19CSE213- OPERATING SYSTEM

CASESTUDY

MINIX OS



GROUP MEMBERS

|  |  |
| --- | --- |
| ROLL NUMBER | NAME |
| CB.EN.U4CSE22410 | C.H. ASMITHA |
| CB.EN.U4CSE22411 | C. RISHITHA |
| CB.EN.U4CSE22431 | P. TRSIHIKA REDDY |
| CB.EN.U4CSE22441 | S . INDIRA SATYA SRI |
| CB.EN.U4CSE22449 | V. SUHARSHITHA |

|  |  |
| --- | --- |
| 9 | FILE SYSTEM |
| 10 | MEMORY MANAGEMENT |
| 11 | COMPARISION WITH LINUX |

|  |  |
| --- | --- |
| 12 | THREADS AND SYNCRONIZATION |
| 13 | CRITICAL SECTION |
| 14 | ADVANCED FEATURES |

TABLE OF CONTENTS

|  |  |
| --- | --- |
| 1 | INTRODUCTION |
| 2 | SYSTEM REQUIREMENTS |
| 3 | USER AND SOFTWARE MANAGEMENT |
| 4 | INSTALLATION AND SPECIAL FEATURES |
| 5 | MINIX SYSTEM CALLS |
| 6 | SCHEDULING ALGORITHM |
| 7 | COMMANDS |
| 8 | USABILITY AS A SERVER AND  DESKTOP |

INTRODUCTION

Minix is a Unix-based operating system .It follows a microkernel architecture. Initially created by Andrew S.

Tanenbaum for educational purposes. Evolved to prioritize reliability and self

healing capabilities. Open-source, allowing access and modification of its source code.

Released in 1987, designed to be understandable for beginners.

Written primarily in the C programming language.

Easily portable to various computer architectures.

Latest version is Minix 3.3, emphasizing reliability and self-healing.

# WHERE IS THIS OS USED IN REAL LIFE :

Education: Minix is often used in universities and schools to teach students about how operating systems work. The source code is designed to be easy to understand, so students can learn by looking at and modifying the code.

Research: Researchers use Minix to study ways to make operating systems more reliable and secure. Minix has some special features that make it good for this kind of research.

Embedded Systems: Minix can also be used in small, specialized computers called "embedded systems." These are computers that are built into other devices, like your smartphone or your car. Minix works well for these kinds of systems because it is small and efficient.

# KEY POINTS IN THE OS:

Microkernel Architecture: MINIX is a microkernel operating system, which means that the core functionality of the OS is minimal and most services run as separate user-space processes. This design makes it more modular and flexible, allowing for easier customization and extension. source

🔁 Self-Healing Capabilities: MINIX has the ability to automatically restart a crashed driver without affecting running processes. This self-healing feature makes it more reliable and suitable for applications demanding high uptime. source

💾 Virtual Memory Support: MINIX 3 introduced support for virtual memory management, which allows the OS to use disk space as an extension of RAM. This enables it to run larger programs and handle more data than earlier versions.

SYSTEM REQUIREMENTS

Intel i3 or higher dual core 64-bit processor with 4GB OF RAM or higher .

Solid state drive(SSD) or hard disk drive (HDD) with atleast 32 GB of free space.

Built in or wired mouse/touchpad and keyboard ,internet connection if needed and 1024 x 768 resolution.

SOFTWARE MANAGEMENT TOOLS

GCC: The GNU Compiler Collection, providing support for compiling C, C++, and other programming languages.

Clang: An alternative C/C++ compiler that can also be used for building software on Minix 3.

Emacs: A popular text editor included in the Minix 3 distribution, useful for software development and other tasks.

Bash: The Bourne-Again SHell, a widely-used command-line interface and scripting language.

Python: A high-level programming language included in Minix 3, useful for scripting, automation, and more.

USER MANAGEMENT TOOLS

Useradd: This command is used to create a new user account. It allows administrators to specify various options such as username, home directory, and user ID.

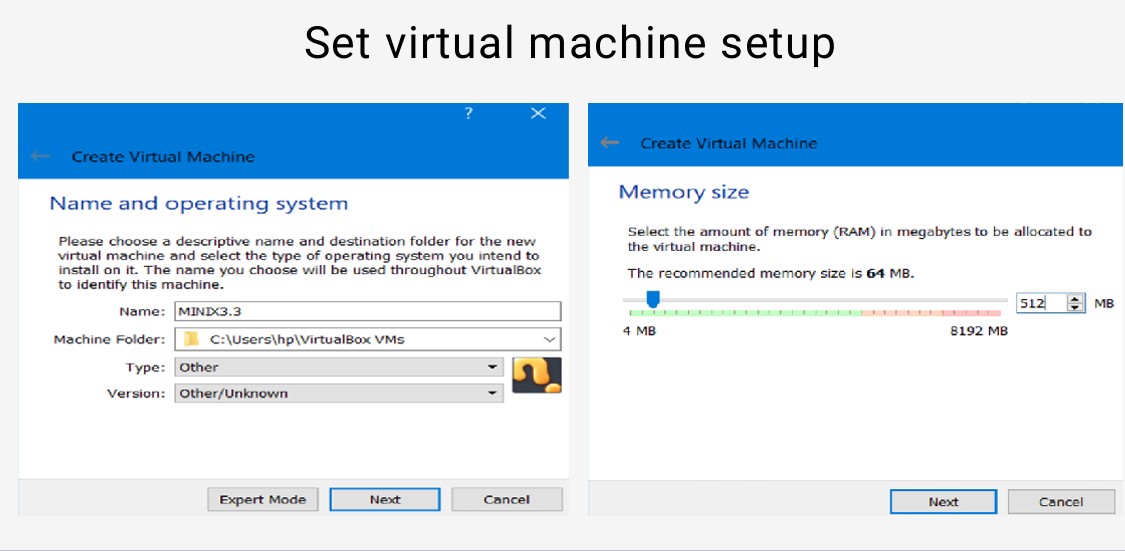
Usermod: This command is used to modify the properties of an existing user account. It can be used to change the username, home directory, user ID, or group ID associated with a user.

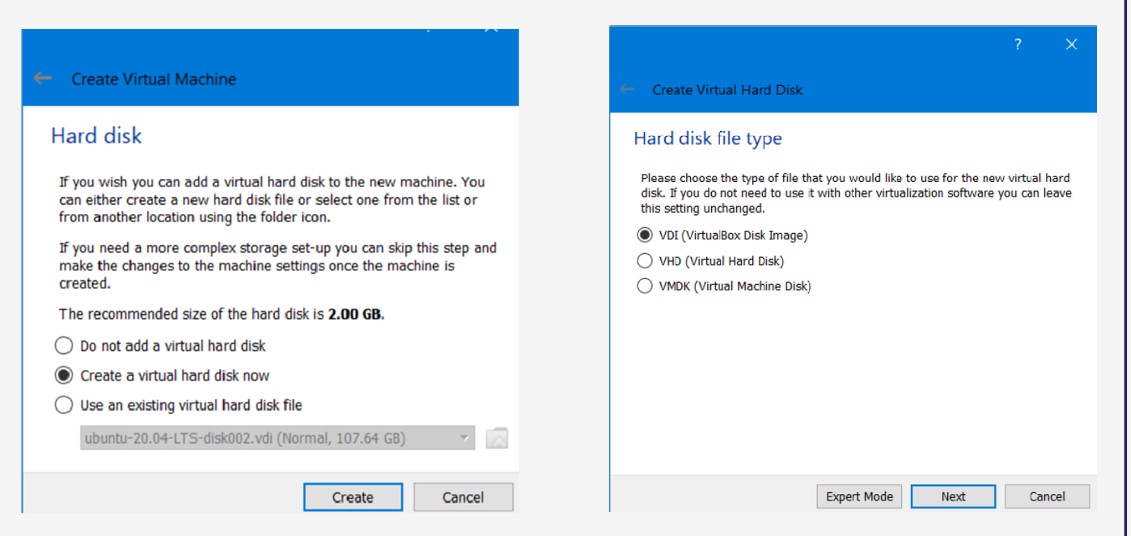
Userdel: This command is used to delete a user account. It ensures that the user no longer has access to system resources.

