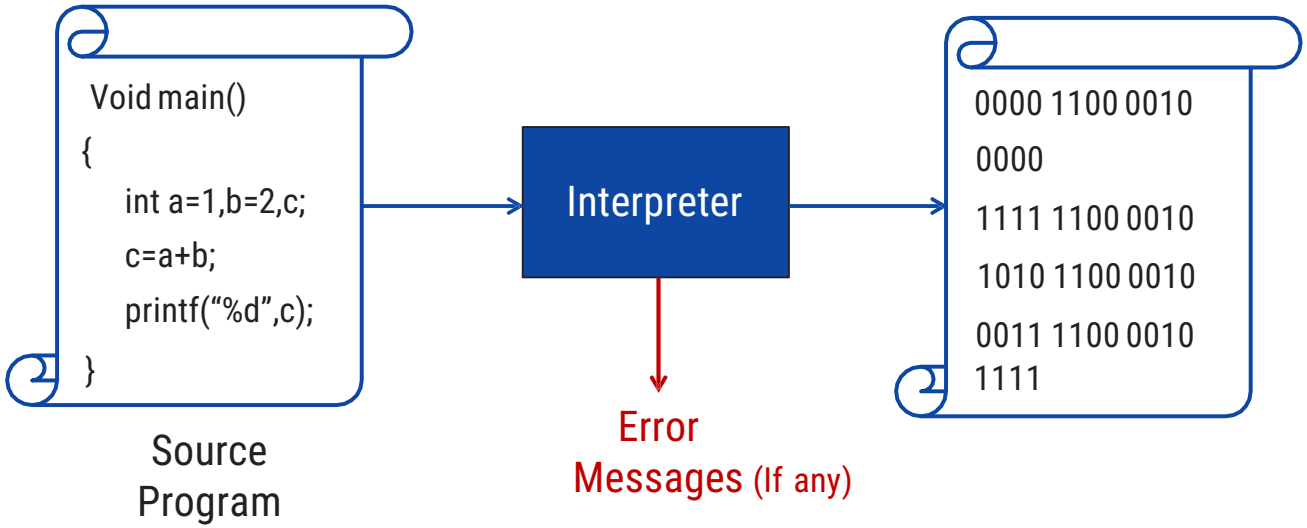
Compiler

- A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language



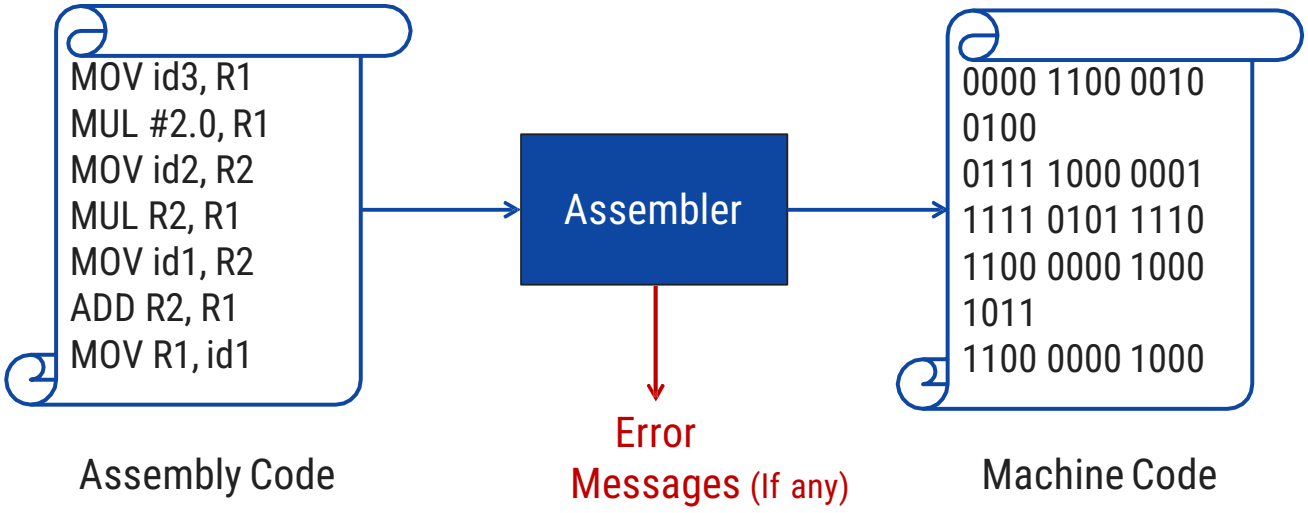
Interpreter

- Interpreter is also program that reads a program written in source language and translates it into an equivalent program in target language line by line.



Assembler

- Assembler is a translator which takes the assembly code as an input and generates the machine code as an output.

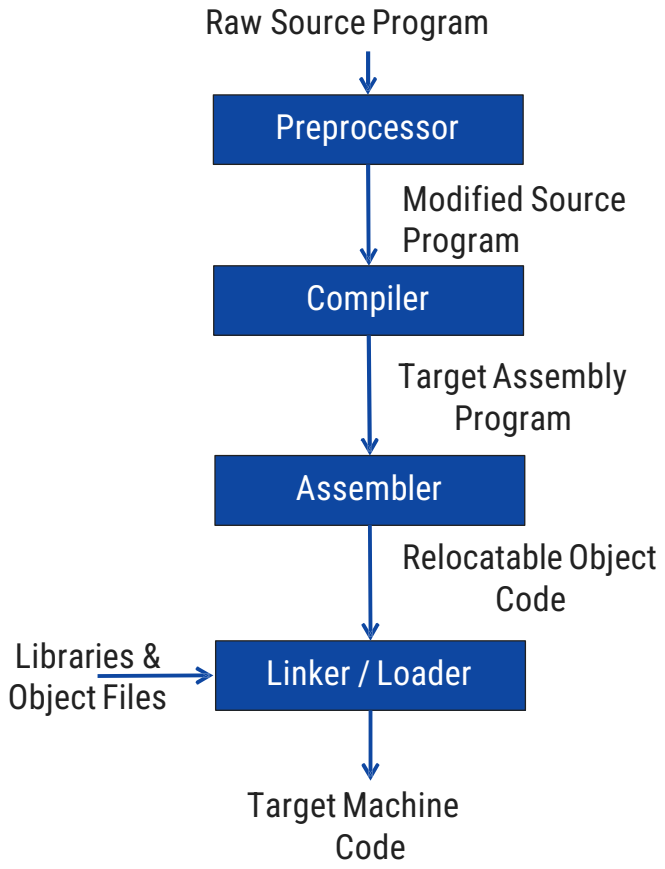


Difference between compiler & interpreter

Compiler	Interpreter
Scans the entire program and translates it as a whole into machine code.	It translates program's one statement at a time.
It generates intermediate code.	It does not generate intermediate code.
An error is displayed after entire program is checked.	An error is displayed for every instruction interpreted if any.
Memory requirement is more.	Memory requirement is less.
Example: C compiler	Example: Basic, Python, Ruby

Language Processing System

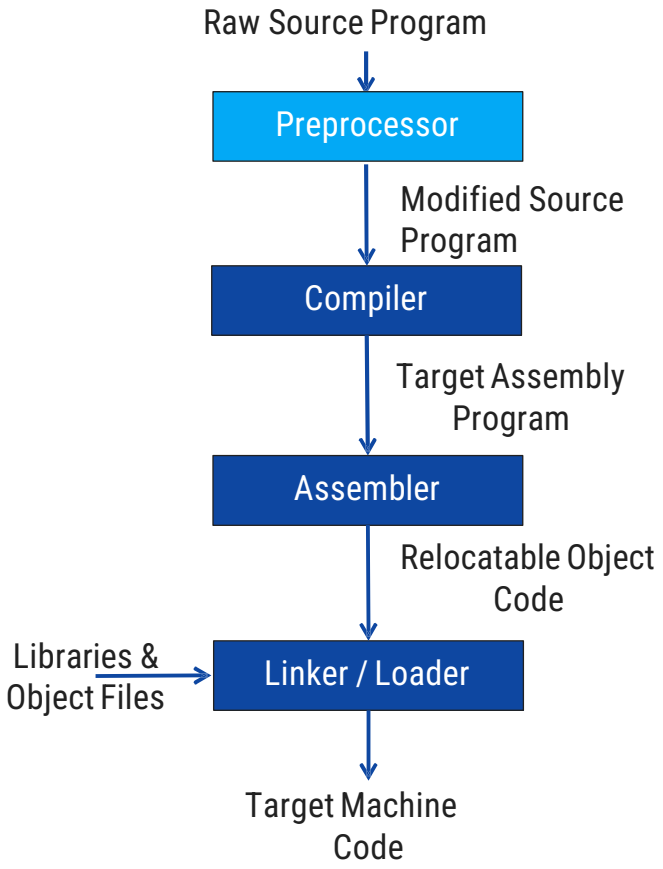
- In addition to compiler, many other system programs are required to generate absolute machine code.
- These system programs are:
  - ➥ Preprocessor
  - ➥ Assembler
  - ➥ Linker
  - ➥ Loader



Language Processing System

- **Some of the task performed by preprocessor:**

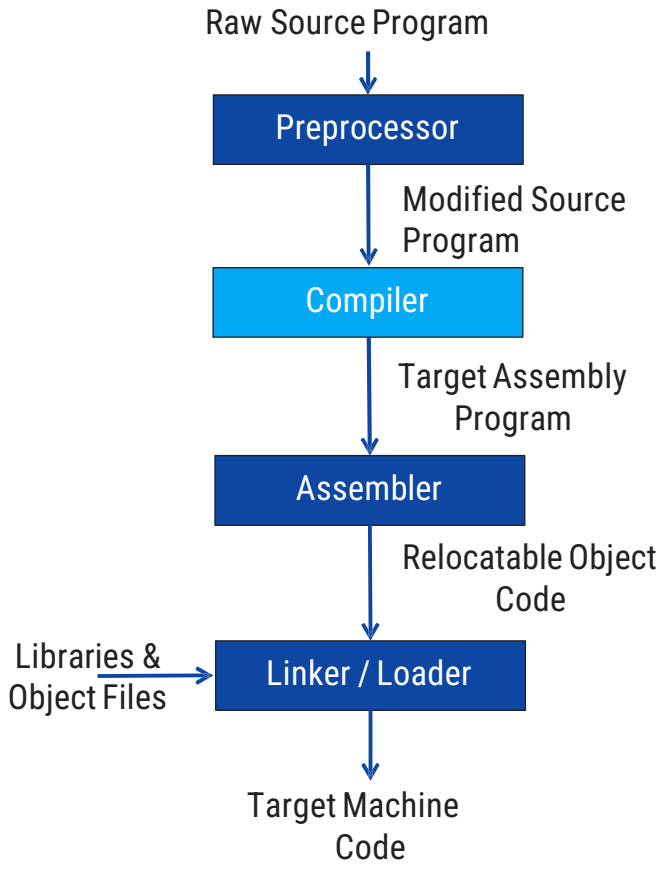
1. **Macro processing**: Allows user to define macros.
   Ex: #define PI 3.14159265358979323846
2. **File inclusion**: A preprocessor may include the header file into the program. Ex: #include<stdio.h>
3. **Rational preprocessor**: It provides built in macro for construct like while statement or if statement.
4. **Language extensions**: Add capabilities to the language by using built-in macros.
   - Ex: the language equal is a database query language embedded in C. Statement beginning with ## are taken by preprocessor to be database access, statement unrelated to C and translated into procedure call on routines that perform the database access.



