Programming Club, IIT Kanpur

Summer Project: NFS Goes Online

Documentation

Abhay Tripathi — Advaith GS — Arsh Aryan — Aryan Mahesh Gupta — Devansh Bhardwaj — Harshit Tomar — Himanshu Sharma — Jayanth Matam — Manas Jain Kuniya — Rishi — Nikhil

Contents

1 Project Overview

The Summer Project NFS Goes Online offered by Programming Club, IIT Kanpur aims to introduce the basics of systems programming to the mentees. The project has been divided into two segments: **Building a single-threaded web Server using Rust** and **Building a Network File System using C**.

The first half of the project was focused on the **networking** part, wherein the mentees first learned the RUST programming language, secondly learned about the 7 Layers of the OSI Model and Networking Protocols, and lastly utilized the learnings in order to implement A Single-Threaded Web Server using Rust. Rust was primarily used due to its memory safety, ability to listen with ports efficiently (being a low-level language) and it's immense speed as compared to other High-Level Languages. For learning RUST, the primary resource was The Rust Programming Language by Steve Klabnik and Carol Nichols; while secondary resources were Rustlings Github repository, documentation, tutorials, etc.

The second half of the project health with the **Operating Systems** part, wherein the mentees first learned about **Virtualization** and **Concurrency** in order to understand how Operating Systems work and what problems they actually solve. Post that, they delved deep into **Persistence** and implemented a Distributed File System, more specifically the Sun Network File System. The primary learning resource for this part was the Operating Systems: Three East Peaces (OSTEP) book by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison), and some GitHub repositories were used to understand the implementation part of the project.

In this project report, we shall be briefly cover the theory of the concepts used in the projects, post which we shall elaborate on the implementation.

2 RUST Language

2.1 Features and Advantages

1. Reliable Memory Safety: Rust's ownership concept guarantees memory safety. Memory safety via handling of variables, pointers, and the borrow checker system is enforced at compile-time, reducing potential errors.

2. Zero Cost Abstraction: Rust developers can use abstractions in the code without sacrificing performance.

3. Control Over Low-Level Details: Rust allows for fine-grained control over memory management, allowing you to control the handling of memory and ensuring memory safety.
4. Error Handling: Using the Result and Option type, combined with its type safety measures, Rust reduces the possibility of runtime errors, instead catching them during compilation itself. which is one of its greatest achievements.
5. Smart Move Semantics: Rust's move semantics efficiently transfer data between variables, thus avoiding costly copies.
6. Fearless Concurrency: Built-in support for concurrent and parallel programming.
7. Versatile Pattern Matching: greater control over program flow
8. Strong Static Typing: allows for type inferring, reducing the need for explicit type annotation in many cases, leading to type safety.

2.2 Ownership

There are two ways by which every language manages memory. 1. Using a garbage collector (like Java), 2. Using manual memory management(like in C, C++). Both have their advantages and disadvantages.

Garbage collector	Manual memory allocation	Rust's ownership model
no fine grain control	leads to memory bugs (if not used correctly)	extremely safe
makes the program heavy	learning curve is high	light and fast
memory safe	fine grain control	easier than MMA

Figure 1: Ownership

- Ownership is a method Rust uses to get around memory problems. Ownership rules:
  - Each value in Rust has an owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value will be dropped.
- When a value is passed into a function, ownership is transferred to the function. And when the function is executed, the value goes out of scope.
- Because this is problematic, Rust lets us reference the values when passing into functions. This is called borrowing. The value stored inside the function variable is actually a pointer to the real value.

2.3 Structs and Enums

Two of the most important constructs in Rust's type system are structures (struct) and enumerations (enum), which allow developers to create complex data types that accurately model the problem domain.

- Structs are a way to define custom data types in Rust. They allow you to group related data together into a single, cohesive unit. Structs are similar to classes in other languages but with some important differences.
- Structs can also have methods, which are functions associated with the struct.
- Enums, short for enumerations, are another way to define custom data types in Rust. They allow you to define a type with a fixed set of values, known as variants. Enums are useful when you need to represent a value that can be one of several different options.
- By using enums to group related structs or adding a field to a struct that is an enum type, you can create complex data types that accurately model your problem domain.

2.4 Packages, Crates and Modules

Packages, Crates, and Modules help in managing large projects. As we write large programs, organizing your code will become increasingly important. By grouping related functionality and separating code with distinct features, we can clarify where to find code that implements a particular feature and where to go to change how a feature works.
Packages let us build, test, and share crates, **crates** are a tree of modules that produces a library or executable, **Modules** let us control the organization, scope, and privacy of paths and **paths** are a way of naming an item, such as a struct, function, or module

- A **crate** is synonymous with a 'library' or 'package' in other languages. Hence "Cargo" as the name of Rust's package management tool: you ship your crates to others with Cargo. Crates can produce an executable or a library, depending on the project.
- A **package** is a bundle of one or more crates that provides a set of functionality. A package contains a Cargo.toml file that describes how to build those crates. Cargo is actually a package that contains the binary crate for the command-line tool you've been using to build your code.
- Each crate has an implicit root **module** that contains the code for that crate. You can then define a tree of sub-modules under that root module. Modules allow you to partition your code within the crate itself.

2.5 Collections

Rust's standard collection library provides efficient implementations of the most common general purpose programming data structures. By using the standard implementations, it should be possible for two libraries to communicate without significant data conversion. Some common examples of collections are: Vectors and Hashmaps.

3 Asynchronous Programming

3.1 Threads

An **OS Thread** (spawned using std::thread::spawn in rust) is a concurrency model through which parts of the program can run parallel. Completion of the main thread drops all other threads. Threads communicate with each other using messages (std::sync::mpsc/tokio::sync::mpsc::channel()) where there is a sender and a receiver. This way threads can be dropped together upon receiving the signal.

3.2 Async/Await

async: future is the type of value returned by an async function. Using the block_on(future) function on a future blocks the thread till a future is completed. The **.await** feature, unlike block_on(future), lets other futures in the thread proceed in case the current future is not able to progress (e.g. receiving a signal using mpsc channel. Till the signal is received, we can use await to let rest of the thread function.)

3.3 Threads and Async/Await uses in a Webserver

A Webserver needs to have many concurrent parts running. OS Threads are vital for a multi-threaded Webserver. Even for a single-threaded server, creating a client thread and a server thread and establishing communication between them using messages is necessary so that both parts function parallel and interact. (e.g. we can create a client thread which makes a finite number of requests as pre-test for successful connections before an external client makes a request.) Async/Await functionality is useful in shutting down a server as well as enabling different types of client requests (e.g. .awaiting on all possible types of requests so that till a request is received, all other parts can function.) For shutting down, we can use .await on receiving the shutdown signal (ctrl_c()). There is a helpful macro tokio::select! which has branches defined for completion of different futures. When a future gets completed, it executes its branch and cancels all other branches

4 Networking

4.1 Types of Network Architectures

4.1.1 Client-Server Architecture

- Servers: Providers of Resources, Provides information and access to resources to the client - Clients: Requesters of Resources, Capable of receiving information and using a resource from the server

Browser-Server Interaction

1. User enters the URL of the website or file.
2. The Browser then requests the DNS Server.
3. DNS Server lookup for the address of the WEB Server.
4. The DNS Server responds with the IP address of the WEB Server.
5. The Browser sends over an HTTP/HTTPS request to the WEB Server's IP (provided by the DNS server)
6. The Server sends over the necessary files for the website.
7. The Browser then renders the files and the website is displayed. This rendering is done with the help of DOM (Document Object Model) interpreter, CSS interpreter, and JS Engine collectively known as the JIT or (Just in Time) Compilers.