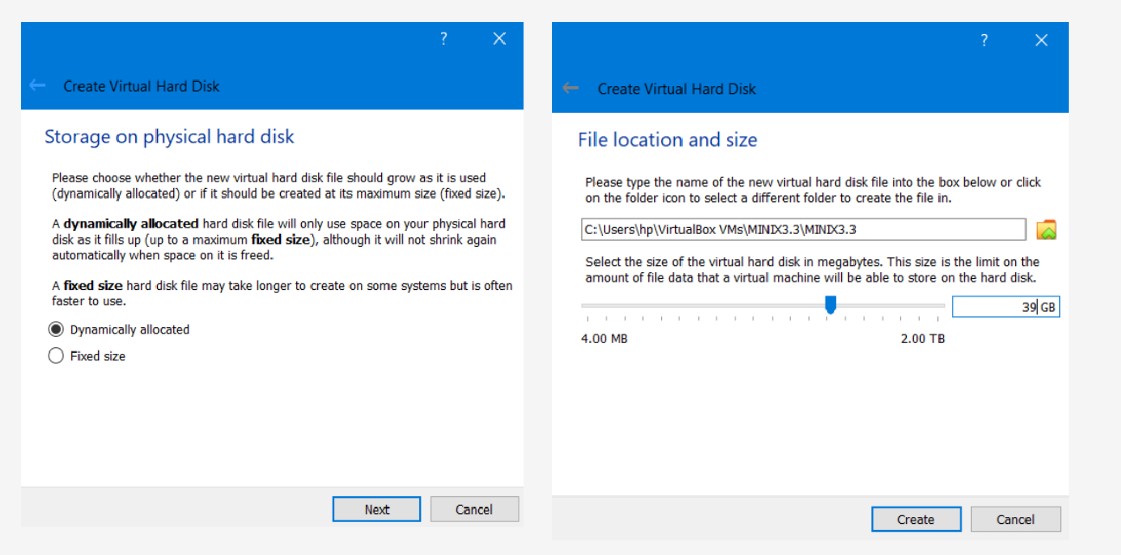
Chgrp: This command is used to change the group ownership of a file or directory.

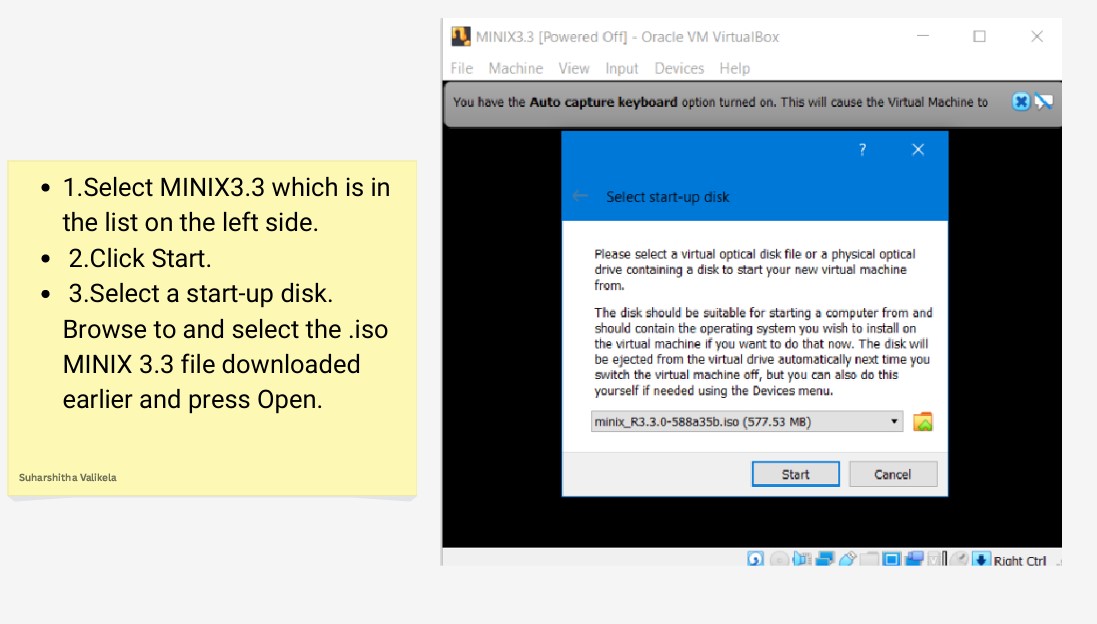
INSTALLATION STEPS

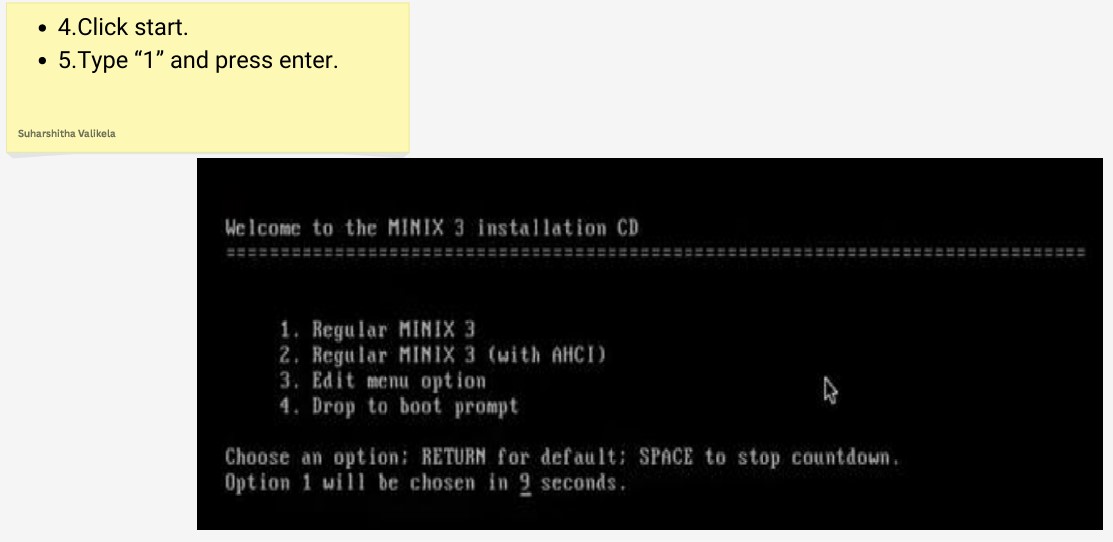
# A) VIRTUAL MACHINE :