Language Processing System

Compiler

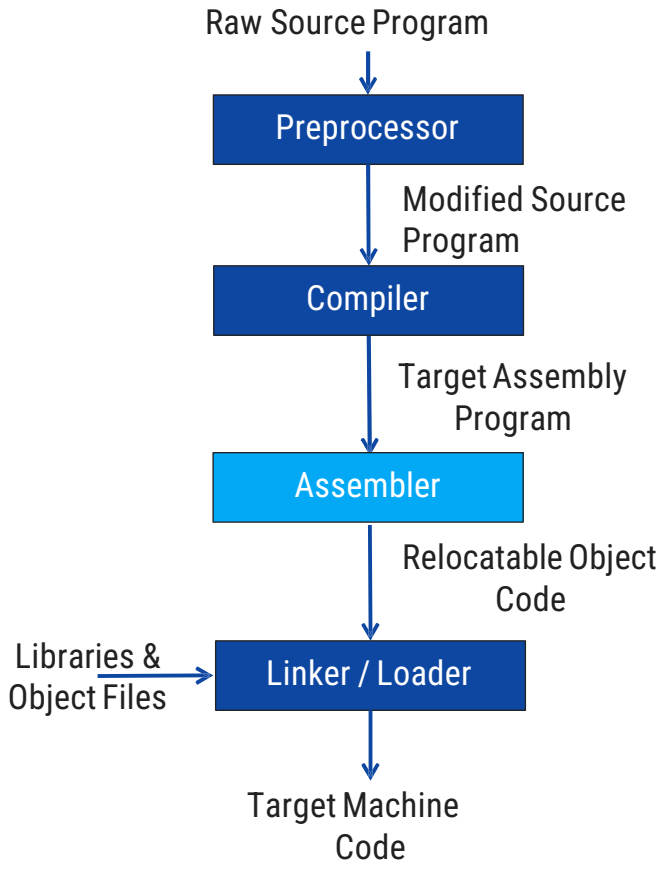
- A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.



Language Processing System

Assembler

- Assembler is a translator which takes the assembly program (mnemonic) as an input and generates the machine code as an output.



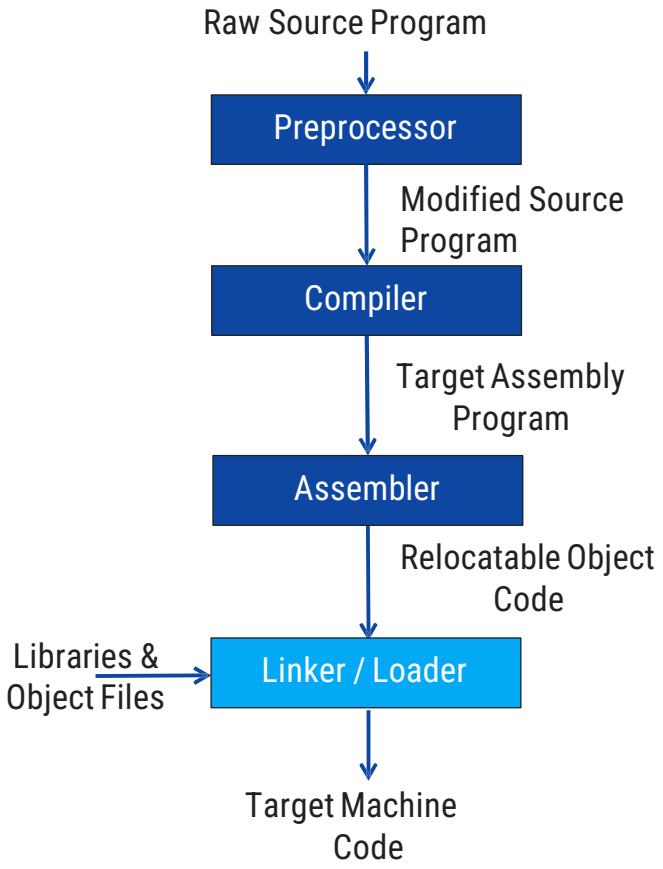
Language Processing System

Linker

- Linker makes a single program from a several files of relocatable machine code.
- These files may have been the result of several different compilation, and one or more library

Loader

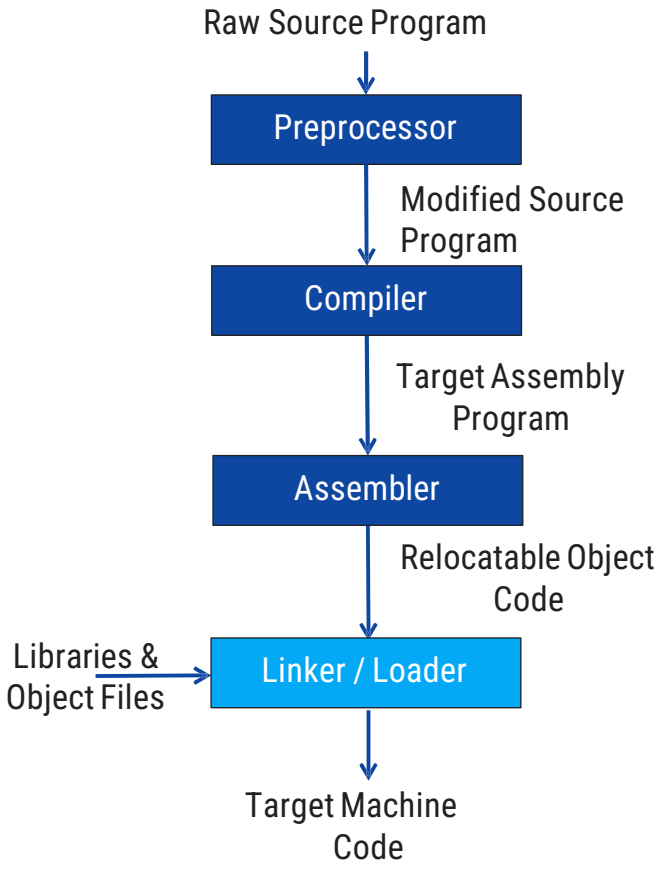
- The process of loading consists of:
  - Taking relocatable machine code
  - Altering the relocatable address
    - Placing the altered instructions and data in memory at the proper location.



Language Processing System

- The linker and loader are essential components in the compilation process.
- The linker is responsible for combining multiple object files generated by the compiler into a single executable file or library. It resolves symbols and references between different object files, ensuring that functions and variables are correctly linked together.
- For example, suppose you have two C source files, main.c and functions.c, which need to be compiled separately:

```
gcc -c main.c
gcc -c functions.c
```

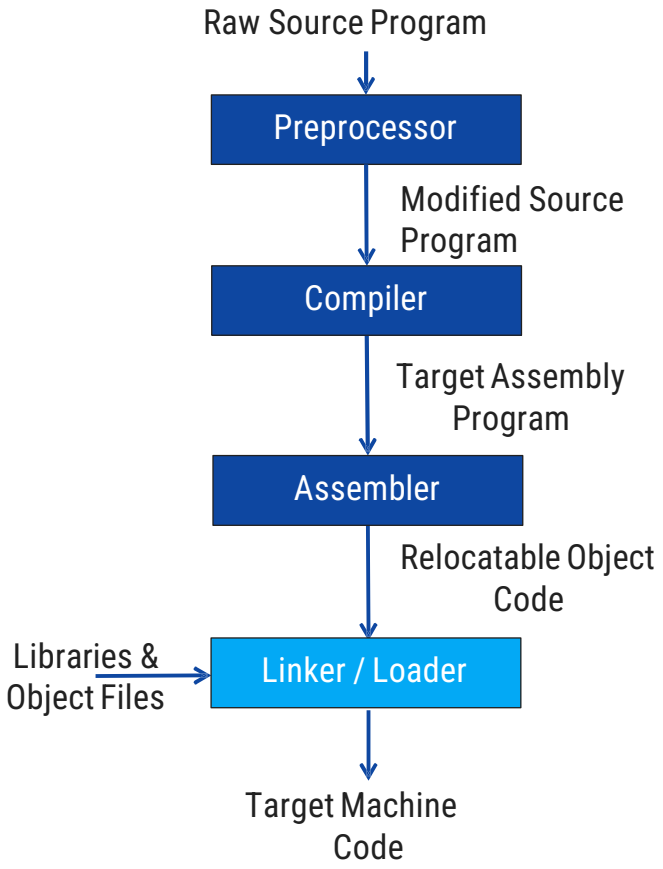


Language Processing System

- This will generate two object files: main.o and functions.o. The linker comes into play when you want to create an executable from these object files:

```
gcc main.o functions.o -o program
```

- Here, the linker (ld) combines both object files (main.o and functions.o) to create an executable called program. It resolves any unresolved symbols or references between them.

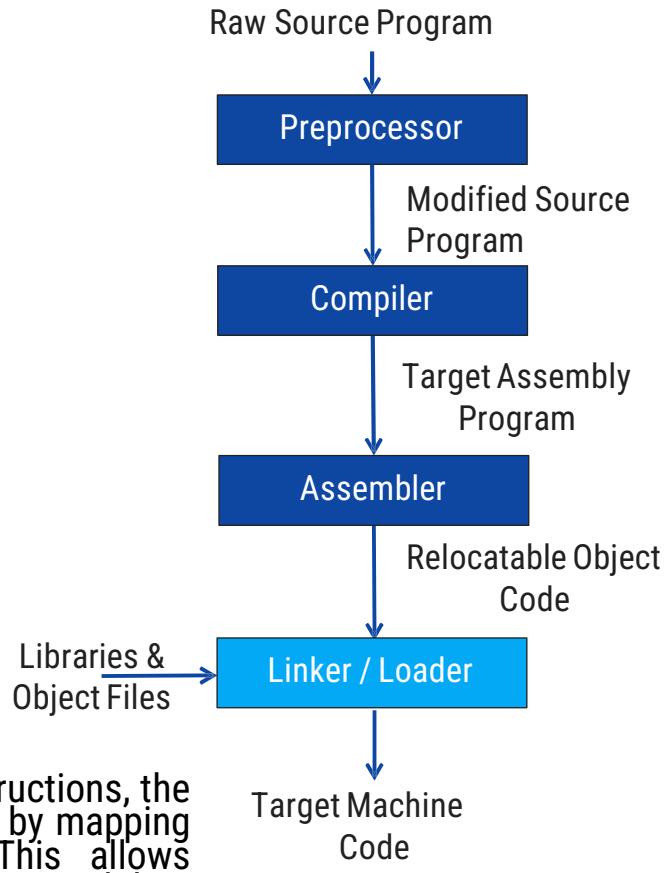


Language Processing System

- Loader: After linking, the resulting executable file needs to be loaded into memory for execution. This is where the loader comes in. The loader is responsible for allocating memory space for the program, resolving dynamic dependencies (if any), and preparing it for execution.
- Continuing with our previous example, once we have our executable file program, we can run it:

```
./program
```

  - At this point, before executing the program instructions, the operating system's loader loads it into memory by mapping its sections/data structures appropriately. This allows direct access to resources such as code segments and data segments during runtime.