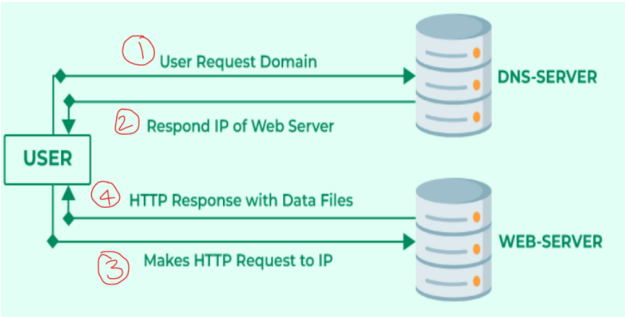


Figure 2: Client-Server Architecture

4.1.2 Peer to Peer (P2P) Architecture

- Each node acts as a server and thus there is no central server in the network
- Tasks are equally divided amongst the nodes.
- Each node connected in the network shares an equal workload.

- For the network to stop working, all the nodes need to individually stop working. This is because each node works independently.

4.2 The OSI Model

The Open Systems Interconnection Model is a 7 layer architecture, wherein each layer is essentially a package of protocols. These Layers work collaboratively to transmit the data from one person to another.

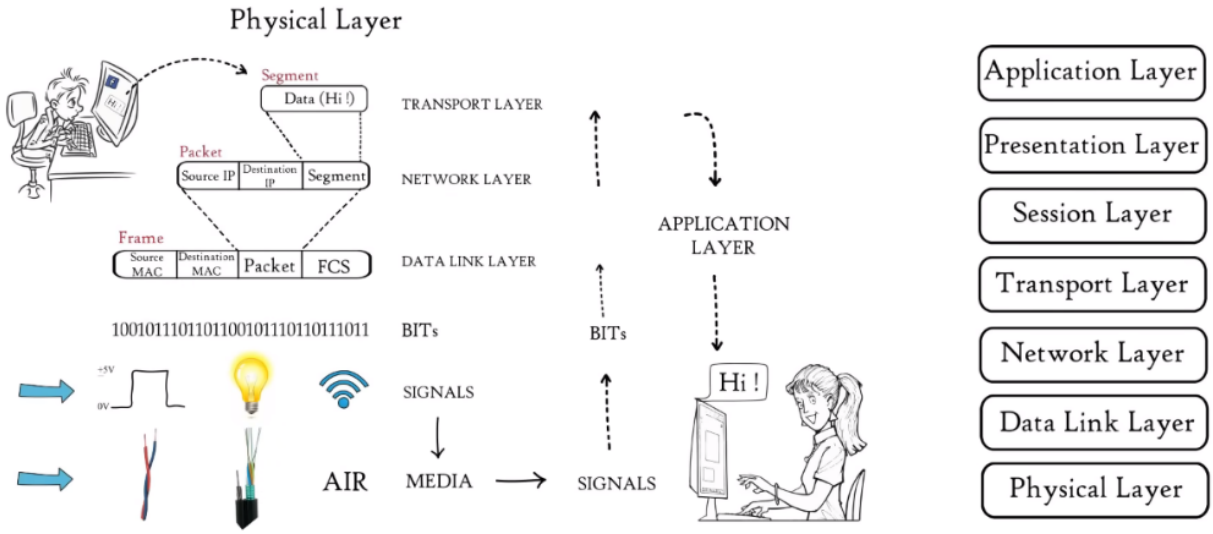


Figure 3: OSI Model

4.2.1 Application Layer

- Used by network applications
- App Layer Protocols: HTTP, HTTPS, DNS, FTP, SMTP, NFS, FMTP, TELNET (for virtual terminals), POP3,

4.2.2 Presentation Layer

- Receives data from application layer
- Converts numbers and characters into binary
- It's functions are:
  1. Translation (ASCII to EBCDIC: extended binary-coded decimal interchange code)
  2. Compression
  3. Encryption/Decryption (SSL protocol: Secure Sockets Layer)

4.2.3 Session Layer

- Has helpers called APIs (Application Programming Interfaces)
- Network Basic Input Output System (NETBIOS) is an example of API
- allowing applications on separate computers to communicate over a Local Area Network
- Files from server are received by the client via DATA PACKETS
- Session Layer keeps a track of which data packet belongs to which file and where it is going
- Functions:
    1. Session Management
    2. Authentication
    3. Authorization

4.2.4 Transport Layer

4.2.4.1 Utilities

Segmentation

- Data from the session layer is divided into small data units (segments)
- Each segment contains:
    1. Sequence Number: Helps to reassemble segments in the correct order to form correct messages on the client side
    2. Port Number: Directs each segment to the correct application

Flow Control The amount of data being transmitted per unit time can be controlled via Flow Control.
Error Control If data at the receiver side is corrupted or missing, the Transport layer uses AUTOMATIC REPEAR REQUEST schemes to retransmit the loss of corrupted data

4.2.4.2 Protocols of Transport Layer

1. **Transmission Control Protocol (TCP):**
    - Connection oriented Transmission
    - Provides Feedback, that is lost data can be retransmitted
    - Slower
    - Used where full data delivery is must

- eg: www, tcp, mails

2. **User Datagram Protocol (UDP):**
   - Connectionless Transmission
   - No Feedback provided
   - used in: online streaming, movies, games, songs, VoIP, TFTP, DNS

4.2.5 Network Layer

- Used for data transmission between two devices on different networks
- Routers reside on this layer
- Functions:
  1. Logical Addressing: The IP addresses of receiver and sender are assigned to the data segment to form IP packet.
  2. Routing: Moving Data packet from source to destination, Done via assignment of a mask.
  3. Path Determination: OSPF (Open Shortest Path First), BGP (Border Gateway Protocol), IS-IS (Intermediate System to Intermediate System) are used to determine best path for data delivery.

4.2.6 Data Link Layer

- Receives Data Packet from Network layer
- Addresses : Logical (done at network layer) and Physical (done at DLL)
- In Physical Addressing, MAC addresses of receiver and sender as assigned to the data packet to form a Frame
- MAC (Media Access Control) Number: 12 digit alphanumeric code embedded in Network Interface Card.
- Function:
  1. It allows upper layers of OSI model to access media (ie wires, air etc) via FRAMING
  2. Controls how data is placed and received from media (ie wires, air etc)
     - Media Access Control: Method of getting the frame on and off the media
     - Carrier Sense Multiple Access: Multiple devices can be connected to the common media. When these devices send media at the same time, then to prevent collision, DLL keeps a check on whether the media is free or not

4.2.7 Physical Layer

The Physical Layer converts binary information into signals and transmits over media. It can be: an electrical signal (Cu Wires), light signal (Optical Fiber), or radio signal (Air)

4.3 Hypertext Transfer Protocol

- Stateless Application-level protocol for distributed, collaborative, hypertext information systems
- A stateless protocol is **a type of communication that doesn't depend on previous communications between computers**. In other words, stateless protocols don't keep track of any information about the packets being sent.
- A Stateless Protocol is a type of network protocol in which clients send a server request after which the server responds based on the current state.
- HTTP is built on top of TCP and APIs are built on top of HTTP