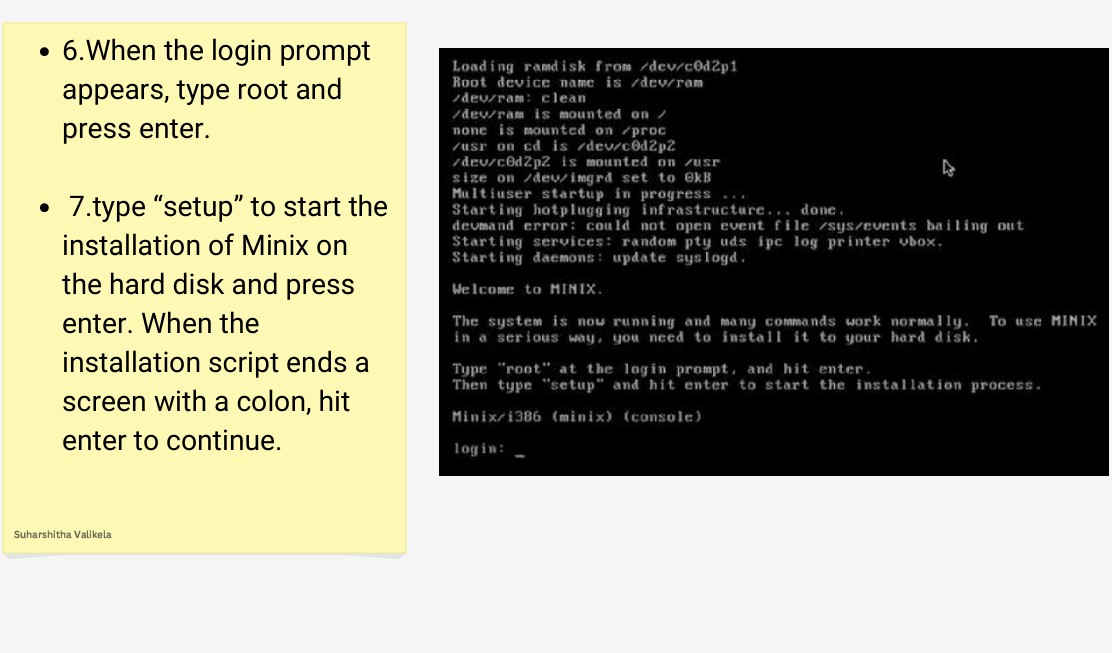


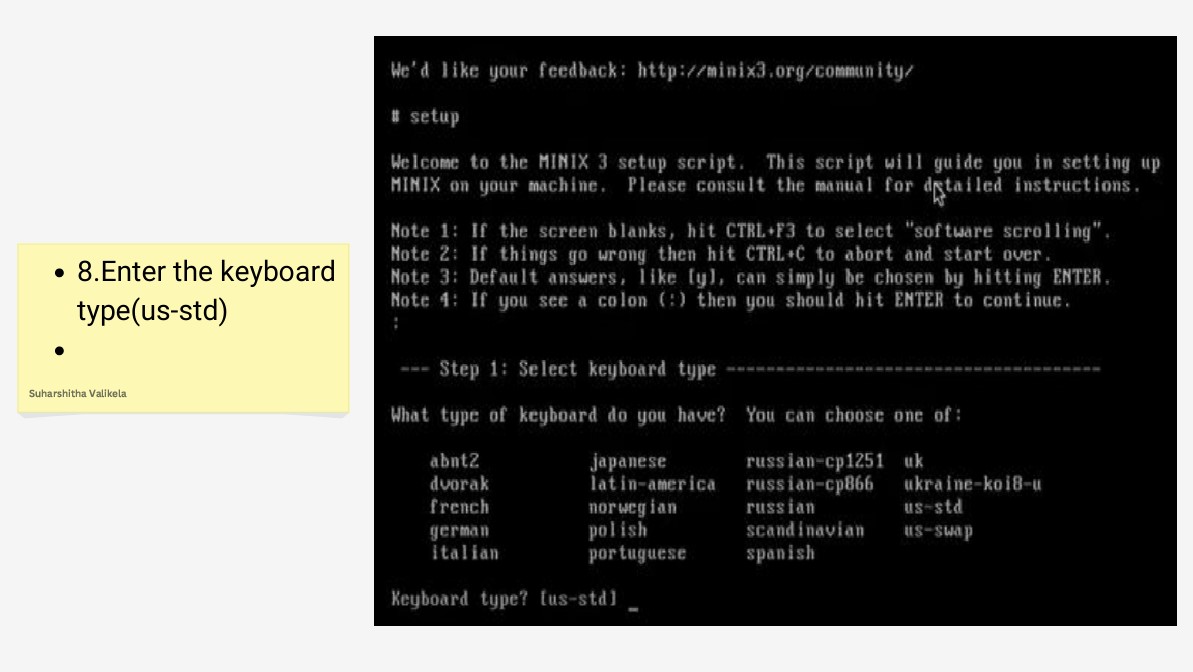


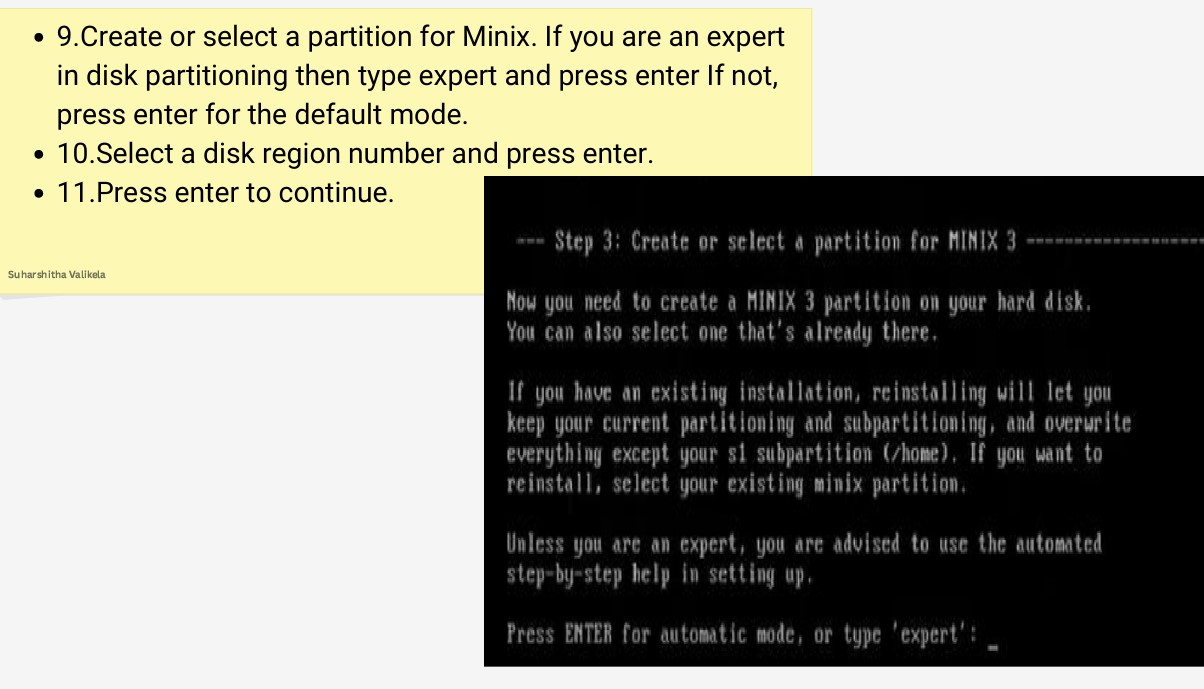


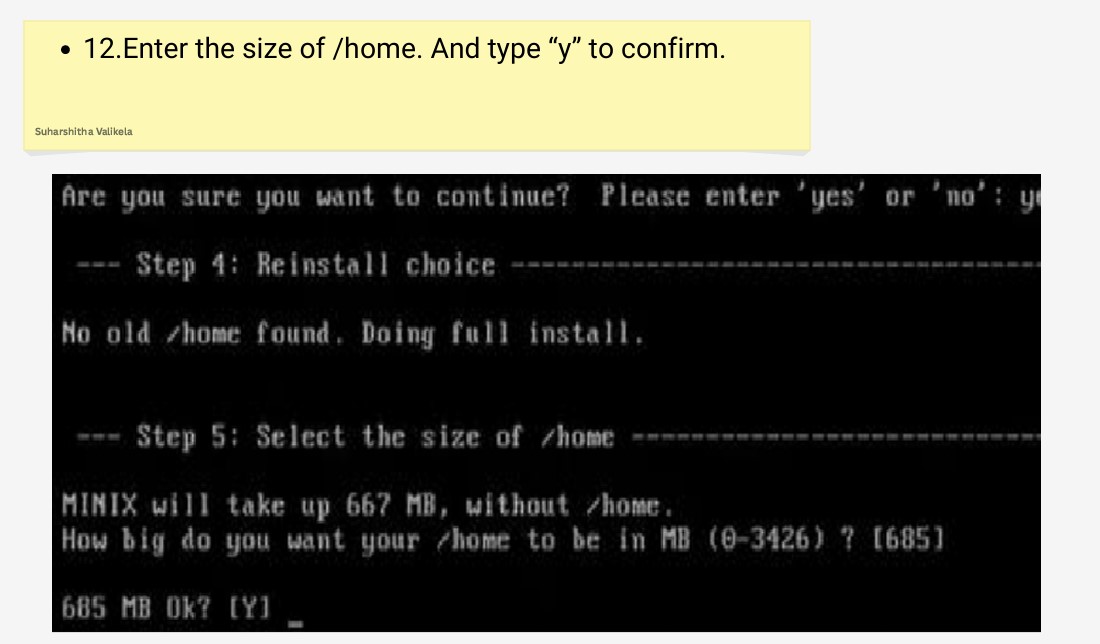


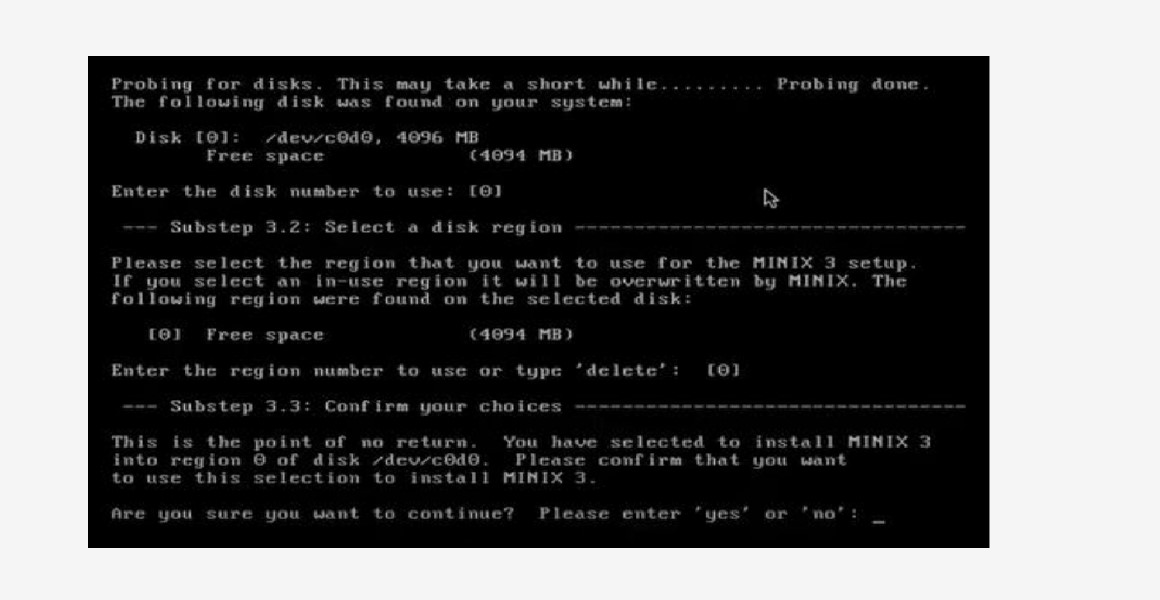


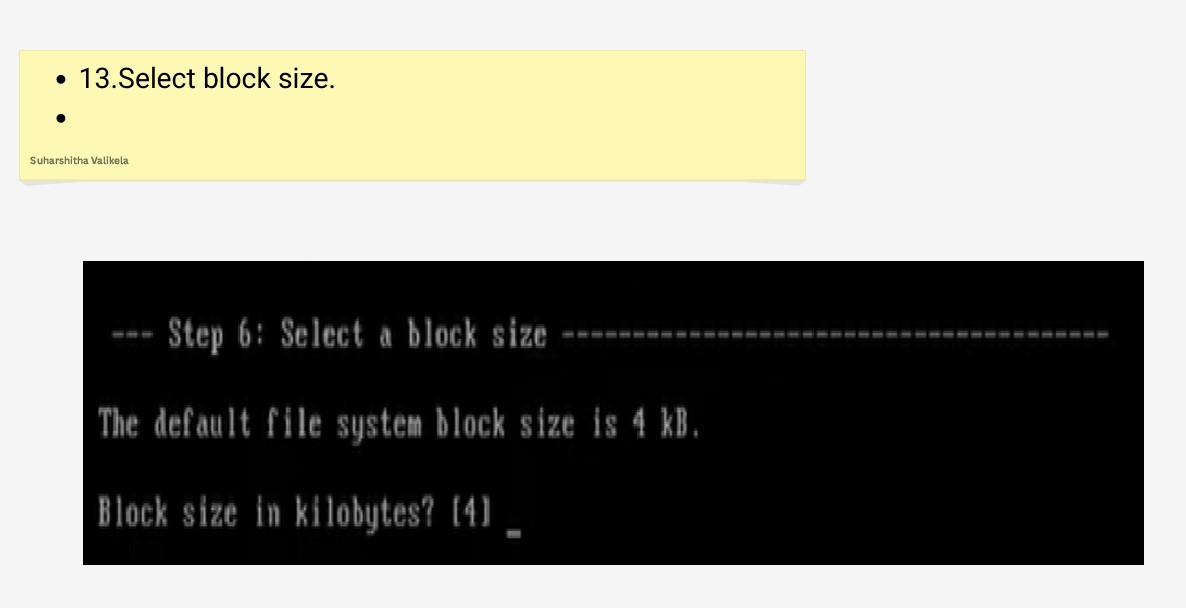


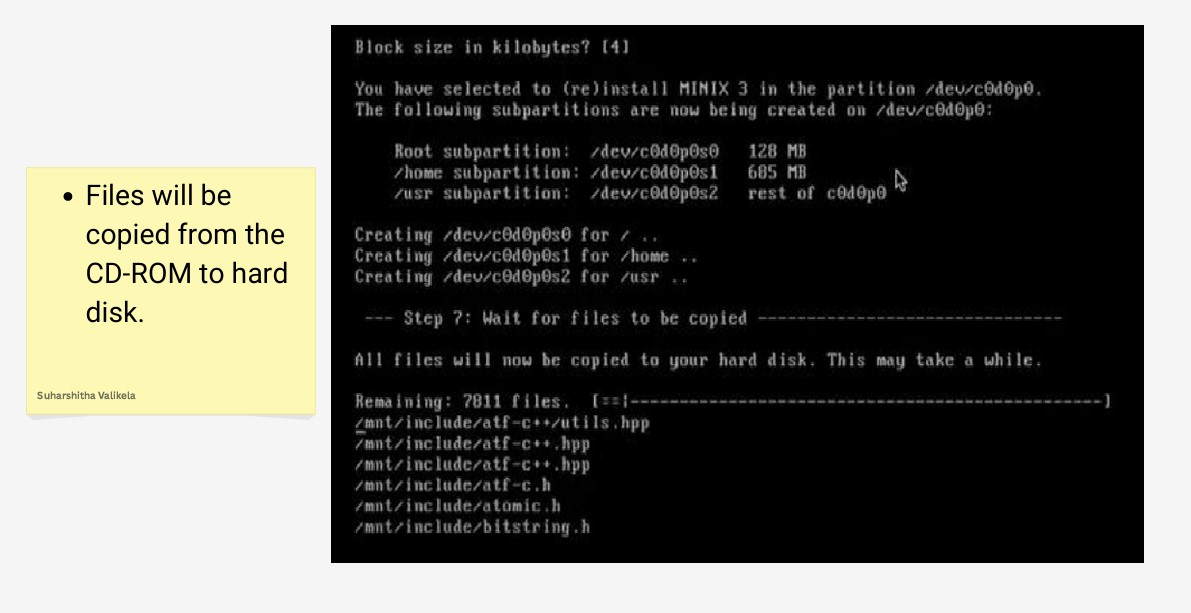


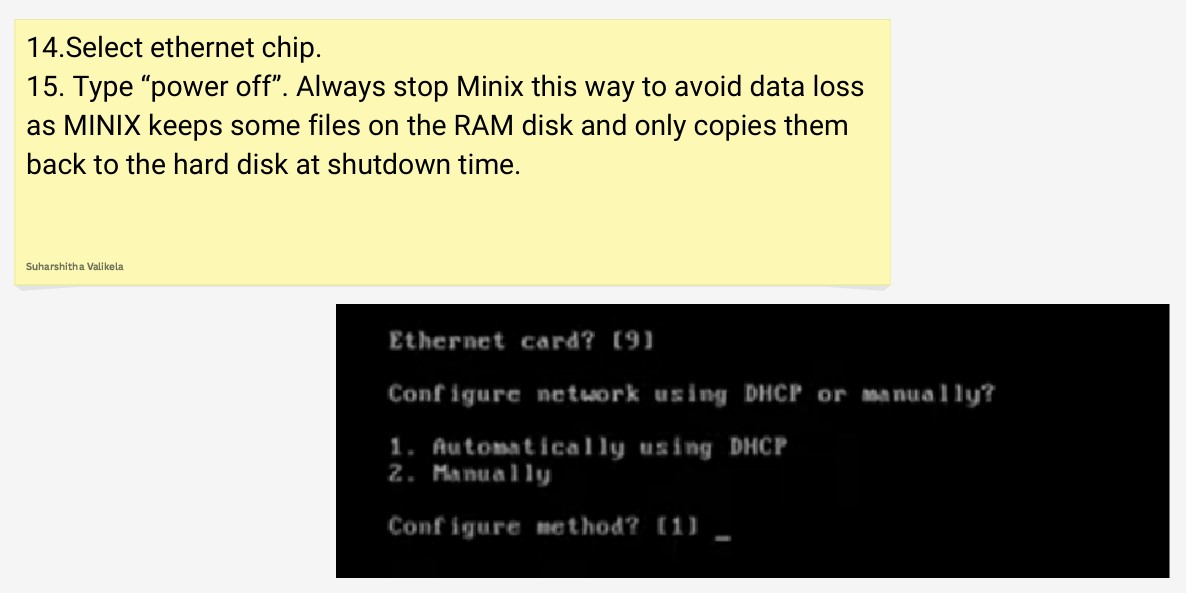












# B) PHYSICAL MACHINE:

1)Download Minix 3 ISO:

2)Visit the official Minix 3 website.

3)Download the latest ISO image.

4)Create a Bootable USB Drive:

5)Use Rufus (Windows), Etcher (Mac/Linux), or the dd command (Linux) to create a bootable USB drive.

6)Select the Minix 3 ISO file and your USB drive in the tool, then start the process.