Pass structure

- One complete scan of a source program is called pass.
- Pass includes **reading an input file** and **writing to the output** file.
- In a single pass compiler analysis of source statement is immediately followed by synthesis of equivalent target statement.
- While in a two pass compiler intermediate code is generated between analysis and synthesis phase.
- It is difficult to compile the source program into single pass due to: **forward reference**

Pass structure

- **Forward reference:** A forward reference of a program entity is a **reference to the entity which precedes its definition** in the program.
- This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available.
- It leads to multi pass model of compilation.

```
int x = y + 5;
int y = 10;
```

- In this case, y is being referenced before it has been declared. This results in a forward reference error because the compiler cannot determine what value y holds at that point.
- To handle forward references, compilers typically perform multiple passes over the source code. In earlier passes, they identify and record all identifiers and symbols encountered. In subsequent passes, they resolve these references by looking up their definitions.
- Overall, handling forward references correctly is crucial for ensuring proper linking and execution of programs during compilation.

Pass structure

- **Forward reference:** A forward reference of a program entity is a **reference to the entity which precedes its definition** in the program.
- This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available.
- It leads to multi pass model of compilation.

Pass I:

- Perform analysis of the source program and note relevant information.

Pass II:

- In Pass II: Generate target code using information noted in pass I.

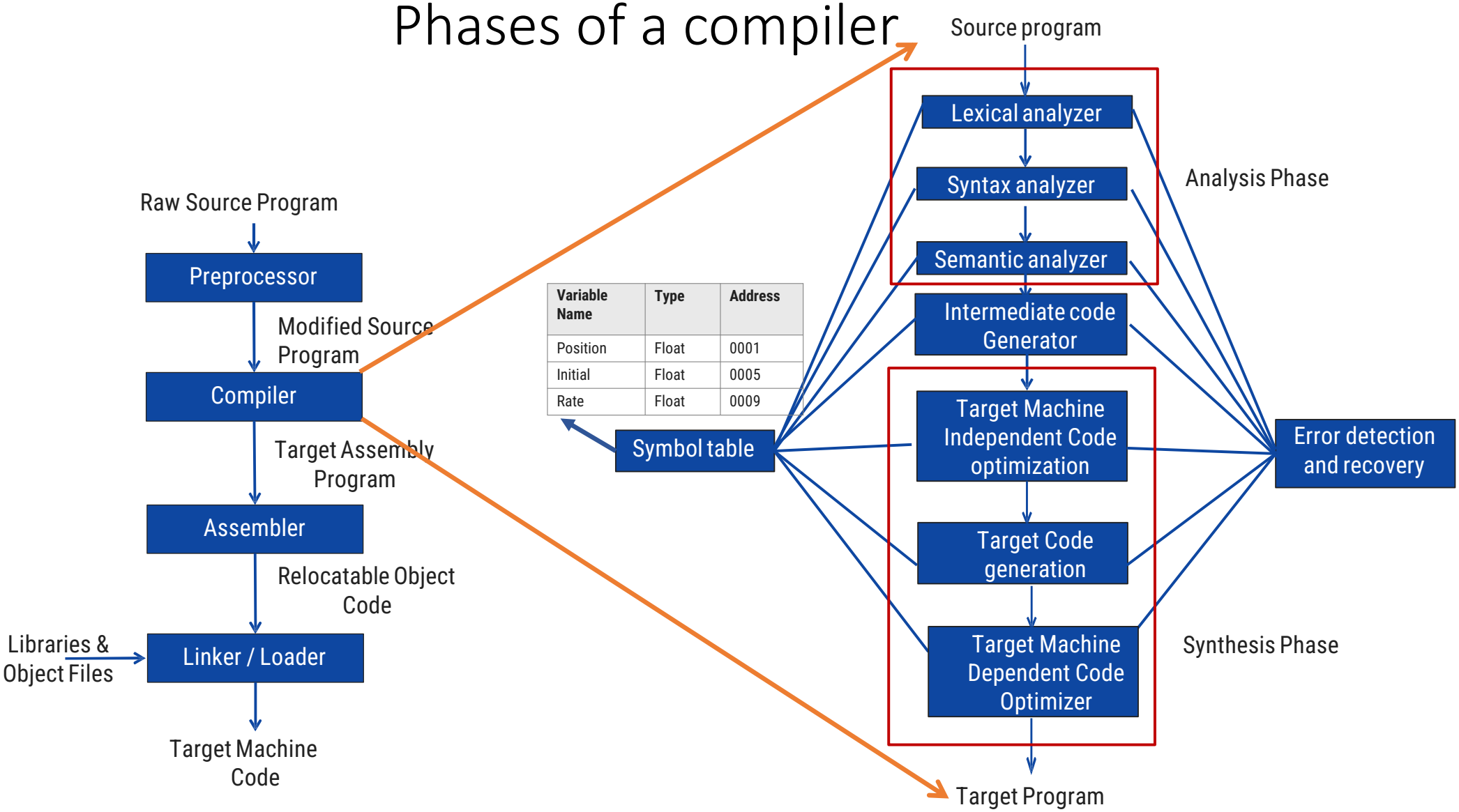
Pass structure

- Effect of reducing the number of passes:
  - It is desirable to have a few passes, because **it takes time to read and write** intermediate file.
  - If we group **several phases into one pass** then **memory requirement** may be **large**.

Types of compiler

1. One pass compiler
   - It is a type of compiler that compiles whole process in one-pass.
2. Two pass compiler
   - It is a type of compiler that compiles whole process in two-pass.
   - It generates intermediate code.
3. Incremental compiler
   - The compiler which compiles only the changed line from the source code and update the object code.
4. Native code compiler
   - The compiler used to compile a source code for a same type of platform only.
5. Cross compiler
   - The compiler used to compile a source code for a different kinds platform.

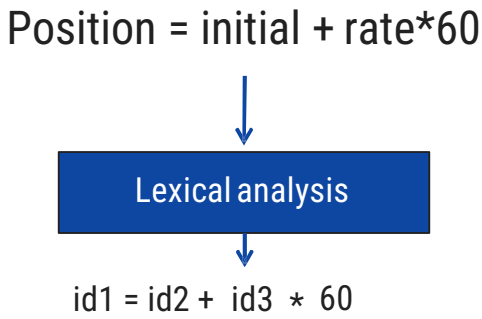
Phases of a compiler



Variable Name	Type	Address
Position	Float	0001
Initial	Float	0005
Rate	Float	0009

Lexical analysis

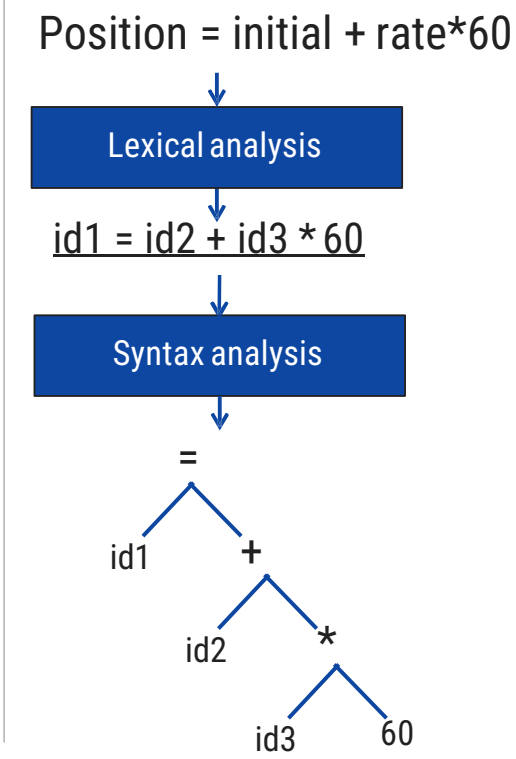
- Lexical Analysis is also called **linear analysis** or **scanning**.
- Lexical Analyzer divides the given source statement into the
- **tokens**.
- Ex: Position = initial + rate * 60 would be grouped into the following tokens:
  - Position (identifier)
  - = (Assignment symbol) initial (identifier)
  - + (Plus symbol) rate (identifier)
  - * (Multiplication symbol) 60 (Number)



Variable Name	Type	Address
Position	Float	0001
Initial	Float	0005
Rate	Float	0009

Syntax analysis

- Syntax Analysis is also called **Parsing** or **Hierarchical Analysis**.
- The syntax analyzer checks each line of the code and spots every tiny mistake.
- If code is error free then syntax analyzer generates the tree.

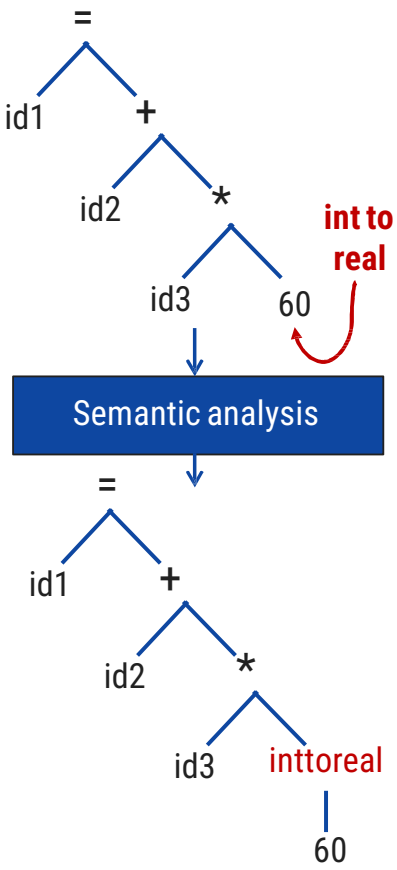


Semantic analysis

- Semantic analyzer determines the **meaning of a source string**.
- It performs following operations:

1. Matching of parenthesis in the expression.
2. Matching of if..else statement.
3. Performing arithmetic operation that are type compatible.
4. Checking scope of operation.