Message Format

```
HTTP-message = start-line CRLF
\*(field-line CRLF)
CRLF
\ [ message-body ]
```

here, CRLF = Character Return and Line Feed
A message can be either a request from client to server or a response from server to client. Syntactically, the two types of messages differ only in the start-line, which is either a request-line (for requests) or a status-line (for responses)

```
start line = request-line / status-line
```

```
request-line = method SP request-target SP HTTP-version
```

here:

- SP is space
- method = GET/POST (case sensitive, else 400 error: Bad Request)

```
status-line = HTTP-version SP status-code SP [ reason-phrase ]
```

status-code is a 3-digit int code describing the result of the server's attempt to understand and satisfy the client's corresponding request

4.4 Transmission Control Protocol

- Divided into two layers: the Transport layer and the Internet layer
    1. Transport Layer
        - TCP (Transmission Control Protocol) and UDP (User Datagram Protocol)
        - for ensuring that data is transmitted reliably from one device to another
    2. Internet Layer:
        - responsible for transmitting data packets between devices
        - Internet Protocol (IP) and the Address Resolution Protocol (ARP).
        - IP is responsible for routing data packets between devices
        - ARP is used to map IP addresses to physical addresses
- Also Contains various Application Layer Protocols:
    1. HTTP: Web Browsing
    2. FTP: File Transfer
    3. SMTP: Mails
- It chooses how the information will be traded over the web through end-to-end communications that incorporate how the information ought to be organized into bundles (bundles of data), addressed, sent, and received at the goal.

Three Way Handshake

1. STEP 1 (SYN)
    - Client will send a packet to the server which contains the initial sequence number (Random) to initiate connection
    - SYN bit will be sent in this packet to synchronize with the server
2. STEP 2 (ACK SYN)
    - Server will send ACK bit set to the client to acknowledge that it has received the seq num
    - In this packet, an acknowledgment number will also be sent.
    - The Acknowledgement number is the next sequence number the server expects from the client
    - In the Same packet, the server will send its own **SYN bit** and Sequence Number
3. STEP 3 (ACK)

- Client will acknowledge the packet sent to it in STEP 2 via another packet
- This packet contains the next sequence number, acknowledgment number from the client side, and ACK bit

THESE 3 STEPS ESTABLISH THE CONNECTION, AND HENCE DATA CAN BE TRANSFERRED BACK AND FORTH
When FIN bit is sent, it means that the connection is to be destroyed

5 Making a Single Threaded Web Server

- A single-threaded web server Handles one request at a time in a single sequence of execution.
- Easier to implement and debug due to the absence of concurrent processing.
- Uses fewer system resources compared to multi-threaded servers, avoiding context-switching overhead.

Steps followed to implement the server

5.0.1 Initialize a basic TCP Listener

Initializing the TCP listener involves creating a server socket bound to a specific IP address and port. This socket listens for incoming TCP connection requests from clients. The server enters a loop where it waits for connection attempts. When a client initiates a connection, the listener accepts it, establishing a TCP connection. This connection is represented by a stream, which enables bidirectional data transfer between the server and client. Error handling mechanisms ensure the server can gracefully handle issues such as binding failures or connection errors, thereby maintaining robustness and reliability in network communication. This setup is crucial for building scalable network services and applications.

5.0.2 Handling the TCP Stream

Managing the incoming TCP stream involves initializing a buffered reader (**BufReader**) to handle the incoming data. As data arrives, the function reads and parses the initial request line by splitting by the whitespace present in the HTTP Request and storing the various parameters of the request in a vector of strings this completes the extraction of essential HTTP details: the HTTP method, target URI, and HTTP version.

5.0.3 Implement functionality to handle various possible HTTP status codes

After parsing the http request and obtaining the http method of the request , http status codes like 200 , 404 , 403 etc can be easily implemented .The implemented server first checks if the request method is "GET" and the HTTP version is "HTTP/1.1". If so, it determines the requested file based on the target URI, defaulting

to "index.html" if the URI is "/". Using match with the result of 'fs::metadata', if the file exists , it reads its contents and prepares a 200 OK response with the file's contents and appropriate headers. If there's an error accessing the file , it responds with a 403 Forbidden status. If the file doesn't exist , it responds with a standard 404 Not Found error.

If the request method is not "GET", it responds with a 405 Method Not Allowed status, indicating that the server does not support the requested method