7)Boot from the USB Drive:

7.A)Insert the USB drive into the physical machine.

7.B)Restart the machine and enter the BIOS/UEFI setup (commonly accessed by pressing F2, F12, Del, or Esc during startup).

7.C)Change the boot order to prioritize the USB drive.

7.D)Save changes and exit the BIOS/UEFI setup.

8)Install Minix 3:

9)Boot from the USB drive to access the Minix 3 installation screen.

10)Follow the on-screen instructions to install Minix 3 (select installation type, partition the disk, confirm settings).

11)Once installation is complete, remove the USB drive and reboot the machine.

12)Post-Installation Configuration:

12.A)After rebooting, complete initial setup tasks such as setting a root password and configuring network settings.

INTERFACE FEATURES IN MINIX

# FEATURES :

MICROKERNEL ARCHITECTURE

RELIABILITY

SECURITY

EDUCATION

PORTABILITY

COMMUNITY SUPPORT

FILE SYSTEMS

NETWORKING

MEMORY MANAGEMENT

PROCESS MANAGEMENT

COMMAND-LINE INTERFACE (CLI):

THE PRIMARY INTERFACE FOR INTERACTING WITH MINIX IS THE COMMAND-LINE INTERFACE. USERS

USERS CAN NAVIGATE THE FILE SYSTEM, MANIPULATE FILES AND DIRECTORIES, MANAGE PROCESSES,

CONFIGURE SYSTEM SETTINGS, AND PERFORM VARIOUS ADMINISTRATIVE TASKS USING THE CLI.

SHELL:

MINIX TYPICALLY USES THE BOURNE AGAIN SHELL (BASH) OR ANOTHER UNIX SHELL AS ITS

DEFAULT COMMAND INTERPRETER.

THE SHELL PROVIDES FEATURES SUCH AS COMMAND-LINE EDITING, COMMAND HISTORY, SHELL

USERS CAN CUSTOMIZE THEIR SHELL ENVIRONMENT BY CONFIGURING ENVIRONMENT VARIABLES,

GRAPHICAL USER INTERFACE (OPTIONAL):

WHILE MINIX PRIMARILY FOCUSES ON THE CLI, GRAPHICAL USER INTERFACES (GUIS) CAN BE INSTALLED ON TOP OF THE CORE SYSTEM FOR USERS WHO PREFER GRAPHICAL INTERACTION.

X11 IS A COMMON WINDOWING SYSTEM USED IN MINIX ENVIRONMENTS TO SUPPORT GUI APPLICATIONS AND DESKTOP ENVIRONMENTS.

VIRTUAL CONSOLES:

MINIX TYPICALLY SUPPORTS MULTIPLE VIRTUAL CONSOLES, ALLOWING USERS TO SWITCH BETWEEN DIFFERENT TEXT-BASED TERMINAL SESSIONS.

EACH VIRTUAL CONSOLE PROVIDES AN INDEPENDENT COMMAND-LINE INTERFACE, ENABLING USERS TO

PERFORM CONCURRENT TASKS OR SWITCH BETWEEN DIFFERENT TASKS SEAMLESSLY.

REMOTE ACCESS:

SUPPORT FOR REMOTE ACCESS PROTOCOLS LIKE SSH AND TELNET, ENABLING USERS TO MANAGE THE SYSTEM REMOTELY OVER A NETWORK.

TEXT-BASED USER INTERFACE (TUI):

TEXT EDITORS: MINIX INCLUDES VARIOUS TEXT EDITORS LIKE VI, NANO, AND EMACS, ALLOWING USERS TO CREATE AND EDIT TEXT FILES DIRECTLY FROM THE COMMAND LINE.

.

SYSTEM MONITORING TOOLS:

UTILITIES SUCH AS TOP, PS, AND VMSTAT PROVIDE INSIGHTS INTO SYSTEM PERFORMANCE, RESOURCE USAGE, AND RUNNING PROCESSES.

FILE MANAGEMENT:

USERS CAN NAVIGATE THE FILE SYSTEM, MANIPULATE FILES AND DIRECTORIES, AND PERFORM FILE-RELATED TASKS USING COMMANDS LIKE LS, CP, MV, AND RM.

PROCESS MANAGEMENT:

MINIX OFFERS COMMANDS AND UTILITIES FOR MANAGING PROCESSES, INCLUDING PROCESS CREATION, TERMINATION, AND MONITORING.

NETWORKING TOOLS:

UTILITIES LIKE IFCONFIG, PING, AND NETSTAT ENABLE USERS TO CONFIGURE NETWORK INTERFACES, TEST NETWORK CONNECTIVITY, AND VIEW NETWORK-RELATE INFORMATION.

FILE AND DIRECTORY MANAGEMENT

• To read and write files, user processes send, messages to the file system

• The file system can be modified, experimented with, and tested completely independently

of the rest of MINIX

Minix File System Functions:

✓ Keeps tracks of disk blocks and free space.

✓ Protects files against unauthorized usage.

✓ Carries out file system calls

MINIX COMMANDS

1)CAL

print a calendar

2) CAT > file.txt

to create file

3) cmp

compare two files

4) chgrp

change group

5) grep

grep and display content

6) clear

clears the terminal screen

7) cut

select out columns of a file

8) dd

convert and copy a file

9) find

searches text in a particular file

10) fgrep

fixed grep

11) login

log in the computer

12) ls

list the contents of the directory

13) mail

send and receive electronic mail

14) mkdir

Make director

15) mkfifo

make a named pipe

16) pwd

prints working directory

17) size

print text, data, and bss size of a program

18) chmod

change access mode for files

19) copy

Copy one or more files to another location

20) date

display or set the time

21) rmdir

removes a directory

22) echo

display message on screen

23) leave

warn when it is time to go home

24) look

find lines in a sorted list

25) hostname

DIsplay hostname

26) pr

print a file

27) sort

sort a file of ASCII lines

28) uname

system info

29) write

send a message to logged-in user

30) create

create a new file

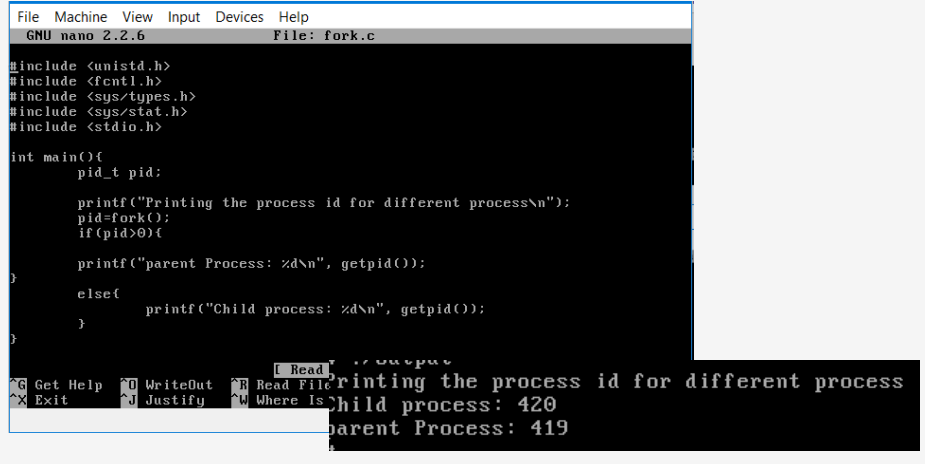
MINIX SYSTEM CALLS

# A) PROCESS MANAGEMENT:

1)pid = getpid() - Return the caller’s process id

2)pid = getpgrp() - Return the id of the caller’s processgroup

3) pid = setsid() - Create a new session and return its process group id



# B)FILE MANAGEMENT:

1) creat(name, mode) - create a new file

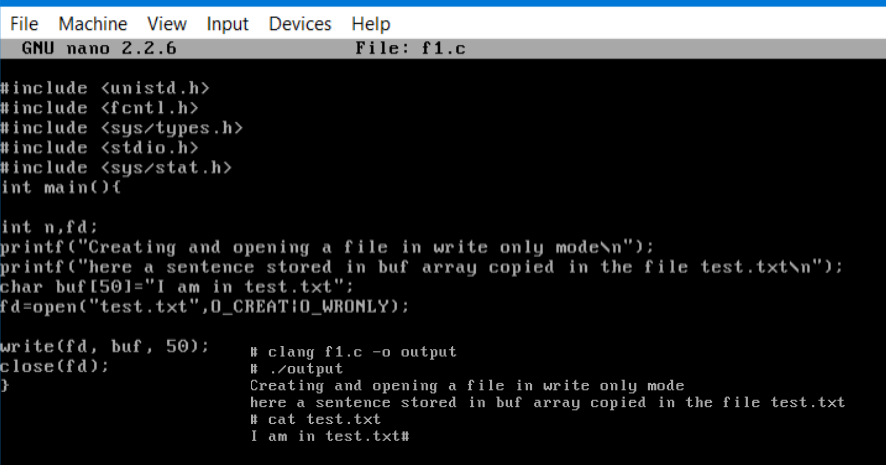
2) mknod(name, mode, addr) - Create a regular, special, or directory i-node