*Note: Consider id1, id2 and id3 are real

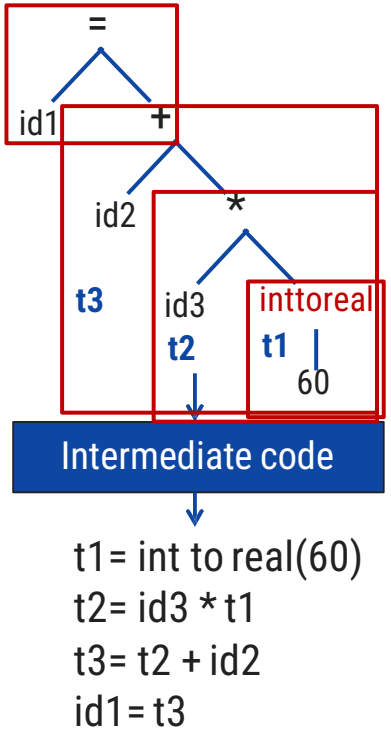


Intermediate code generator

- Two important properties of intermediate code:

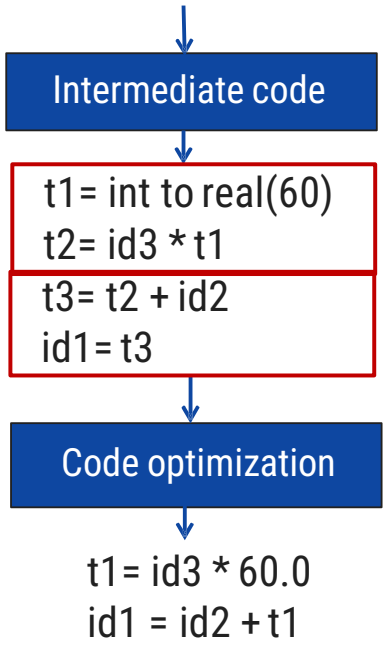
1. It should be **easy to produce**.
2. **Easy to translate** into target program.

- Intermediate form can be represented using **"three address code".**
- Three address code consist of a sequence of instruction, each of which has **at most three** operands.



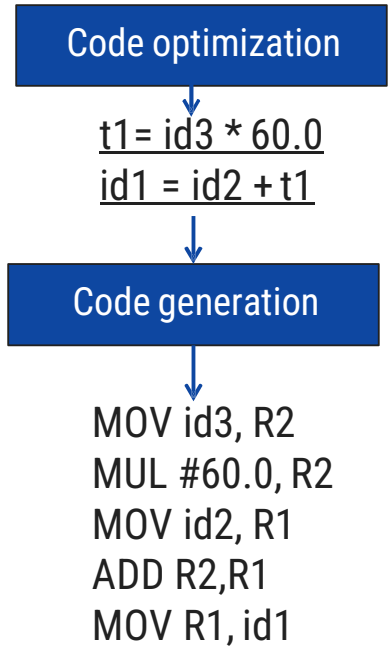
Code optimization

- It **improves** the intermediate code.
- This is necessary to have a **faster execution** of code or **less consumption of memory**.



Code generation

- The intermediate code instructions are **translated into sequence of machine instruction (assembly language)**.



Id3→R2
Id2→R1

Front end & back end (Grouping of phases)

Front end

- Depends primarily on source language and largely independent of the target machine.
- It includes following phases:
    1. Lexical analysis
    2. Syntax analysis
    3. Semantic analysis
    4. Intermediate code generation
    5. Creation of symbol table

Back end

- Depends on target machine and do not depends on source program.
- It includes following phases:
    1. Code optimization
    2. Code generation phase
    3. Error handling and symbol table operation

Characteristics of Good Compiler

- Correctness
- Efficiency
- Portability
- Error Diagnostics
- Optimization Capabilities
- Modularity
- Scalability
- Support for Language Features

GATE CS 2011

1)In a compiler, keywords of a language are recognized during

(A) parsing of the program

(B) the code generation

(C) the lexical analysis of the program

(D) dataflow analysis

GATE CS 2011

1)In a compiler, keywords of a language are recognized during

(A) parsing of the program

(B) the code generation

(C) the lexical analysis of the program

(D) dataflow analysis

OPTION C

GATE CS 2008

- **Some code optimizations are carried out on the intermediate code because**
- (A) they enhance the portability of the compiler to other target processors
- (B) program analysis is more accurate on intermediate code than on machine code
- (C) the information from dataflow analysis cannot otherwise be used for optimization
- (D) the information from the front end cannot otherwise be used for optimization

GATE CS 2008

- **Some code optimizations are carried out on the intermediate code because**
- (A) they enhance the portability of the compiler to other target processors
- (B) program analysis is more accurate on intermediate code than on machine code
- (C) the information from dataflow analysis cannot otherwise be used for optimization
- (D) the information from the front end cannot otherwise be used for optimization
- **Answer: A**

GATE CS 1997

- **A language L allows declaration of arrays whose sizes are not known during compilation. It is required to make efficient use of memory. Which of the following is true?**
- (A) A compiler using static memory allocation can be written for
- (B) A compiler cannot be written for L, an interpreter must be used
- (C) A compiler using dynamic memory allocation can be written for L
- (D) None of the above

GATE CS 1997

- **A language L allows declaration of arrays whose sizes are not known during compilation. It is required to make efficient use of memory. Which of the following is true?**
- (A) A compiler using static memory allocation can be written for
- (B) A compiler cannot be written for L, an interpreter must be used
- (C) A compiler using dynamic memory allocation can be written for L
- (D) None of the above
- **Answer: C**

GATE-CS-2014-(Set-3)

- **One of the purposes of using intermediate code in compilers is to**
- (A) make parsing and semantic analysis simpler.
- (B) improve error recovery and error reporting.
- (C) increase the chances of reusing the machine-independent code optimizer in other compilers.
- (D) improve the register allocation.

GATE-CS-2014-(Set-3)

- **One of the purposes of using intermediate code in compilers is to**
- (A) make parsing and semantic analysis simpler.
- (B) improve error recovery and error reporting.
- (C) increase the chances of reusing the machine-independent code optimizer in other compilers.
- (D) improve the register allocation.
- **Answer: C**

- **In a two-pass assembler, symbol table is**
- (A) Generated in first pass
- (B) Generated in second pass
- (C) Not generated at all
- (D) Generated and used only in second pass

- **In a two-pass assembler, symbol table is**
- (A) Generated in first pass
- (B) Generated in second pass
- (C) Not generated at all
- (D) Generated and used only in second pass
- **Answer: A**

- **How many tokens will be generated by the scanner for the following statement ?**
- **x = x ∗ (a + b) – 5;**
- (A) 12
- (B) 11
- (C) 10
- (D) 07

- **How many tokens will be generated by the scanner for the following statement ?**
- **x = x ∗ (a + b) – 5;**
- (A) 12
- (B) 11
- (C) 10
- (D) 07
- **Answer: A**

- **Symbol table can be used for:**
- A) Checking type compatibility
- B) Suppressing duplication of error message
- C) Storage allocation
- D) All of these

- **Symbol table can be used for:**
- A) Checking type compatibility
- B) Suppressing duplication of error message
- C) Storage allocation
- D) All of these
- **Answer: D**

- **The access time of the symbol table will be logarithmic if it is implemented by**
- A)Linear list
- B) Search tree
- C) Hash table
- D) Self organization list

- **The access time of the symbol table will be logarithmic if it is implemented by**
- A)Linear list
- B) Search tree
- C) Hash table
- D) Self organization list
- **Answer: B**

GATE CSE 2009

- **Match all items in Group 1 with the correct options from those given in Group 2.**

Group 1	Group 2
P. Regular expression	1. Syntax Analysis
Q. Pushdown automata	2. Code Generation
R. Dataflow analysis	3. Lexical Analysis
S. Register allocation	4. Code Optimization

- (A) P-4, Q-1, R-2, S-3
- (B) P-3, Q-1, R-4, S-2
- (C) P-3, Q-4, R-1, S-2
- (D) P-2, Q-1, R-4, S-3

GATE CSE 2009

- **Match all items in Group 1 with the correct options from those given in Group 2.**

Group 1	Group 2
A. Regular expression	1. Syntax Analysis
B. Pushdown automata	2. Code Generation
C. Dataflow analysis	3. Lexical Analysis
D. Register allocation	4. Code Optimization

- (A) P-4, Q-1, R-2, S-3
- (B) P-3, Q-1, R-4, S-2
- (C) P-3, Q-4, R-1, S-2
- (D) P-2, Q-1, R-4, S-3
- **Answer: B**

GATE CSE 2010

- Which data structure in a compiler is used for managing information about variables and their attributes?
- (A) Abstract Syntax Tree
- (B) Symbol Table
- (C) Semantic Stack
- (D) Parse Table

GATE CSE 2010

- Which data structure in a compiler is used for managing information about variables and their attributes?
- (A) Abstract Syntax Tree
- (B) Symbol Table
- (C) Semantic Stack
- (D) Parse Table
- **Answer: B**

ISRO 2020

- The number of tokens in the following C code segment is
- A. 27
- B. 29
- C. 26
- D. 24

```
1.switch(inputvalue)
2.{
3.case 1 : b =c*d; break;
4.default : b =b++; break;
5.}
```

ISRO 2020

- The number of tokens in the following C code segment is
- A. 27
- B. 29
- C. 26
- D. 24

```
1.switch(inputvalue)
2.{
3.case 1 : b =c*d; break;
4.default : b =b++; break;
5.}
```

- Answer: 26

ISRO 2020

```
1.switch(inputvalue)
2.{
3.case 1 : b =c*d; break;
4.default : b =b++; break;
5.}
```

- **The tokens in the code segment are as follows:**
- switch - Keyword
- (- Punctuation
- inputvalue - Identifier
-) - Punctuation
- { - Punctuation
- case - Keyword
- 1 - Integer Literal
- : - Punctuation
- b - Identifier
- = - Operator
- …..

GATE CSE 2020

- **Consider the following statements.**
- I. Symbol table is accessed only during lexical analysis and syntax analysis.
- II. Compilers for programming languages that support recursion necessarily need heap storage for memory allocation in the run-time environment.
- III. Errors violating the condition ‘any variable must be declared before its use’ are detected during syntax analysis.
- Which of the above statements is/are TRUE?

A. I only
B. I and III only
C. II only
D. None of I, II and III

GATE CSE 2020

- **Consider the following statements.**
- I. Symbol table is accessed only during lexical analysis and syntax analysis.
- II. Compilers for programming languages that support recursion necessarily need heap storage for memory allocation in the run-time environment.
- III. Errors violating the condition ‘any variable must be declared before its use’ are detected during syntax analysis.
- Which of the above statements is/are TRUE?

A. I only
B. I and III only
C. II only
D. None of I, II and III

- **Answer: D**

GATE CSE 2020-Explanation

- Symbol table is a data structure used by compilers to store information about the variables, functions, and other entities in a program. It is accessed during both lexical analysis and syntax analysis, but may also be used by later stages of the compilation process, such as code generation.
- Compilers for programming languages that support recursion may or may not need heap storage for memory allocation in the run-time environment. The decision to use heap storage or not depends on the specific implementation of the compiler and the features of the programming language.
- Errors violating the condition 'any variable must be declared before its use' are typically detected during the semantic analysis phase of compilation, not during syntax analysis. **Syntax analysis checks the structure of the program** to ensure it follows the rules of the programming language's grammar, while semantic analysis checks the meaning of the program to ensure it is semantically correct.

Incremental-Compiler is a compiler?