5.0.4 Final Implementation

The final implementation was made to handle-

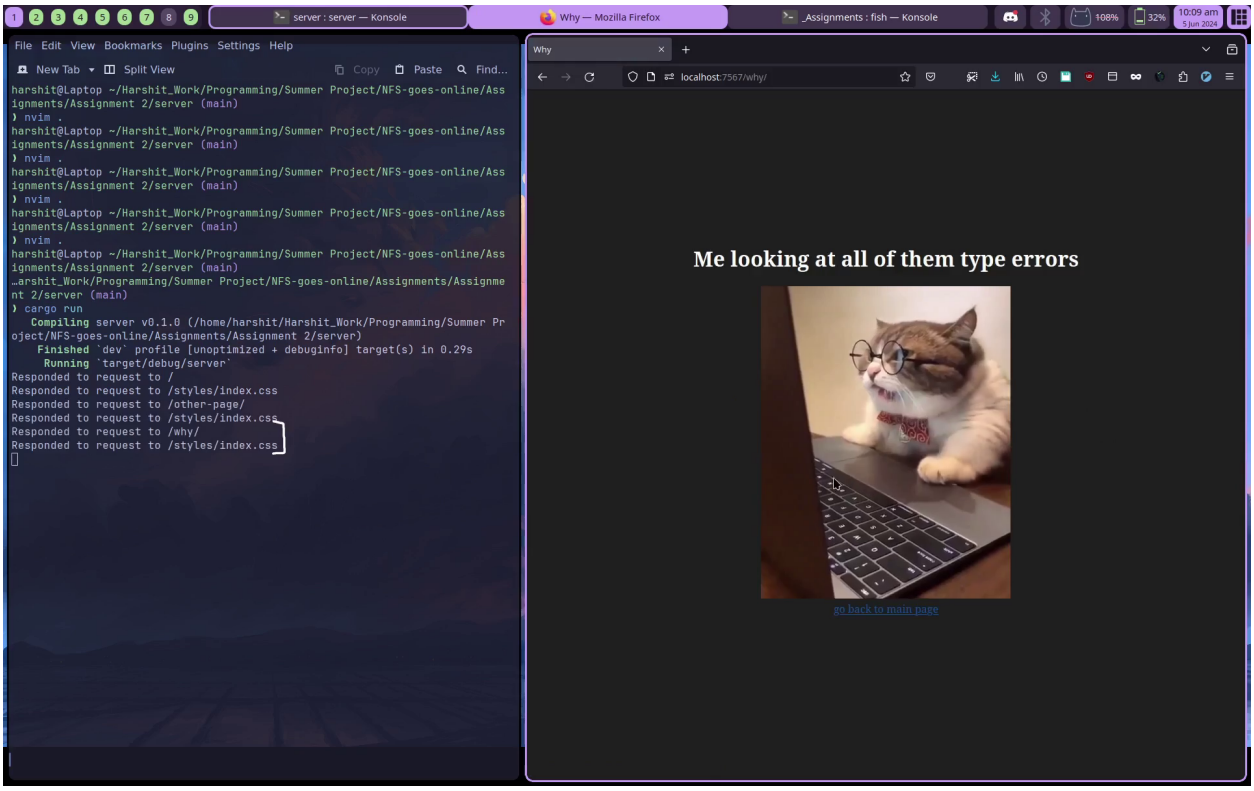


Figure 4: Requests with 200 Responses

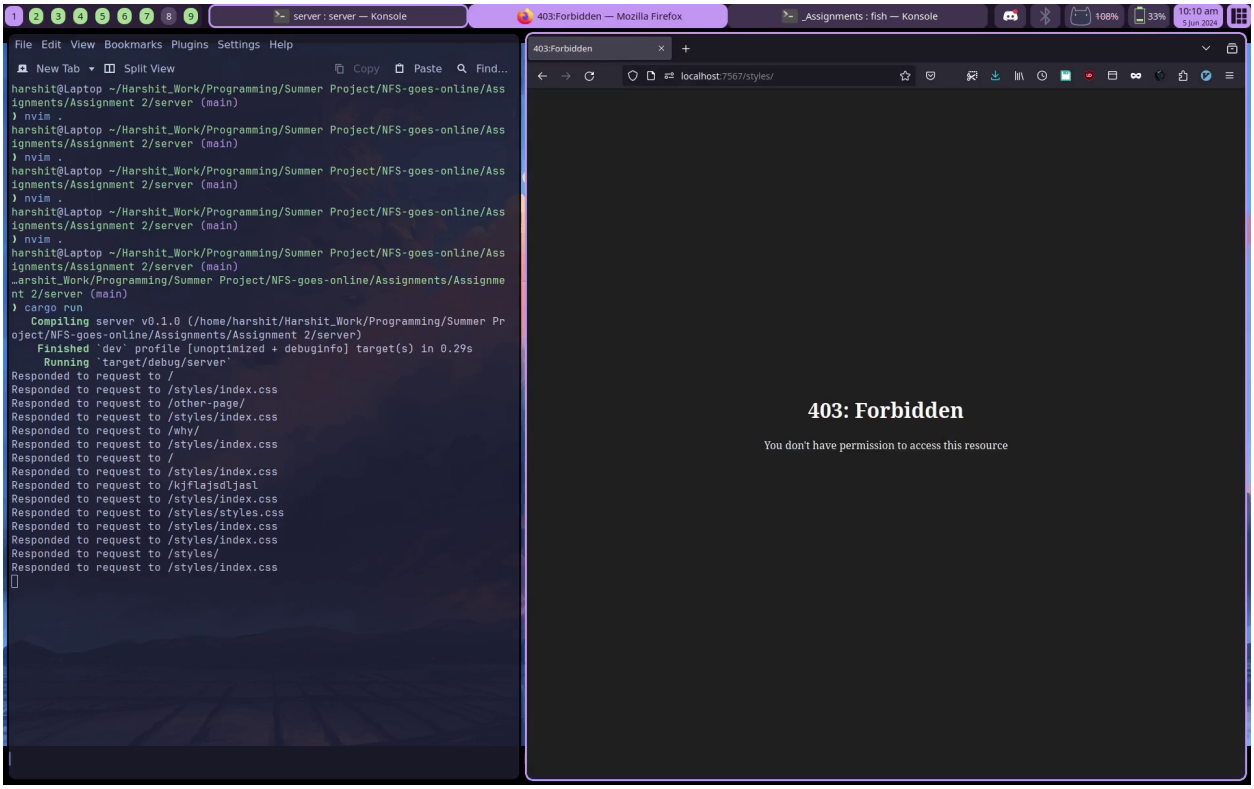


Figure 5: Requests with 403 Responses

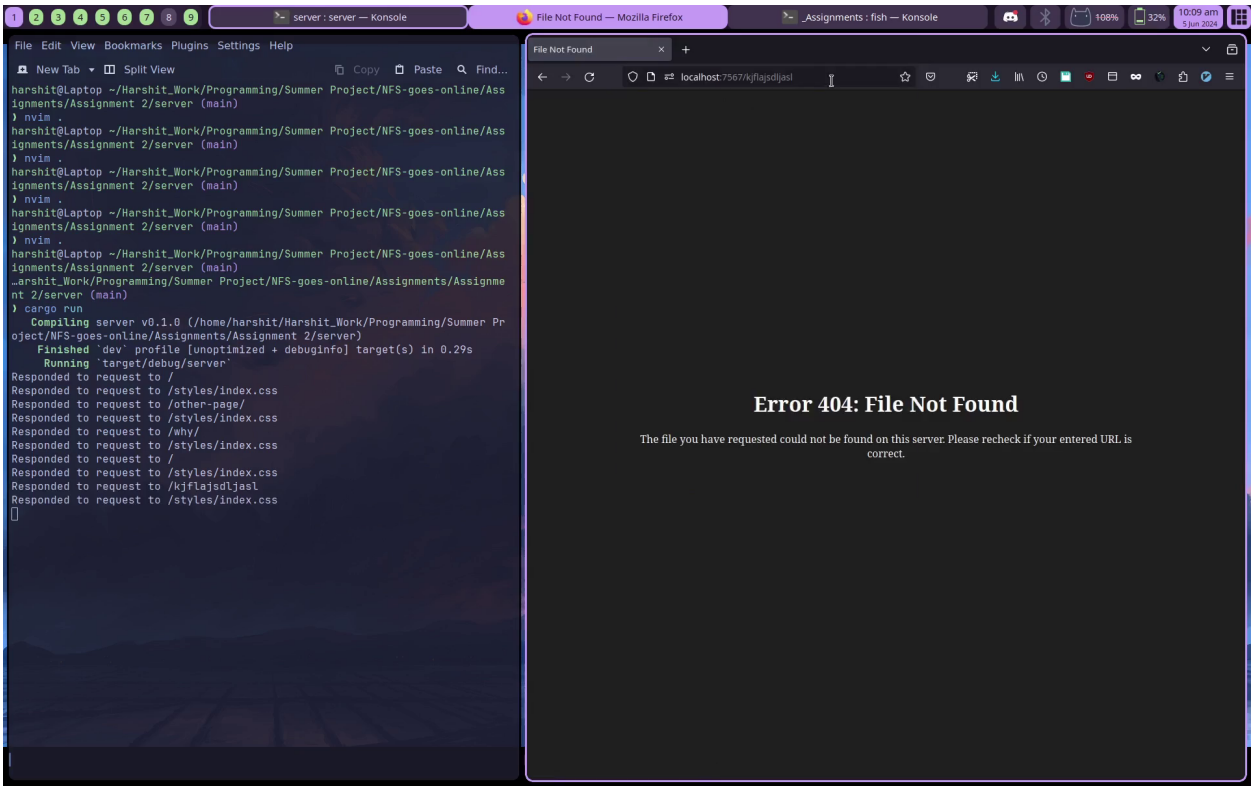


Figure 6: Requests with 404 Responses

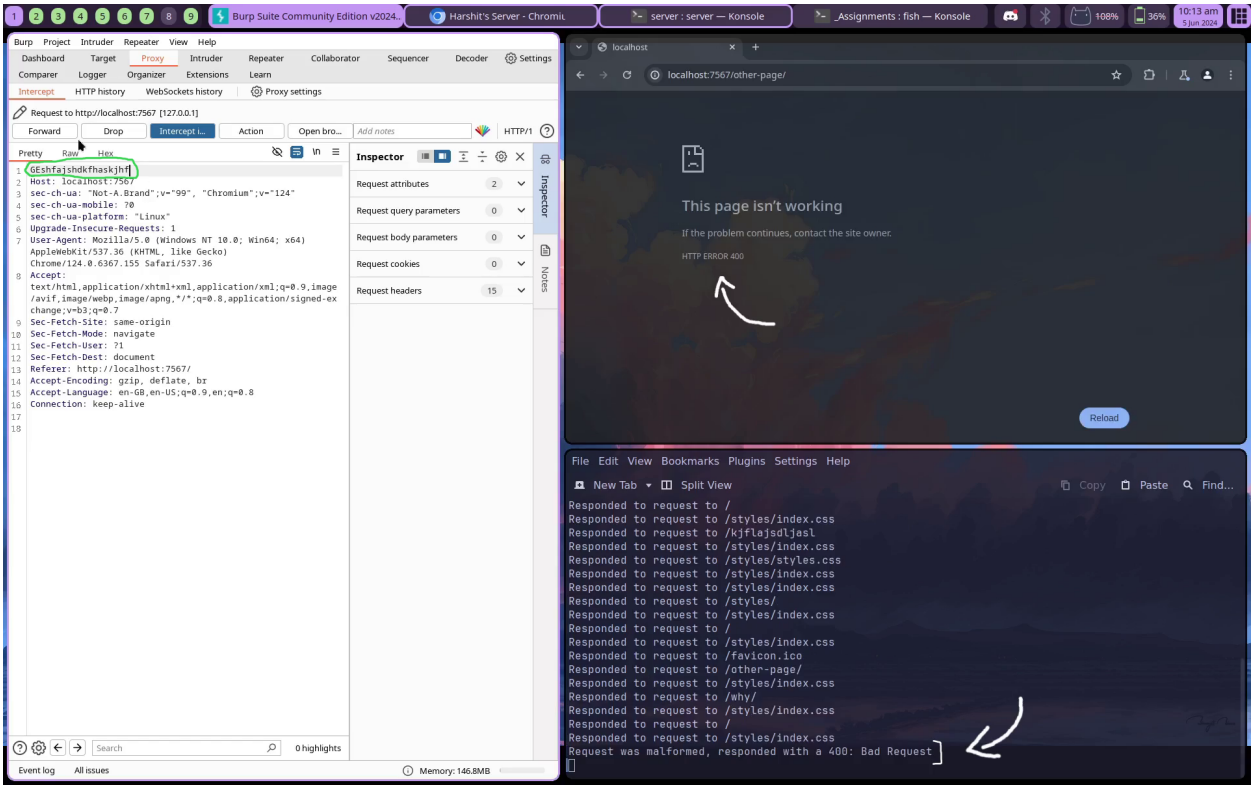


Figure 7: Malformed Requests (400 Bad Request)

6 Operating Systems

Operating System is software that converts hardware into a useful form for applications.
Role of Operating Systems:

1. Provides **Abstractions** to expose the Applications to the Hardware (provide a standard library).
2. **Resource Management**: To develop mechanisms and policies to provide efficient and fair access to resources for various applications. Moreover, this protects applications from one another.

Abstractions which OS provides for various hardware components:

- CPU: Process or Thread
- Memory: Virtual Address Space

6.1 Virtualization

The objective of Virtualization is to give each process the impression that it alone is actively using the CPU.

6.1.1 Processes

Simply, a Process is a running program. It is a stream of executing instructions and their context in the address space.

6.1.1.1 Process Creation

The code and static data of the program that is used to run the process resides in the persistent storage. A process is forked (created) by loading (copying) the data from the persistent storage to the Memory (address space of the process) in an executable format, initializing the process registers to some known state, initializing the static data to the initial values defined by the program, and initializing the empty heap and stack. For this to happen, the OS reads those bytes from disk and places them in the memory.
This loading process happens **lazily**, which means by loading pieces of code or data only as they are needed during program execution. This is opposed the loading process in early OSes, which was an **eager loading**, i.e. loading was done all at once before running the program.

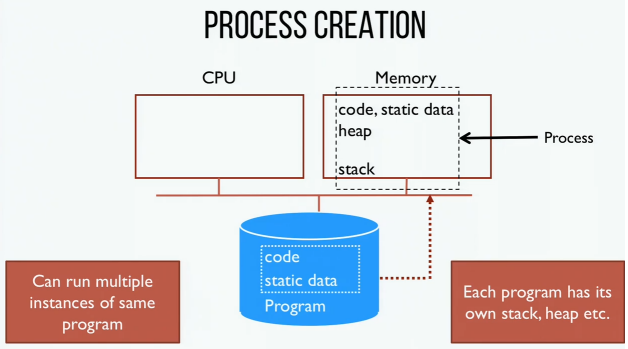


Figure 8: Process Creation

6.1.1.2 Memory allocation in Process Creation

As mentioned earlier, during process creation, stacks and heaps are initialized in the memory. In this section we'll see how exactly memory allocation takes place and what is the utility of stacks and heaps.

- **Stack** is used for local variables, function parameters, and return addresses. The OS allocates this memory and allocates to the process.
- **Heap** us used for explicitly requested dynamically-allocated. C Programs can request this space my calling malloc() and free it explicitly by calling free(). The heap is small at first, but can be expanded by requesting more memory via malloc().

6.1.1.3 Machine State of a Process

The Machine State of a Process simply defines what a program can read or update when it is running. Following are the components of the Machine State:

- Memory and Address Space: Memory, which contains the instructions (program) and data is a part of the Machine State of a Process. However, the process can only access the memory which is a part of its address space. It can't access all the memory.