3) close(fd) - Close an open file

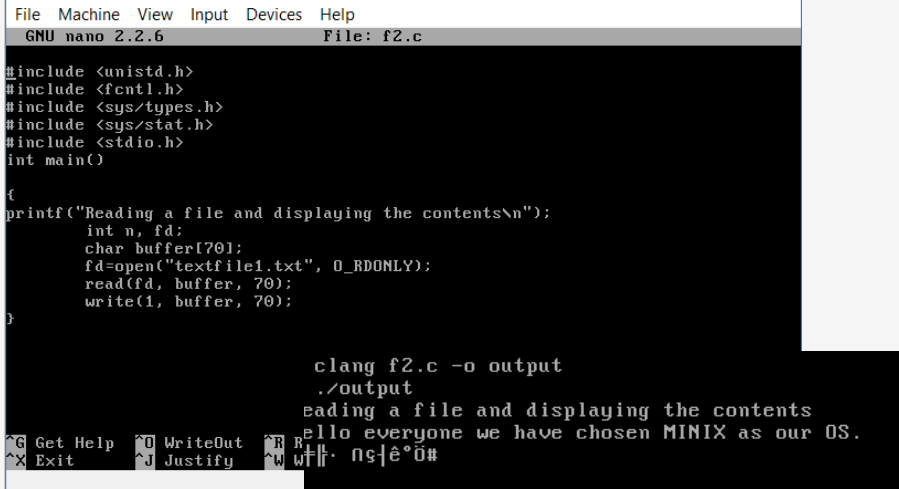
4) open(file, how, ...) - Open a file for reading, writing or both

5) read(fd, buffer, nbytes) - Read data from a file into a buffer

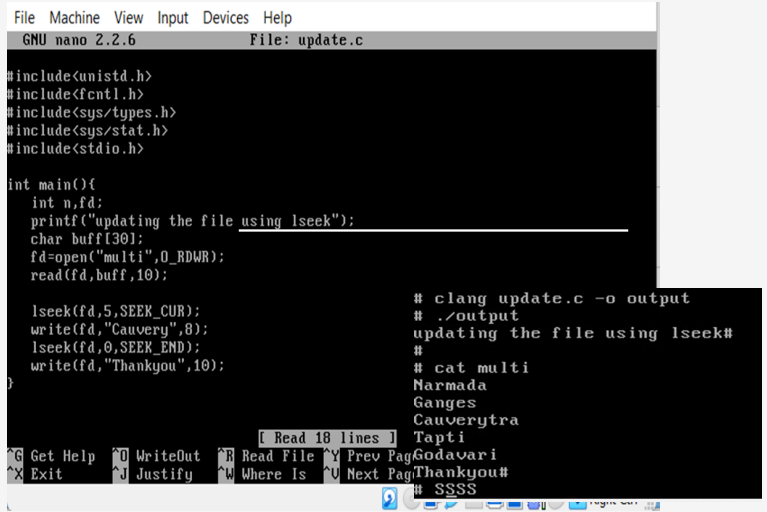
# CREATING AND OPENING A FILE IN WRITE ONLY MODE:



# READING A FILE AND DISPLAYING THE CONTENTS:



# UPDATING THE FILE USING LSEEK:



# C) DIRECTORY & FILE SYSTEM MANAGEMENT:

1) mkdir(name, mode) - Create a new directory

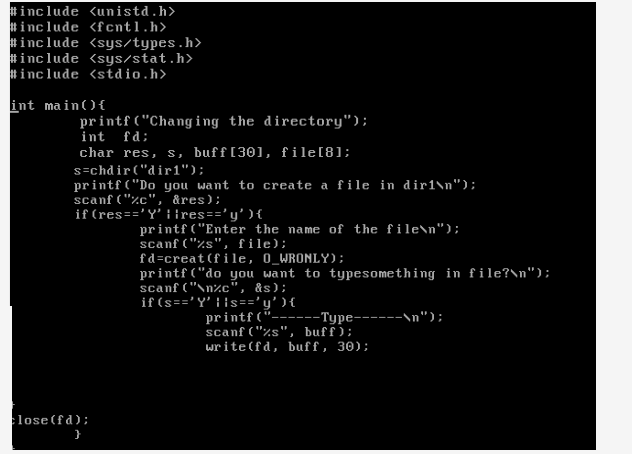
2) rmdir(name) - Remove an empty directory

3) link(name1, name2) - Create a new entry, name2, pointing to name1

4) unlink(name) - Remove a directory entry

5) sync() - Flush all cached blocks to the disk

# CHANGING THE DIRECTORY AND CREATING THE FILE AND ENTERING THE CONTENTS IN A MENU DRIVEN PROGRAM USING SYSTEM CALLS



SCHEDULING ALOGORITHM IN MINIX

Minix3 uses multilevel queuing system for scheduling. Sixteen levels of queues are defined,

• The clock and system tasks are in layer 1 receive the highest priority.

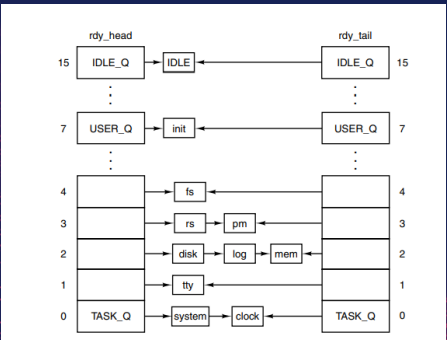
• Device drivers of layer 2 gets the lowest priority, but each device drivers are not considered to be same.

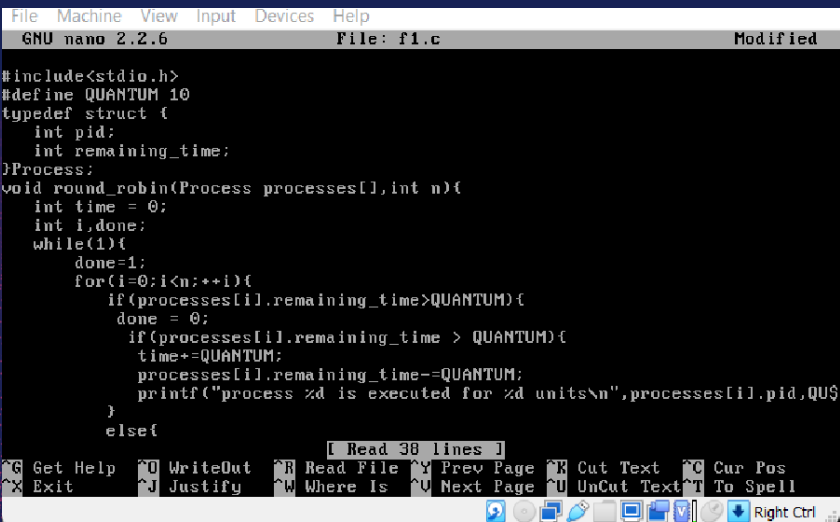
• Server processes (operating process) of layer 3 gets the lowest priority than drivers.

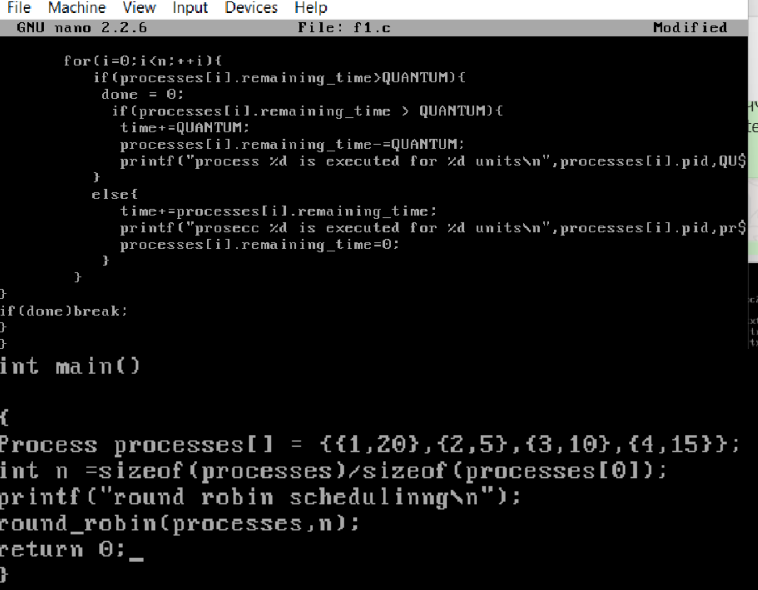
• User processes starts with less priority, a command from the user can raise or lower the priority of these process.