- (A) which is written in a language that is different from the source language
- (B) compiles the whole source code to generate object code afresh
- (C)compiles only those portion of source code that have been modified.
- (D)that runs on one machine but produces object code for another machine

Incremental-Compiler is a compiler?

- (A) which is written in a language that is different from the source language
- (B) compiles the whole source code to generate object code afresh
- (C)compiles only those portion of source code that have been modified.
- (D)that runs on one machine but produces object code for another machine
- **Answer: C**

Which phase of compiler generates stream of atoms?

- (A) Syntax Analysis
- (B) Lexical Analysis
- (C) Code Generation
- (D)Code Optimization

Which phase of compiler generates stream of atoms?

- (A) Syntax Analysis
- (B) Lexical Analysis
- (C) Code Generation
- (D)Code Optimization
- **Answer: B**

Which one of the following is NOT performed during compilation?

- A)Dynamic memory allocation
- B)Type checking
- C)Symbol table management
- D)Inline expansion

Which one of the following is NOT performed during compilation?

- A)Dynamic memory allocation
- B)Type checking
- C)Symbol table management
- D)Inline expansion
- **Answer: A**

Distributed File Systems (DFS)

(Big Data Analytics)

Distributed File System (DFS)

Local File System	Distributed File System
Manages files on a single machine	Manages files on a multiple machines
LFS uses Tree format to store Data.	Provides Master-Slave architecture or Cluster based architecture for Data storage.
Generally store the data as single block in single machine	Store the data in multiple blocks in multiple machines
It is not reliable because LFS data does not replicate the Data files.	It is reliable because in DFS data blocks are replicated into different DataNodes.
Files can be accessed directly in LFS.	Files can not be accessed directly in DFS because the actual location of data blocks are only known by NameNode.
LFS is cheaper because it does not needs extra memory for storing any data file.	DFS is expensive because it needs extra memory to replicate the same data blocks.
LFS is not appropriate for analysis of very big file of data because it needs large time to process.	DFS is appropriate for analysis of big file of data because it needs less amount of time to process as compare to Local file system.

Distributed File System (DFS)

- DFS differs from typical file systems (i.e., FAT32,FAT64, NTFS) in that it allows direct host access to the same file data from multiple locations. Indeed, the data behind a DFS can reside in a different location from all of the hosts that access it.
- A Distributed File System (DFS) as the name suggests, is a file system that is distributed on multiple file servers or multiple locations.
- Since more than one client may access the same data simultaneously, the server must have a mechanism in place (such as maintaining information about the times of access) to organize updates so that the client always receives the most current version of data and that data conflicts do not arise.
- Distributed file systems typically use file or database replication (distributing copies of data on multiple servers) to protect against data access failures.
- Examples of distributed file systems:
  - Sun Microsystems' Network File System (NFS),
  - Novell NetWare,
  - Microsoft's Distributed File System,
  - Google File System

Design Goals

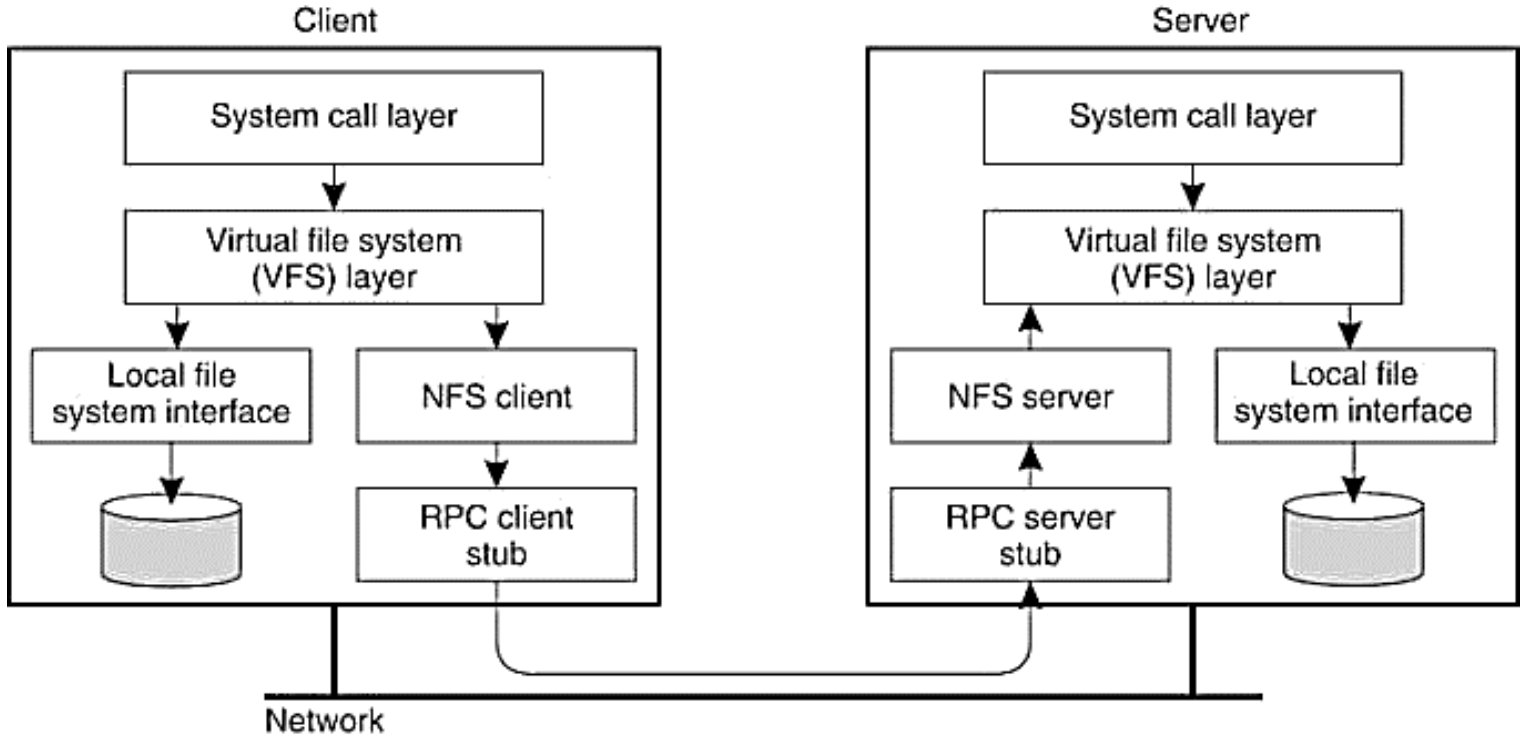
A DFS should provide:

- **Network transparency:**
  - Hide the details of *where* a file is located.
- **High availability:**
  - Ease of accessibility irrespective of the physical location of the file.
- **Scalability:**
  - Number of users, Servers, and files handled etc.
- **Concurrency:**
  - Handle concurrent access by different clients in the network

DFS Architecture

Client-Server based Architecture (Example: Network File System (NFS))

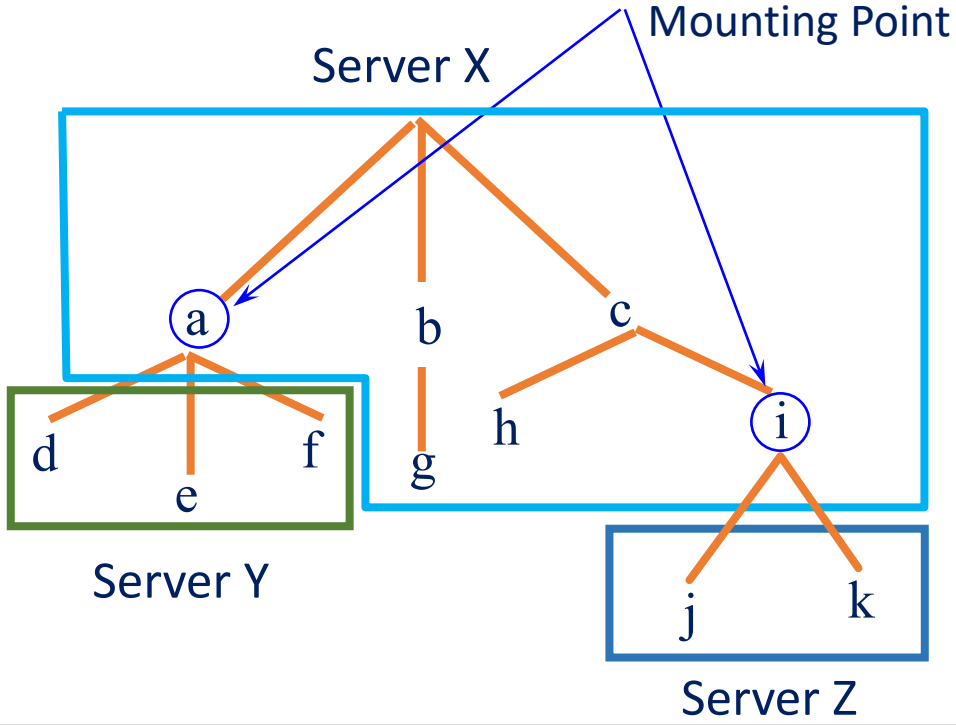
- Client-Server is one of the common architecture in DFS.
- Sun Microsystem's Network File System (NFS) being one of the most widely-deployed.
- NFS has been implemented for a large number of different operating systems.



The basic NFS architecture for LINUX/UNIX systems

Distributed File System (DFS)

- **DFS:** It is a file system that is distributed on multiple file servers or multiple locations.
- **Mechanisms for building DFS:** (Mounting, Caching, Hints, Bulk transfer, and Security)



Name: File or directories
Name resolution: process of mapping the Name to Physical Storage
Namespace: Collection of names (files/Directories)

Mounting: Mounting makes file systems, files, directories, and devices available for use at a particular location. It is the only way a file system is made accessible. It allows the binding together of different file namespaces to form a single file system structure (Single hierarchical namespace).

Caching: When files are in different servers, caching might be needed to improve the response time. A copy of data (in files) is brought to the client (when referenced). Subsequent data accesses are made on the client cache. (Client cache and server cache)

Hints: Data must be consistent in the cached memory. An alternative approach to maintaining consistency is to treat cached data as hints. A time-stamp based method is used for validate cache block before they are used.

Bulk data transfer: Helps in reducing the delay due to transfer of files over the network. Obtain multiple number of blocks with a single seek

Encryption: Establish a key for encryption with the help of an authentication server.

Mechanisms for DFS

Caching:

- Performance of distributed file system, in terms of response time, depends on the ability to “get” the files to the user.
- When files are in different servers, caching might be needed to improve the response time.
- A copy of data (in files) is brought to the client (when referenced). Subsequent data accesses are made on the client cache.
- Client cache can be on disk or main memory.
- Data cached may include future blocks that may be referenced too.
- Caching implies DFS needs to guarantee consistency of data.

Mechanisms for DFS

Hints:

- Caching is an important mechanism to improve the performance of a file system. But for guaranteeing the consistency of the cached items requires elaborate and expensive client/server protocols.
- An alternative approach to maintaining consistency is to treat cached data as hints.
- A time-stamp based method is used for validate cache block before they are used.
- With this scheme, cached data is not expected to be completely consistent, but when it is, it can dramatically improve performance.
- For maximum performance benefit, a hint should nearly always be correct.