- Registers: They are small storage areas inside the CPU. They hold data temporarily while the CPU processes instructions. Registers help the CPU work faster by providing quick access to important information needed for calculations and operations. Registers are of various types but some important ones are:
  - **Program Counters (PC)**: tells us which instruction of the program will execute next
  - **Stack Pointer and Frame Pointer**: used to manage the stack for function parameters, local variables, and return addresses.
- Some parts of the Persistent Memory, which the process has access to.

6.1.1.4 The Problem

More than often, we need to run multiple processes on our CPU, because of convenience. However, we have a limited number of CPUs on which the processes can run. So, we need to provide an illusion that there is a nearly endless supply of CPUs, so that a huge number of processes can be run.

6.1.1.5 Solution: Virtualization

This "illusion" is created via **Virtualization** of the CPU. This can happen via running one process, then stopping it and running the other process, and so on. This creates an illusion that multiple CPUs are present and can be accessed by multiple processes, despite only one (or a few, based on the number of cores) physical CPU being present. This mechanism is called **Time Sharing**, which allows the users to run multiple concurrent processes.
Let's say there are three processes: A, B, and C; and assume that the system has only a single-core CPU. Since all the 3 processes can't run on the CPU at the same time, first the context of A will be extracted from the memory to the CPU, and process A will run, while B and C will be at a pause. After some time, A will be paused, its context will be stored in the memory, the context of B will be loaded from the memory and B will start running. This cycle shall continue. This job is scheduling is done by the **OS Scheduler**, and the saving of a context and loading of another context into the CPU from the memory is called **Context Switch**.

6.1.1.6 Mechanisms and Policies

For implementing virtualization, two things are required:

1. **Policies**: These are high-level algorithms that choose which activities to perform. They necessarily don't implement any functionality but make a decision. For instance, a **Scheduling Policy** will decide the order in which processes need to be run and for how much time each process will run before the context switch.

2. **Mechanisms**: These are low-level methods or protocols that help in the implementation of the functionality. Mechanisms are ways to implement policies. For instance, **context switch** is a mechanism that helps in the implementation of time-sharing.

6.1.1.7 State Transitions

In virtualization, we learned a process at a given time can either be running, or it can be paused. This means that we can assign can attribute to a Process called state. In this section, we shall study the formalization of Process States and how transitions occur among these states. A Process can be in one of the three states:

1. **Running**: Process is executing instructions.