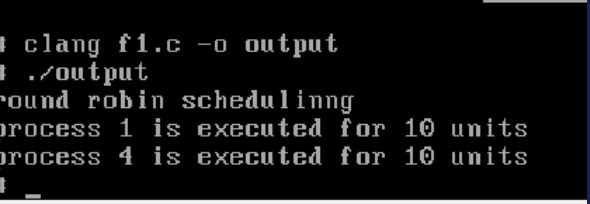
• The lowest priority queue is used only by the idle process Priority of these processes may change during its execution.

Scheduling is round robin in each queue. If a running process uses up its quantum it is moved to the tail of the queue and a new quantum is given.









USABILITY AS SERVER

Microkernel Architecture:

Higher reliability and stability due to essential functions in the microkernel.Enhanced security and robustness through service isolation.

Lightweight:

Efficient operation on older or less powerful hardware. Suitable for small-scale servers or embedded systems.

Security:

Minimized kernel mode code enhances overall system security.User-space processes for drivers and services reduce risk of system-wide crashes.

Educational Value:

Ideal for learning about operating systems and server management.

Simple, well-documented codebase for educational use.

USABILITY AS DESKTOP

Minix, while primarily designed as an educational tool for understanding operating system concepts and the microkernel architecture, has certain features that can be considered when evaluating its usability as a desktop operating system. Here’s a detailed look at the usability of Minix as a desktop OS

Educational Value:

Excellent platform for learning about operating systems and microkernel architecture.Simple and well-documented codebase for educational purposes.

Security:

Microkernel architecture enhances security by isolating services in user space.Reduced risk of system-wide crashes and vulnerabilities.

Lightweight:

Efficient performance on older or less powerful hardware.

Minimal resource requirements make it suitable for basic desktop tasks.

COMPARISION WITH LINUX

Modularity:

Both systems embrace modularity through their architectures.

The microkernel design of Minix separates essential kernel services into distinct modules for a lean core and system extensibility.

Linux supports dynamic loading and unloading of modules, allowing for runtime extension and customization without rebooting.

Flexibility:

Minix supports networking protocols, multiuser environments, real-time capabilities, and fault tolerance.

These features enable Minix to handle various computing scenarios from networked environments to real-time systems.

Linux similarly offers extensive support for diverse computing needs, making both OSes versatile for educational, research, and production purposes.

Customizability:

Minix offers extensive customization options for security policies and fault tolerance strategies.

Administrators can fine-tune authentication mechanisms, define access control rules, and tailor fault tolerance parameters.

This high degree of customizability allows both Minix and Linux to meet specific organizational requirements and compliance standards, optimizing security, performance, and resource utilization.

Adaptability:

Both systems dynamically respond to changing conditions, user requirements, and system events.

Features like adjusting access control policies and reallocating resources ensure seamless operation in dynamic environments.

Scalability:

Designed to scale seamlessly with growing system demands.

Support for varying workloads, user populations, and operational complexities without compromising security or reliability.

Interoperability:

Designed to interoperate with other system components, third-party applications, and external services.

Adherence to industry standards ensures compatibility and promotes efficiency, flexibility, and innovation in heterogeneous environments.

Continuous Monitoring:

Real-time detection and response to security threats, system anomalies, and performance issues.

Enhances situational awareness and allows for timely corrective actions and risk mitigation.

Resource Efficiency:

Optimized security features and fault tolerance mechanisms to minimize overhead.

Efficient protocols and strategies conserve computational resources, crucial for embedded and resource-constrained environments.

Documentation and Support:

Comprehensive guidance, troubleshooting resources, and community support.

Ensures users have the necessary information to effectively deploy and maintain security and fault tolerance features.

Continuous Improvement:

Regular updates, patches, and community contributions for security and fault tolerance.

Active feedback incorporation ensures adaptation to new security threats and technological advancements.

Integration with Development Tools:

Seamless integration with development tools, debuggers, and performance analysis utilities.

Facilitates system development, testing, and optimization, improving system quality and robustness.

Compliance and Certification:

Designed to meet industry-specific compliance requirements and certification standards.

Ensures regulatory compliance, data privacy, and system reliability, instilling confidence and opening new business opportunities.

Dissimilarities between MINIX and LINUX

# COMPARISION CHART

|  |  |  |
| --- | --- | --- |
| **Aspect** | **MINIX** | **Linux** |
| **Design Philosophy** | Simplicity, reliability, educational value | Performance, flexibility, widespread adoption |
| **Kernel Architecture** | Microkernel | Monolithic |
| **System Requirements** | Efficient on resource-constrained systems | Optimized for a wide range of hardware platforms |
| **Development Community** | Smaller, academically focused | Large, diverse, rapid innovation |
| **Customization and Flexibility** | Modular, easy customization | Extensive customization options, vast ecosystem |
| **File System Support** | Native (MFS, MFS2), compatible (ext2, FAT) | Wide range (ext4, Btrfs, XFS, etc.) |
| **User Interface** | Command-line focused | Variety of graphical desktop environments |
| **Target Audience** | Educational institutions, researchers | Hobbyists, businesses, organizations |
| **Support and Documentation** | Extensive educational resources, forums | Wealth of documentation, forums, professional support |
| **Development Model** | Centralized, stability-focuseD | Distributed, rapid innovation |
| **License** | Permissive license | GNU General Public License (GPL) |

ADVANCED FEATURES USED IN MINIX OS

Supporting Threads:

MINIX offers a POSIX-compliant thread library for thread management and synchronization.

Lightweight Processes (LWPs) enable parallel execution with efficient context switching.

Supporting CPU Scheduling:

MINIX implements a Multilevel Feedback Queue (MLFQ) for dynamic priority adjustment.

Preemptive scheduling maintains system responsiveness by allowing higher-priority tasks to interrupt lower-priority ones.

Memory Management:

MINIX includes advanced memory management techniques such as virtual memory and memory protection mechanisms.

Virtual memory allows efficient memory utilization by providing each process with its own virtual address space.

Device Drivers:

MINIX includes a wide range of device drivers for hardware components such as network interfaces, storage devices, and input/output peripherals.

Advanced features may include support for hot-swapping, power management, and plug-and-play functionality for seamless integration of new devices

Security Features:

MINIX emphasizes security with features like role-based access control (RBAC), mandatory access control (MAC), and stack protection mechanisms.

RBAC and MAC mechanisms restrict access based on user roles and predefined policies, enhancing system security.

Distributed System Support:

MINIX may include support for distributed computing environments, allowing multiple instances of the OS to collaborate and share resources across a network.

Distributed file systems, distributed process management, and remote procedure call (RPC) mechanisms enable seamless interaction between distributed components.

Debugging and Profiling Tools:

MINIX provides developers with robust debugging and profiling tools for analyzing system behavior, identifying performance bottlenecks, and diagnosing software defects.

Tools like tracers, profilers, and debuggers aid in the development and optimization of software running on the MINIX platform.

SUPPORT FOR SYNCHRONIZATION

# SUPPORT FOR THREAD SYNCHRONIZATION:

Thread synchronization is defined as mechanism which ensures that two or more concurrent threads don’t simultaneously execute in critical section. When one thread stars executing the critical section the other thread should wait until the first thread finishes.

# MUTEX LOCK FOR MINIX OS (THREAD SYNCHRONIZATION)

Mutex is a lock that will set before using a shared resource and releases after using it . when the lock is set no other thread access the locked region of code.

pthread\_mutex\_init()- Initialize the Mutex

pthread\_mutex\_lock()-When a thread needs to enter a critical section, it locks the mutex.

pthread\_mutex\_unlock()-After the thread has finished executing the critical section, it unlocks the mutex.

pthread\_mutex\_destroy()-When the mutex is no longer needed, it should be destroyed.

# SUPPORT FOR PROCESS SYNCRONIZATION:

Process synchronization in MINIX OS, much like in other operating systems, is crucial for managing concurrent processes that share resources. MINIX OS provides several mechanisms for process synchronization, ensuring that processes do not interfere with each other while accessing shared resources.