Mechanisms for DFS

Bulk data transfer:

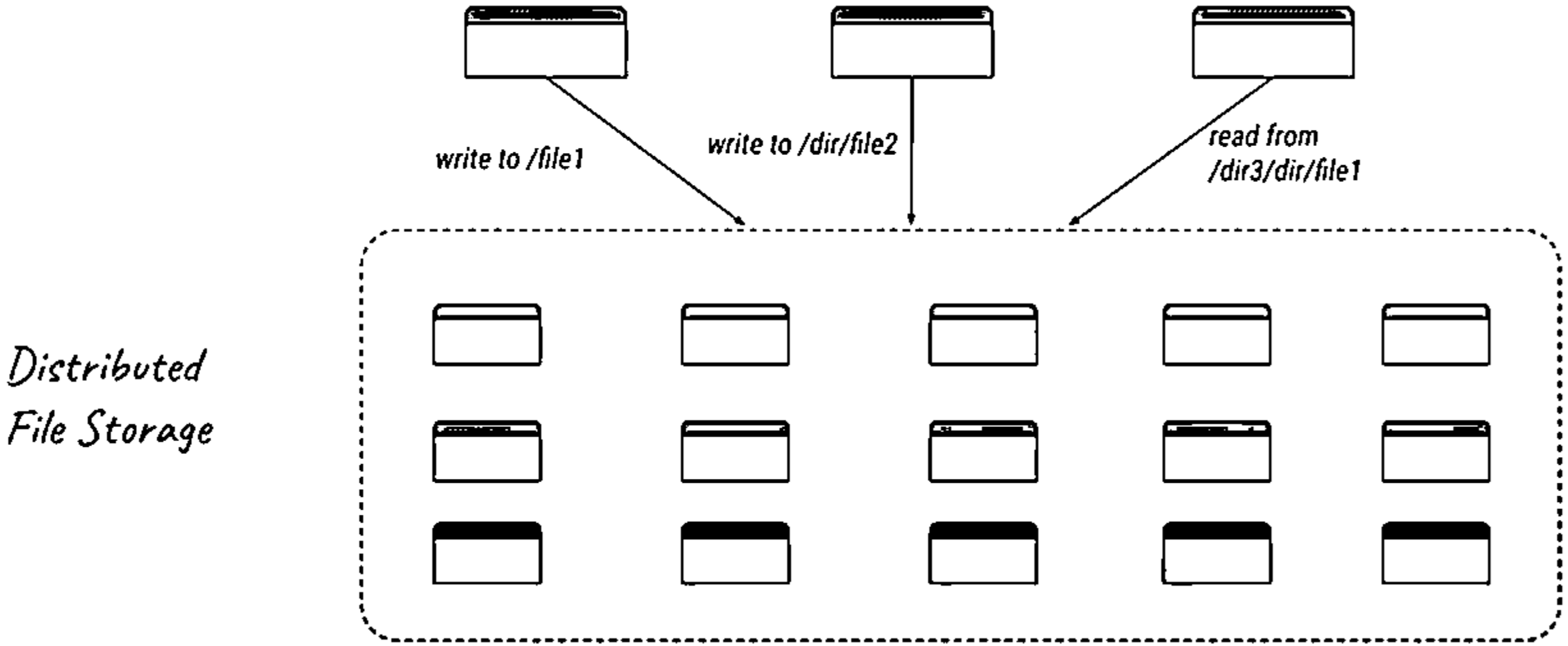
- helps in reducing the delay due to transfer of files over the network.
- *Bulk*:
  - Obtain multiple number of blocks with a single seek
  - Format, transfer large number of packets in a single context switch.
  - Reduce the number of acknowledgements to be sent.

- **Encryption:**
  - Establish a key for encryption with the help of an authentication server.

Google File System (GFS) (Cluster based Architecture)

- Google uses the GFS to organize and manipulate huge files and to allow application developers the research and development resources they require.
- The GFS is unique to Google and is not for sale.
- Google doesn't reveal how many computers it uses to operate the GFS. In official Google papers, they only says that there are "thousands" of computers in the system.
- GFS:
  - Support and deal with large files.
  - Scalable
  - Use the general file commands: Open, Create, Read, Write and Close.
  - Also support specialized command: Append and Snapshot
- File is divided into number of “Chunks” and each one is associated “Chunk handle”
- GFS balance the workload of the chunks.

Google File System (GFS)



1 cluster = 100s of commodity servers

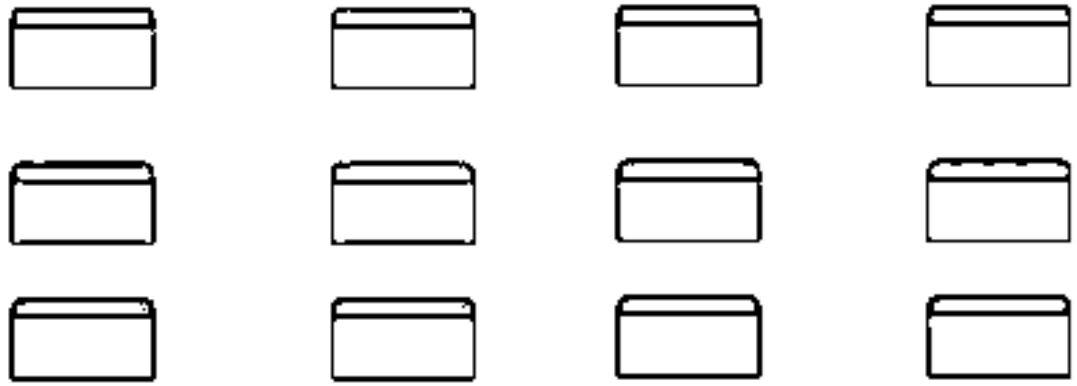
Google File System (GFS) – Design Consideration

Commodity Hardware

Failures are common

- disk / network / server
- OS bugs
- human errors

Commodity servers are cheap & can be made to scale horizontally with right software

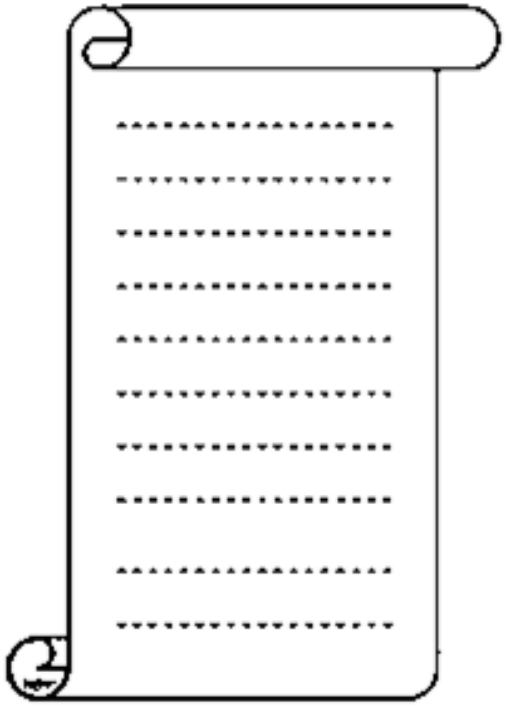


Google File System (GFS) – Design Consideration

Large files

100MB to multi-GB files

- Crawled web documents
- Batch processing

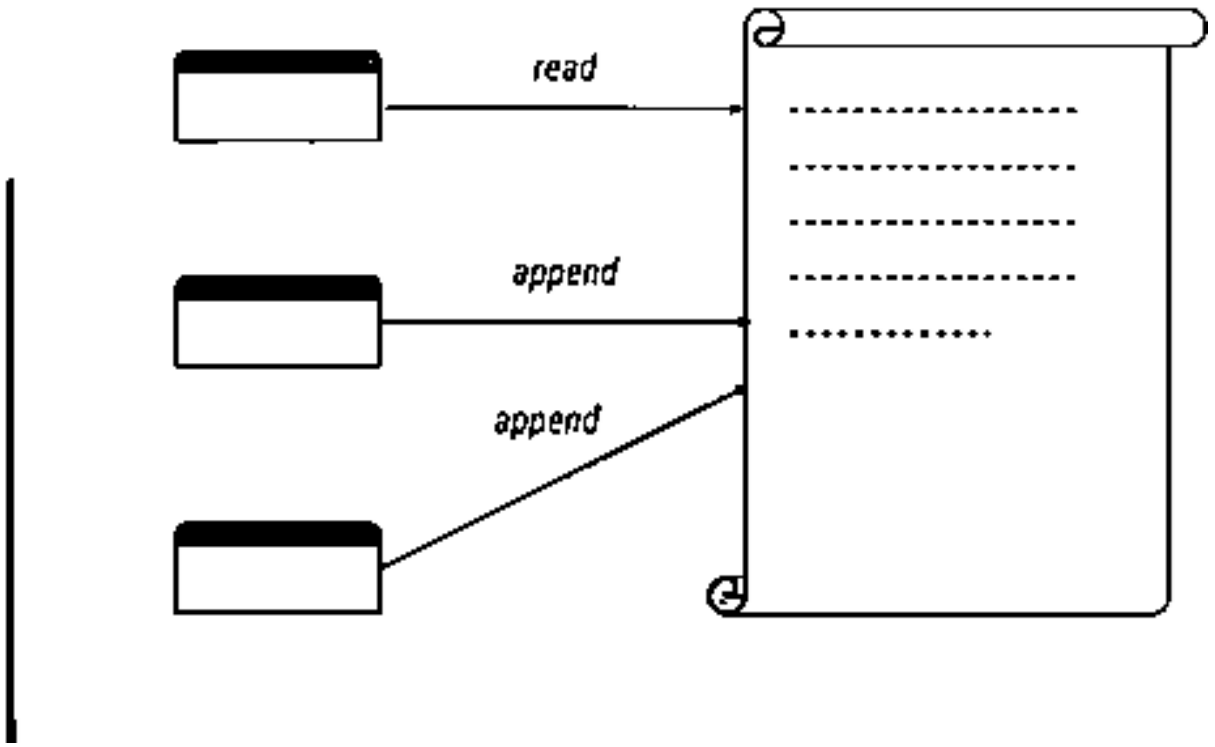


Google File System (GFS) – Design Consideration

File Operations

Read + Append only

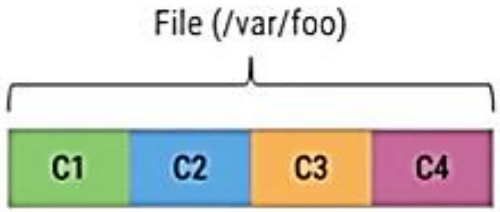
- No random writes
- Mostly sequential reads



Think of web crawling – keep appending crawled Content & use batch processing (Reads) to create index