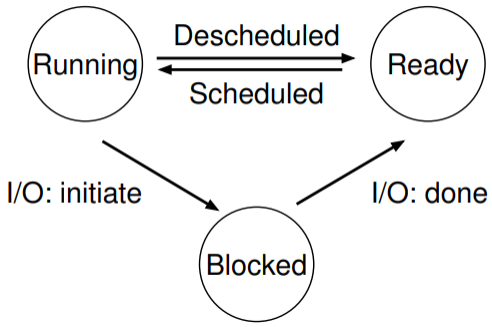


Figure 9: Process State Transitions

2. **Ready**: Process is ready to run, but OS has made the decision to not run the process at that time. When we informally said that in Time Sharing, all processes except one are paused, it means that they are in Ready state, but not Running.

3. **Blocked**: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. After that operation has been performed, it is transferred to the Ready State.

6.1.1.8 Process List

A process list contains information about all processes in the system. Each entry is found in what is sometimes called a process control block (PCB), which is really just a structure that contains information about a specific process.

6.1.2 Process Creation in UNIX systems

Process Creation inn UNIX systems is handled via system calls: fork() and exec(), along with a third routine wait().

6.1.2.1 fork() System Call

It is used to create a new process.

6.1.3 Direct Execution

6.1.3.1 Challenges in Execution of Processes

1. If the process wants to do something restricted, it should either not be allowed access, or the user should explicitly be asked whether to provide access or not.

2. The process might be malicious and buggy, leading to the process running forever

6.2 File Systems and Implementation

6.2.1 What is a File System?

A file system, is a key operating system abstraction, that organises data in persistent memory (hard drive) predefined ways so as to optimize information storage, retrieval and modification. Most file systems allow for further abstractions such as files and directories.
The file system is pure software, unlike other core aspects of the OS, there will be no hardware features that improve upon its performance. Because of this great versatility, there are multiple kinds of file systems that have been built, each with their own specificities, benefits, and usecases.
In this project, we shall use the Very Simple File System (or vsfs) to demonstrate the concept and it will also be the file system we implement.

6.2.2 Files and Directories

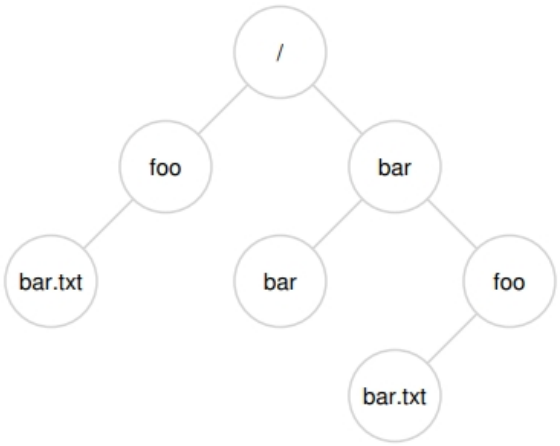


Figure 10: An example directory tree

A **file** is simply a linear array of bytes, stored with some user-level name, and an system wide, low-level name. The low-level name is called an **inode number, or i-number**. This can be used to uniquely identify each entity in the permanent storage.

A **directory**, like a file also has a low level name - an inode number, but its contents are a list of `(user level name, low level name)` pairs. Each entry in a directory refers to either files or other directories.
By placing directories within other directories, users are able to build an **arbitrary directory tree (or directory hierarchy)**, under which all files and directories are stored.

In UNIX based systems, the directory hierarchy starts at the root directory, which is usually referred to as `/`, and then goes down to all files and folders in storage. In Figure 10, we see that we could refer to the file

`bar.txt` in the directory `foo` within the directory `bar` using its **absolute pathname** which in this case will be `/bar/foo/bar.txt`.

Also, most files seem to have two parts in their filename separated by a period. This is entirely arbitrary, used to indicate the type of the file - whether it is a document, a zip file, or a video file. There is usually no enforcement on the data. The responsibility of the file system is to simply store the data on the disk and give the appropriate response when that data is requested again.

6.2.3 Overall Organisation of a File System

File systems like vsfs segment the disk into **blocks**, let us take a size of 4KB for the purposes of this project. For example in a 64 block file system (which means a total storage space of $64 \times 4 = 256$KB). We designate every block with a specific role and type of data to contain.

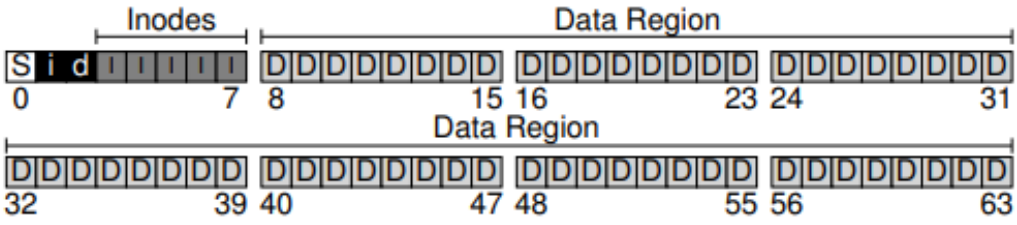


Figure 11: Organization of a 64 block disk

6.2.3.1 Data Region

The **data region**, as suggestible from its name, constitutes majority of the storage and contains user data.

6.2.3.2 Inodes

The **Inodes**, as earlier explained tracks information of each file and directory, some of this could be its size, whether it is a file or directory, and its location in the data blocks.
Inodes are typically not that big, usually 128 or 256 bytes per inode. So the number of possible inodes denotes the total number of files and folders we can have. Although we could simply allocate another block and thus accommodate more files. This will be further elaborated in the next section.

6.2.3.3 Bitmaps

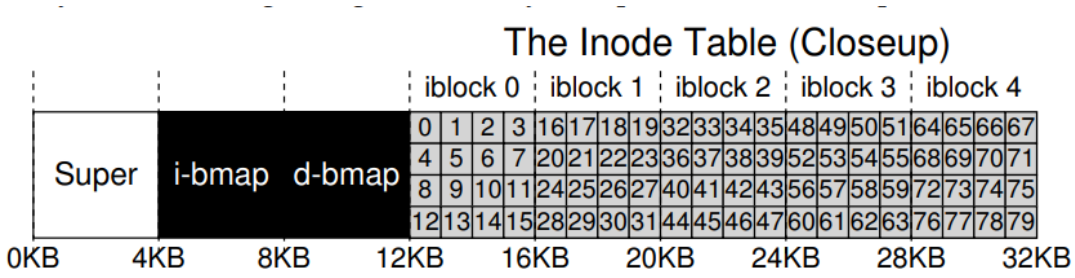
Our file system does have datablocks and inode blocks, but no way to keep track of which ones are in use for a file/folder. Thus another structure called a **bitmap** is vital in tracking singular inodes and datablocks. A bitmap is in essence a collection of bits, and the the value at each index indicates whether the corresponding object/block is free(0) or in-use(1).

The **data bitmap** (block at index 2) is responsible for keeping track of each data block. So the value of every bit in the data block corresponds to the availability of the data block. In this specific case, we have 56 useful bits in the data bitmap.
The **inode bitmap** (block at index 1) has a similar purpose, each bit represents whether the equivalent inode in the inode blocks is in use or not.
Therefore in order to access, modify, or issue new blocks/inodes, we refer to the bitmaps, modifying them to reflect any changes.