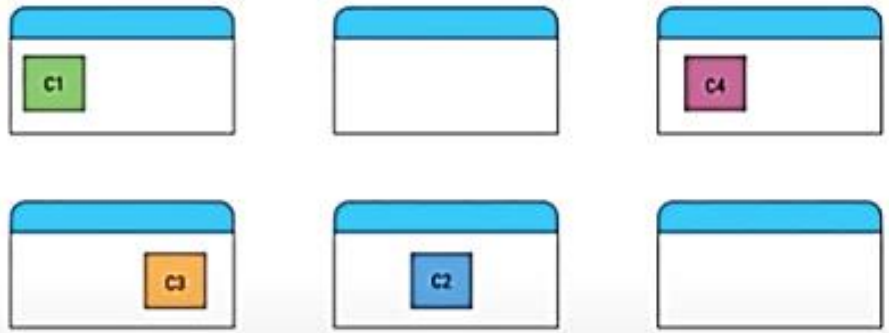
Google File System (GFS) – Design Consideration

Chunks



Files split into chunks

- Each chunk of 64MB
- Identified by 64 bit ID
- Stored in Chunkservers



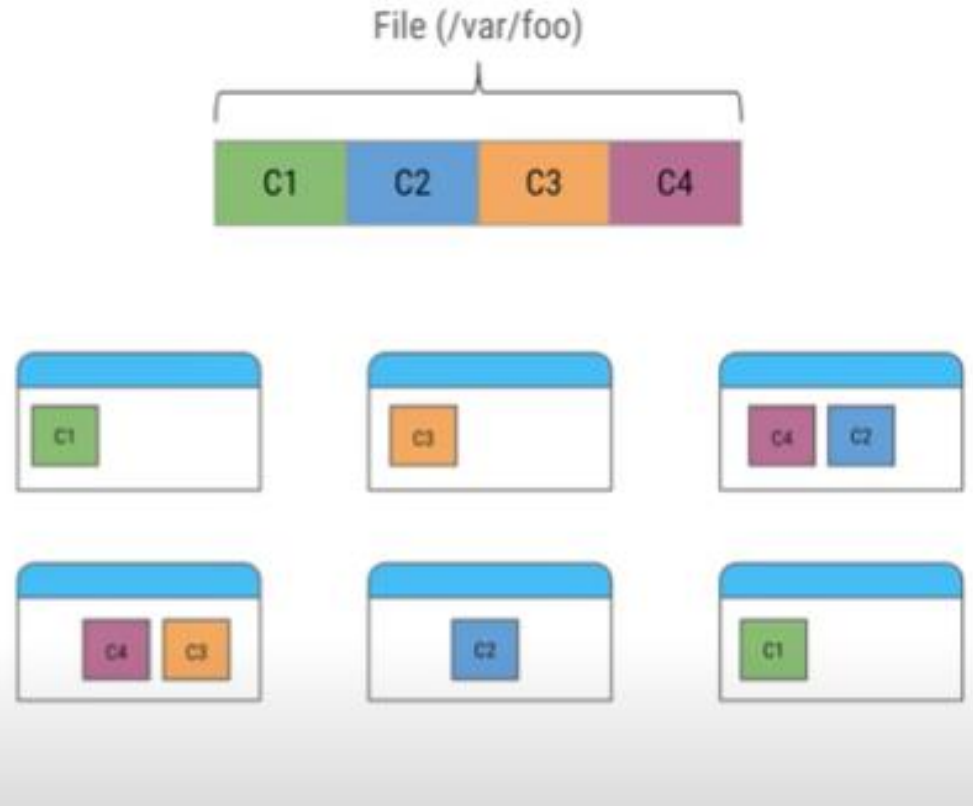
Chunkservers – Chunks of single file are distributed on multiple machines

Google File System (GFS) – Design Consideration

Replicas

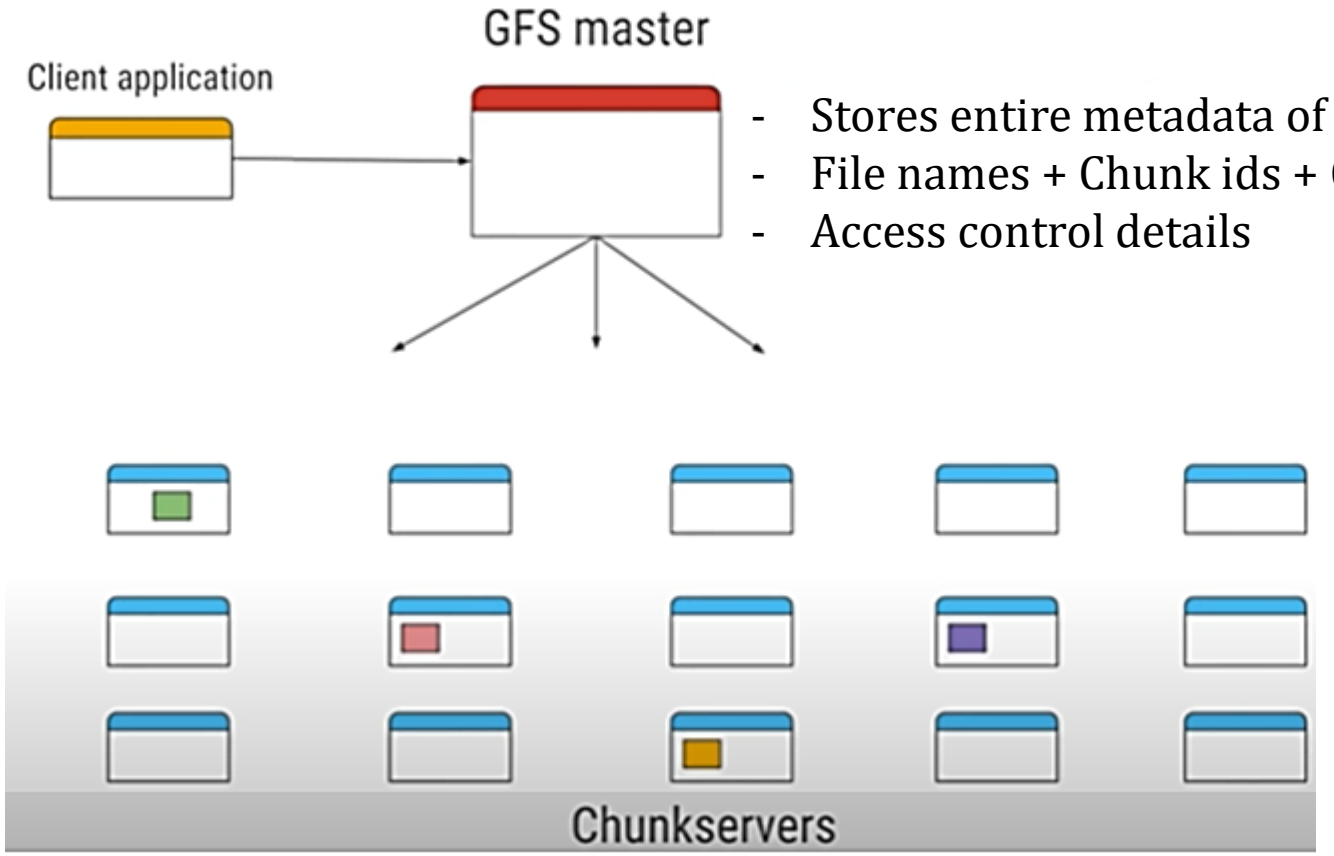
Files split into chunks

- Replica count by client
- commodity server failures



Replicas ensure durability of data if chunkserver goes down

Google File System (GFS) – Design Consideration



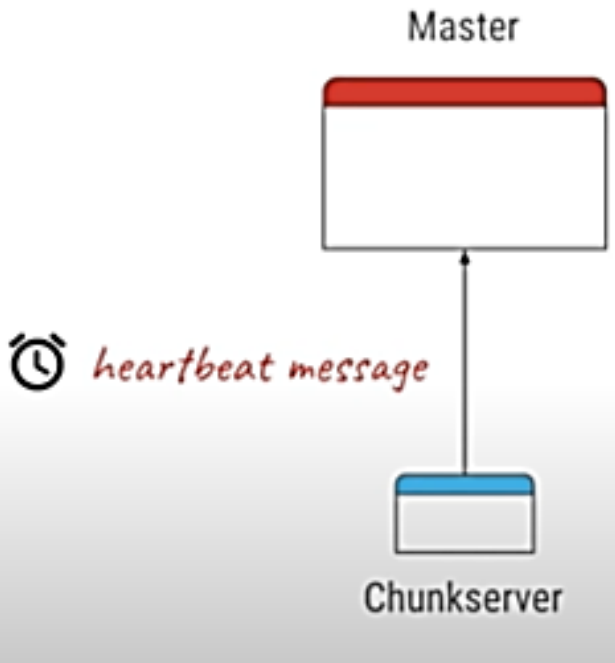
- Stores entire metadata of the cluster
- File names + Chunk ids + Chunk Locations
- Access control details

File	Chunks	Locations
/var/foo	ffe0	server1 (replica 1) server2 (replica 2)
	ff21	server 4 (replica 1)

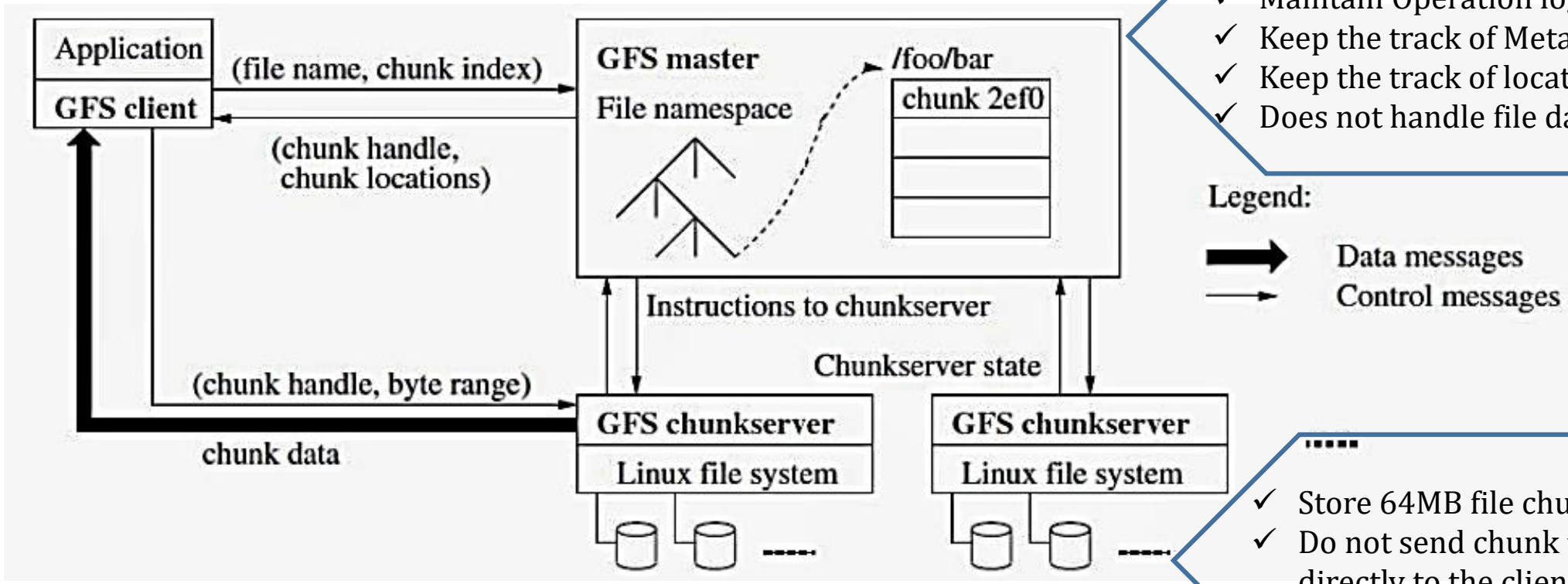
Google File System (GFS) – Design Consideration