6.2.3.4 Super Block

The first block is reserved for the **superblock**. The superblock contains information about this specific file system, including the total number of data blocks, size of each block, total number of inodes, it will also include a magic number of some sort to identify the file system type (in our case, vsfs).

6.3 A Closer Look - Inodes



The **inode, or indexed node** is the name for the structure that, in many file systems holds the metadata of a given file, such as its length, permissions, location in the datablocks, etc. In vsfs and aligned file systems, given an i-number, you should directly be able to calculate where on disk where the corresponding inode is located.
A crucial choice must be made with regards to how an inode refers to where data blocks are. A simple approach would be to use one or more **direct pointers (or disk addresses)** that point to specific datablocks that contain the file. But this approach is appropriate only for systems having small files.
For systems with large files: Usage of a **indirect pointer** - wherein the inode points to a block that contains even more pointers, each of which point to datablocks.

In order to read any inode given the i-number, the file system would first look into the inode bitmap to check the validity of the i-number. Lets say we want to obtain inode with i-number 42. From the superblock we have `inodeStartAddr = 12KB`, and the specific inode `inodePos = 42*sizeof(inode)`. Therefore the position of the inode 42 becomes `inodeStartAddr + inodePos`. This can be used to locate and read any inode.

An important aspect to note is that in the case of hard disks, since they are not byte accessible (a hard disk has a circular platter on which data is stored by induction of magnetic charges using a spindle), we divide its circular surface into sectors of a predetermined size and so the sector address can be accessed by

```
blk = (inumber * sizeof(inode)) / blockSize
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links	count how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file acl	a new permissions model beyond mode bits
4	dir acl	called access control lists

Table 1: A Ext2 Inode

Thus, inodes can carry varied types of information, depending on the sophistication and complexity of the file system. Simpler file systems like vsfs design inodes only to hold information like size, type - file/directory, direct pointers etc.

6.4 Directory Organisation

Most file systems have a very similar organisation for directories; a list of `(user name, inode number)` pairs. From Figure 10 the on-disk data for directory `bar` looks like:

inum	reclen	strlen	name
0	12	2	.
2	12	3	..
4	12	4	bar
5	12	4	foo

Table 2: A Ext2 Inode

Here the inode numbers are taken with no specific pattern, and are solely to demonstrate the concept at hand. In this example, each entry has an inode number, a record length(total bytes allocated for the name),

and string length (the actual length of the name) followed by the name itself,
Note that each entry has two entries . and .. the `"."` directory is just the current directory (`bar` in this case), whereas the `".."` directory refers to the parent (the root, or `"/"` in this case).

Since file systems treat directories as just special files, this array of pairs is stored in the datablocks, with an inode pointing to it and the "`type`" field marked as "directory" instead of "regular file". This keeps everything else unchanged.

6.5 Flow of Operations

Now that we have established how files and directories are stored, we should be able to understand the procedure of reading and writing files. This is where it gets interesting.

6.5.1 Reading

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0] `"."`	bar data[1] `".."`	bar data[2] `"bar"`
open(bar)			**read**	**read**	**read** (dir size)	**read** (children)	**read** (children)			
read()					**read** (index 0) **write** (update access)			**read**		
read()					**read** **write** (update access)				**read**	
read()					**read** **write** (update access)					**read**

Table 3: `"/foo/bar"` read timeline, **time increases downward**

In the example directory tree, let us open a file `"/foo/bar"`, and close it. Let us simulate what happens when you use the `open()` system call.
The file system first needs to identify the inode for `bar`, but all we know is its absolute path, from root (in `/foo/bar`). Thus the file system must **traverse** the path name and locate the inode.

1. All traversals begin at the root (or `/`), which in all file systems has a predefined inode number (0 in vsfs).
2. Once this inode is read, the file system looks into the sub-directories/files inside, cross matching with the `/`-separated path name given.
3. The file system does this recursively until the desired inode is found.
4. `open()`, thus does a final permissions check, and allocates a file descriptor pointing to the datablocks mentioned in the inode and returns to the user.

Using the file descriptor, the program can now issue a `read()` call to read from the file.
In this example, it reads the block containing the inode of `/foo` and then its directory data, finding the inode for `bar`. Depending on the file system, calling `open()` may also require an update of the inode with a new last-accessed time.
Thus `read()` will take care of changing the file offset in the file descriptor such that every read points to the correct data block.

6.5.2 Writing

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0] (block)	bar data[1] (block)	bar data[2] (block)
create (`/foo/bar`)		**read** **write** (new)	**read**	**read** **write**	**read** **write**	**read**	**read** **write**			
write()	**read** **write** (new)				**read** **write** (block num)			**write** (file data)		
write()	**read** **write**				**read** **write**				**write** (file data)	
write()	**read** **write**				**read** **write**					**write** (file data)

Table 4: `"/foo/bar"` create+write timeline, **time increases downward**

Writing is a similar process, but unlike reading, a write may entail having to allocate a new block for the file, in the situation where it exceeds the block size. But then it has to check which block to allocate to the file and thus update other structures accordingly(bitmaps, inode).

We shall look into a situation where `/foo/bar` is created, and three blocks are written to it. Table 6.5.2 shows what happens during the creation and the 3 subsequent writes to the file(for each block).
A cursory overview of the entire create/write procedure:

1. The file system recursively moves through the path to find the parent directory where the desired file shall be created.

2. Check for other objects with the same name in parent directory data.

3. Allocate a new inode for file, update inode bitmap.
4. Write new file name and inode in parent directory data.
5. Update both parent and child inodes.
6. Allocate data blocks and write to data blocks.
7. Update file inodes with direct pointers.
8. Allocate and write until completion.

How file systems solve I/O traffic

As we can see, in this entire process, the amount of I/O traffic is much worse than read, hence many file systems use a caching and buffering mechanism to ease I/O congestion. In this, popular blocks are held in a **fixed-size** cache in the system memory(DRAM). **Cache replacement policies** such as LRU, TLRU and MRU would decide which blocks to keep in cache. This cache would be allocated usually at boot time and is roughly 10% of total memory.

6.6 Distributed Systems

When a user interacts with a with a web service, there is a large collection of machines working to provide different parts of the service.
A distributed system is a computer program utilising computations from various such devices (called computational nodes) in order to complete the same process.
A major goal in distributed systems is to prevent the failure of entire system in case a sub-process fails, in short, to project to the client a non-failing server.
There are other issues such as security and performance which we would not be considering due to the several levels of intricacies involved in them. As we have already seen the basics of networks and various protocols used at different layers, let us look at some of the possible problems in communication and their possible solutions.

6.6.1 Packet loss and reliable communication layers

As we know, network protocols break a message into data packets before sending them over to the receiver. There is a common checksum method employed. UDP/IP protocols use the **checksum** method to ensure that all data has been received, but this method does not let receiver be informed about packet loss. TCP protocol involves **syn-ack** procedure to communicate receival of packets to the sender, and the **timeout** mechanism, which after a specified time interval of not receiving acknowledgement, considers the packet lost. However, this mechanism clearly poses a problem if it is the acknowledgement that is lost and not the

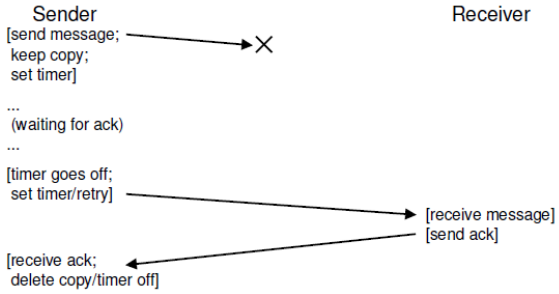


Figure 12: Timeout/Retry Mechanism

packet.

The condition calls for an additional, albeit requiring some more memory, mechanism called **sequence counter**. In this mechanism, the sender and receiver maintain an individual counter of packets starting from a common number (say, 1). The counter value is sent along with the packet, and if it agrees with the receiver's counter value, the receiver increments its counter by 1. So if the acknowledgement for packet *N* has been lost, and due to timeout, the sender sends the packet again, this time the receiver would have counter value *N+1*, so only acknowledgement would be sent, but the packet will not be added to data.

6.6.2 RPC: Remote Process Call

Communication abstractions have to be used when building a distributed system.

DSM(distributed shared memory) is a popular OS abstraction which enables processes running in different systems to share the same Virtual Memory Space. The obvious problem is that data structures using memory, which is spread across different devices, would run into complications if a machine fails. Also, in case the OS tries to access a page already not present on machine, it causes a page fault and fetching of data from the other machine where the page is open locally, which is a quite expensive process and makes access to memory space not as simple as would be on a simple machine.

Thus, the more widely used and accepted is the PL(Programming Language) abstraction of **RPC**.

7 Building a Network File System using C

7.1 A Basic File Server

Our file server is built as a stand-alone UDP-based server. It should wait for a message and then process the message as need be, replying to the given client.

Our file server will store all of its data in an on-disk, fixed-sized file which will be referred to as the file system image. This image contains the on-disk representation of our data structures; we use these system

calls to access it: open(), read(), write(), close().

To access the file server, we build a client library. The interface that the library supports is defined in mfs.h. The library should be called libmfs.so, and any programs that wish to access our file server will link with it and call its various routines.

7.2 On-Disk File System: A Basic Unix File System

The on-disk file system structures follow that of the very simple file system. On-disk, the structures are as follows:

- A single block (4KB) super block
- An inode bitmap (can be one or more 4KB blocks, depending on the number of inodes)
- A data bitmap (can be one or more 4KB blocks, depending on the number of data blocks)
- The inode table (a multiple of 4KB-sized blocks, depending on the number of inodes)
- The data region (some number of 4KB blocks, depending on the number of data blocks)

More details about on-disk structures can be found in the header ufs.h. Specifically, this has a very specific format for the super block, inode, and directory entries. Bitmaps just have one bit per allocated unit.

As for directories, here is a little more detail. Each directory has an inode, and points to one or more data blocks that contain directory entries. Each directory entry should be simple, and consist of 32 bytes: a name and an inode number pair. The name should be a fixed-length field of size 28 bytes; the inode number is just an integer (4 bytes). When a directory is created, it should contain two entries: the name . (dot), which refers to this new directory's inode number, and .. (dot-dot), which refers to the parent directory's inode number. For directory entries that are not yet in use (in an allocated 4-KB directory block), the inode number should be set to -1. This way, utilities can scan through the entries to check if they are valid.

When our server is started, it is passed the name of the file system image file. The image is created by a tool we've made called mkfs.

When booting off of an existing image, our server reads in the superblock, bitmaps, and inode table, and keep in-memory versions of these. When writing to the image, the system updates these on-disk structures accordingly.

Importantly, we cannot change the file-system on-disk format.

7.3 Client library

- int MFS_Init(char *hostname, int port): MFS_Init() takes a host name and port number and uses those to find the server exporting the file system.
- int MFS_Lookup(int pinum, char *name): MFS_Lookup() takes the parent inode number (which should be the inode number of a directory) and looks up the entry name in it. The inode number of name is returned. Success: return inode number of name; failure: return -1. Failure modes: invalid pinum, name does not exist in pinum.
- int MFS_Stat(int inum, MFS_Stat_t *m): MFS_Stat() returns some information about the file specified by inum. Upon success, return 0, otherwise -1. The exact info returned is defined by MFS_Stat_t. Failure modes: inum does not exist. File and directory sizes are described below.
- int MFS_Write(int inum, char *buffer, int offset, int nbytes): MFS_Write() writes a buffer of size nbytes (max size: 4096 bytes) at the byte offset specified by offset. Returns 0 on success, -1 on failure. Failure modes: invalid inum, invalid nbytes, invalid offset, not a regular file (because you can't write to directories).
- int MFS_Read(int inum, char *buffer, int offset, int nbytes): MFS_Read() reads nbytes of data (max size 4096 bytes) specified by the byte offset offset into the buffer from file specified by inum. The routine should work for either a file or directory; directories should return data in the format specified by MFS_DirEnt_t. Success: 0, failure: -1. Failure modes: invalid inum, invalid offset, invalid nbytes.
- int MFS_Creat(int pinum, int type, char *name): MFS_Creat() makes a file (type == MFS_REGULAR_FILE) or directory (type == MFS_DIRECTORY) in the parent directory specified by pinum of name name. Returns 0 on success, -1 on failure. Failure modes: pinum does not exist, or name is too long. If name already exists, return success.
- int MFS_Unlink(int pinum, char *name): MFS_Unlink() removes the file or directory name from the directory specified by pinum. 0 on success, -1 on failure. Failure modes: pinum does not exist, and the directory is NOT empty. Note that the name not existing is NOT a failure. Why? Idempotency! By not treating the absence of the file or directory as an error, the filesystem API simplifies the logic for the client applications. Clients do not need to explicitly check for the existence of the file before attempting to remove it. They can call MFS_Unlink directly and rely on the return value to indicate success or failure due to other reasons.

- int MFS_Shutdown(): MFS_Shutdown() just tells the server to force all of its data structures to disk and shutdown by calling exit(0). This interface will mostly be used for testing purposes.
- Size: The size of a file is the offset of the last valid byte written to the file. Specifically, if you write 100 bytes to an empty file at offset 0, the size is 100; if you write 100 bytes to an empty file at offset 10, the size is 110. For a directory, it is the same (i.e., the byte offset of the last byte of the last valid entry).

7.4 Server Idempotency

The key behavior implemented by the server is idempotency. Specifically, on any change to the file system state (such as a MFS_Write, MFS_Creat, or MFS_Unlink), all the dirtied buffers in the server are committed to the disk. The server can achieved this end by calling fsync() on the file system image. Thus, before returning a success code, the file system should always fsync() the image.

Why do this? Simple: if the server crashes, the client can simply timeout and retry the operation and know that it is OK to do so.

How do we implement a timeout? Simple, with the select() interface. The select() calls allows us to wait for a reply on a certain socket descriptor (or more than one, though that is not needed here). we can even specify a timeout so that the client does not block forever waiting for data to be returned from the server. By doing so, we can wait for a reply for a certain amount of time, and if nothing is returned, try the operation again until it is successful.

7.5 UDP Code

- **client.c:** example client code, sends a message to the server and waits for a reply.
- **server.c:** example server code, waits for messages indefinitely and replies.

Both use udp.c as a simple UDP communication library.

7.6 Program Specifications

The server program must be invoked exactly as follows:

```
prompt> server [portnum] [file-system-image]
```

The command line arguments to the file server are to be interpreted as follows.

- **portnum:** the port number that the file server should listen on.

- **file-system-image:** a file that contains the file system image.

If the file system image does not exist, we print out an error message (image does not exist) and exit with exit code 1.

client library is called libmfs.so. It implements the interface as specified by mfs.h, and in particular deals with the case where the server does not reply in a timely fashion; the way it deals with that is simply by retrying the operation, after a timeout of some kind (default: five second timeout).