Heartbeats

Regular heartbeats to ensure chunkservers are alive



GFS Architecture



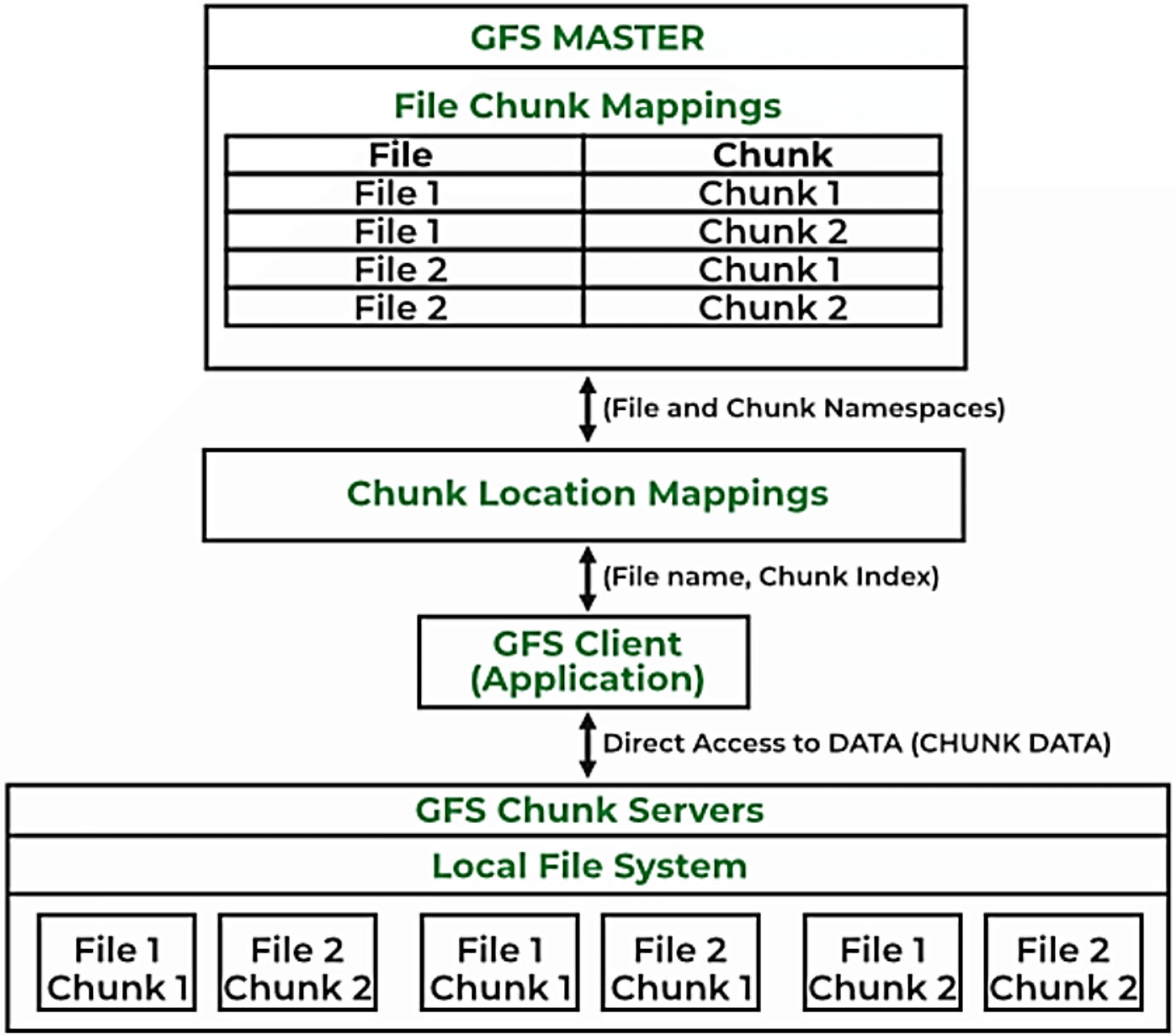
- ✓ Act as Coordinator
- ✓ Maintain Operation log
- ✓ Keep the track of Metadata
- ✓ Keep the track of location of chunks
- ✓ Does not handle file data

- ✓ Store 64MB file chunks
- ✓ Do not send chunk to master-server but directly to the client
- ✓ Chunk Replica (Primary and Secondary)

Reads

In GFS, There are three main entities: *Client*, *Master Server* and *Chunk Server*

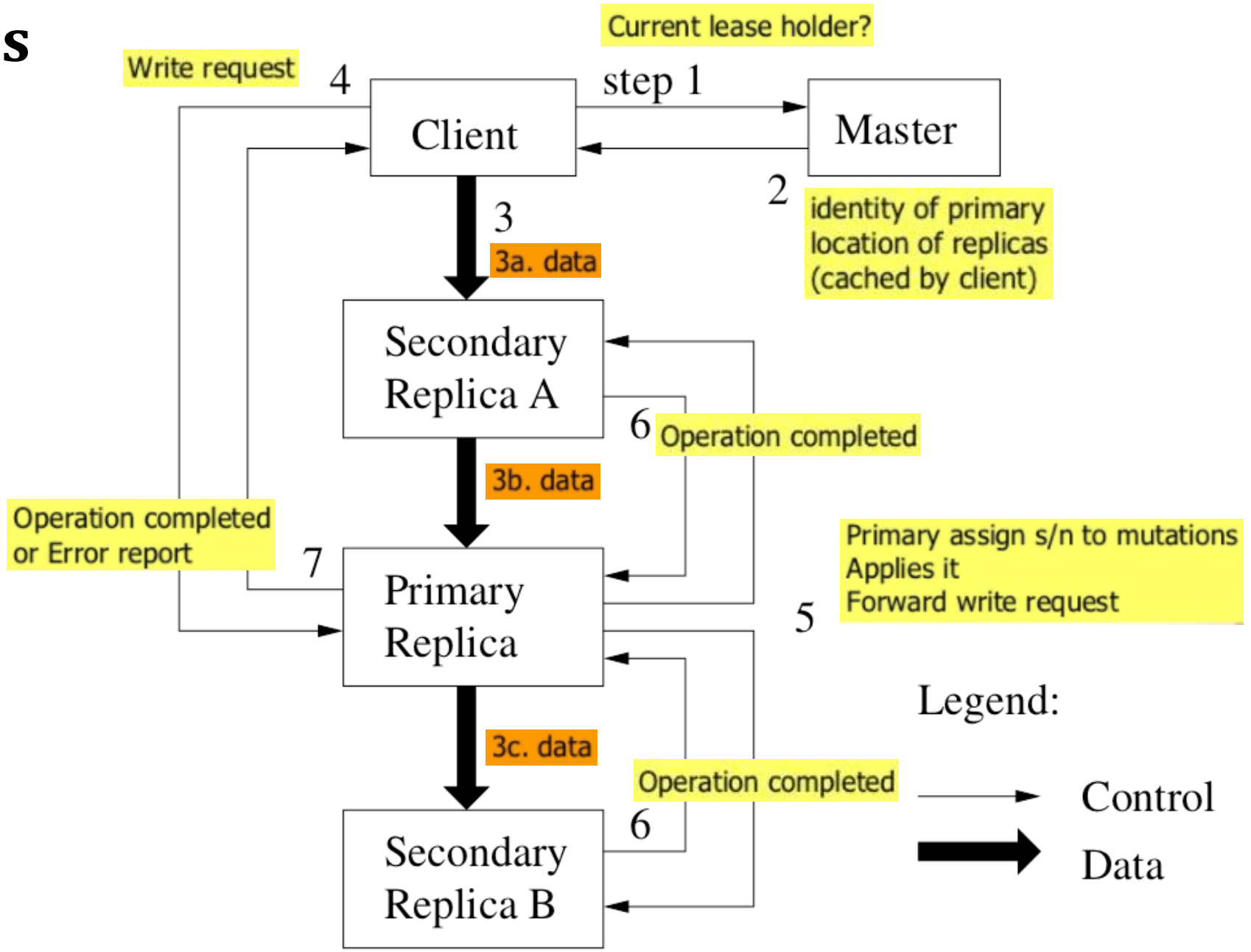
GFS Architecture



System Interactions

Writes

1. Ask for locations to write
2. Get replicate locations
3. Write data to closest replica.
4. Request commit to primary
5. Primary instructs order of writes to secondaries
6. Secondaries acknowledge
7. Primary ack to client



System Interactions

1. Client contact to master for all Chunk-servers.
2. Master grants new lease on chunk, increase chunk version no's including replica’s.
3. Client pushes data to all servers.
4. Once data is asked, client sends write request to primary. Primary decides serialization order for all incoming modifications and applies them to chunk.
5. After finishing modification, primary forwards write request and serialize order to secondaries.
6. All secondaries reply back to the primary once they finish the modifications.
7. Primary replay back to client, either with success or error
   - If write success at primary but fails at any of secondaries then we have in inconsistent state. So error returned to client.
   - If error, client can retry step-3 to step-7

Major limitations of the existing Google File System

- **Single master bottleneck:** GFS relied on a single master node for metadata management, creating a scalability bottleneck as data volume and access requests grew. Imagine a single librarian managing a massive library.
- **Limited metadata scalability:** The centralized metadata storage on the master node couldn't scale efficiently.
- **High latency for real-time applications:** GFS was optimized for batch processing, leading to higher latency for real-time applications like Search and Gmail. Think of searching for a specific book in a large library with one librarian — it can take time.
- **Static data distribution:** GFS had a predefined data chunk size (64 MB) and replication strategy, lacking flexibility for diverse workloads and storage options like flash memory. Imagine using only one type of box for all your belongings — some things might not fit well.

Colossus File System(extended version of GFS)

- Solution of “Major limitations of the existing GFS” ---> Colossus File System

- The need for Colossus arose from Google’s rapid growth and data demands.

- GFS were becoming inadequate in handling the ever-increasing volume of data generated by Google’s core services.

- Colossus offered a solution with its scalability, reliability, and efficiency. Its unique features allowed Google to scale data storage and access seamlessly, ensuring smooth operation and fast performance for its services.