PRODUCER CONSUMER USING SEMAPHORE

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/sem.h>

#define BUFFER\_SIZE 5

int sem\_id;

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

} sem\_ctl;

void wait\_semaphore(int sem\_id, int sem\_num) {

struct sembuf sem\_op;

sem\_op.sem\_num = sem\_num;

sem\_op.sem\_op = -1;

sem\_op.sem\_flg = 0;

semop(sem\_id, &sem\_op, 1);

}

void signal\_semaphore(int sem\_id, int sem\_num) {

struct sembuf sem\_op;

sem\_op.sem\_num = sem\_num;

sem\_op.sem\_op = 1;

sem\_op.sem\_flg = 0;

semop(sem\_id, &sem\_op, 1);

}

void producer() {

int buffer[BUFFER\_SIZE];

int in = 0;

for (int i = 0; i < 10; i++) {

wait\_semaphore(sem\_id, 1); // Wait for space in the buffer

buffer[in] = i;

printf("Produced: %d\n", i);

in = (in + 1) % BUFFER\_SIZE;

signal\_semaphore(sem\_id, 0); // Signal that there's data in the buffer

sleep(1); // Simulate some processing time

}

}

void consumer() {

int buffer[BUFFER\_SIZE];

int out = 0;

for (int i = 0; i < 10; i++) {

wait\_semaphore(sem\_id, 0); // Wait for data in the buffer

int item = buffer[out];

printf("Consumed: %d\n", item);

out = (out + 1) % BUFFER\_SIZE;

signal\_semaphore(sem\_id, 1); // Signal that there's space in the buffer

sleep(1); // Simulate some processing time

}

}

int main() {

sem\_id = semget(IPC\_PRIVATE, 2, IPC\_CREAT | 0666); // Create two semaphores

// Initialize the semaphores

sem\_ctl.val = BUFFER\_SIZE;

semctl(sem\_id, 0, SETVAL, sem\_ctl);

sem\_ctl.val = 0;

semctl(sem\_id, 1, SETVAL, sem\_ctl);

// Fork producer and consumer processes

pid\_t pid = fork();

if (pid == 0) {

producer();

} else if (pid > 0) {

consumer();

wait(NULL); // Wait for the child process (producer) to finish

} else {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

// Cleanup

semctl(sem\_id, 0, IPC\_RMID, 0);

semctl(sem\_id, 1, IPC\_RMID, 0);

return 0;

}

THREADS:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define MAX\_SIZE 100

int arr[MAX\_SIZE];

int n;

// Mutex to ensure safe access to shared data

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

// Thread function to calculate the average

void\* th0(void\* arg) {

int sum = 0;

float average;

printf("Enter the number of elements: ");

scanf("%d", &n);

printf("Enter the elements:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &arr[i]);

}

for (int i = 0; i < n; i++) {

sum += arr[i];

}

average = (float)sum / n;

printf("The AVERAGE value is %.2f\n", average);

pthread\_exit(NULL);

}

// Thread function to find the minimum value

void\* th1(void\* arg) {

int temp;

pthread\_mutex\_lock(&mutex); // Lock mutex before accessing shared data

if (n <= 0) {

pthread\_mutex\_unlock(&mutex); // Release mutex before returning

pthread\_exit(NULL);

}

temp = arr[0];

for (int i = 1; i < n; i++) {

if (temp > arr[i]) {

temp = arr[i];

}

}

printf("The MINIMUM value is %d\n", temp);

pthread\_mutex\_unlock(&mutex); // Release mutex after accessing shared data

pthread\_exit(NULL);

}

// Thread function to find the maximum value

void\* th2(void\* arg) {

int temp;

pthread\_mutex\_lock(&mutex); // Lock mutex before accessing shared data

if (n <= 0) {

pthread\_mutex\_unlock(&mutex); // Release mutex before returning

pthread\_exit(NULL);

}

temp = arr[0];

for (int i = 1; i < n; i++) {

if (temp < arr[i]) {

temp = arr[i];

}

}

printf("The MAXIMUM value is %d\n", temp);

pthread\_mutex\_unlock(&mutex); // Release mutex after accessing shared data

pthread\_exit(NULL);

}

int main() {

pthread\_t tid[3];

// Create threads

pthread\_create(&tid[0], NULL, th0, NULL);

pthread\_join(tid[0], NULL);

pthread\_create(&tid[1], NULL, th1, NULL);

pthread\_join(tid[1], NULL);

pthread\_create(&tid[2], NULL, th2, NULL);

pthread\_join(tid[2], NULL);

pthread\_mutex\_destroy(&mutex); // Destroy mutex

return 0;

}

MUTEX FOR MULTI-THREAD PROCESS:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

int MAX=10;

int count = 1;

pthread\_mutex\_t thr = pth PTHREAD\_MUTEX\_INITIALISER;

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALISER;

Void \*even ()

{

while (Count < MAX)

{

pthread\_mutex\_lock(&thr);

while(count%2!=0)

{

pthread\_cond\_wait(&cond, &thr);

}

printf("%d \n", count++);

pthread\_mutex\_unlock (&thr);

pthread\_cond\_signal (&cond);

}

pthread\_exit(0);

}

void \*odd()

{

while (Count <MAX)

{

pthread\_mutex\_lock(&thr);

while (count %2!= 1)

{

pthread\_cond\_wait(&cond,&thr);

}

printf ("%d \n", count++);

pthread\_mutex\_unlock(&thr);

pthread\_cond\_signal (&cond);

}

pthread\_exit(0);

}

int main()

{

pthread\_t thread1, thread2;

pthread\_mutex\_init(&thr, NULL);

pthread\_cond\_init(&cond, NULL);

pthread\_create(&thread1, NULL, &even, NULL);

pthread\_create (&thread2, NULL, &odd, NULL);

pthread\_join (thread1, NULL);

pthread\_join (thread2, NULL);

pthread\_mutex\_destroy(&thr);

pthread\_cond\_destroy (&cond);

return 0;

}

MEMORY MANAGEMENTAND ALLOCATION IN MINIX

Memory management is tied to process management since processes are allocated memory and use memory.

For pre-3.1 minix, there is no separate memory manager, and we assume that a process is a collection of (stack, text, heap) segments.

Each segment is allocated contiguously in RAM and memory is treated as a collection of holes and allocated segments.

Pre-3.1 Minix does not support paging, virtual memory, and demand paging.

Post-3.2 Minix supports virtualization, paging and demand paging. Moreover it has a new VM (virtual memory) server.

Post-3.2 Minix essentially uses segmented paging with virtual memory

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#include <stdbool.h>

#define MAX 25

void allocate(int blocksize[], int m, int processsize[], int n, char \*method) {

int allocation[n], i, j, idx;

for (i = 0; i < n; i++)

allocation[i] = -1;

for (i = 0; i < n; i++) {

idx = -1;

for (j = 0; j < m; j++) {

if (blocksize[j] >= processsize[i]) {

if (strcmp(method, "FF") == 0) {

idx = j;

break;

} else if (strcmp(method, "BF") == 0 && (idx == -1 || blocksize[j] < blocksize[idx])) {

idx = j;

} else if (strcmp(method, "WF") == 0 && (idx == -1 || blocksize[j] > blocksize[idx])) {

idx = j;

}

}

}

if (idx != -1) {

allocation[i] = idx;

blocksize[idx] -= processsize[i];

}

}

printf("\n%s Fit allocation:\n", strcmp(method, "FF") == 0 ? "First" : strcmp(method, "BF") == 0 ? "Best" : "Worst");

for (i = 0; i < n; i++) {

if (allocation[i] != -1) {

printf("Process %d allocated to block %d\n", i + 1, allocation[i] + 1);

} else {

printf("Process %d not allocated\n", i + 1);

}

}

}

int main() {

int blocksize[MAX], processsize[MAX], blocksizecopy[MAX], m, n, i;

printf("Enter the number of blocks: ");

scanf("%d", &m);

printf("Enter the size of each block:\n");

for (i = 0; i < m; i++) {

printf("Block %d: ", i + 1);

scanf("%d", &blocksize[i]);

}

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the size of each process:\n");

for (i = 0; i < n; i++) {

printf("Process %d: ", i + 1);

scanf("%d", &processsize[i]);

}

for (i = 0; i < m; i++)

blocksizecopy[i] = blocksize[i];

allocate(blocksize, m, processsize, n, "FF");

for (i = 0; i < m; i++)

blocksize[i] = blocksizecopy[i];

allocate(blocksize, m, processsize, n, "BF");

for (i = 0; i < m; i++)

blocksize[i] = blocksizecopy[i];

allocate(blocksize, m, processsize, n, "WF");

return 0;

}

Reason for MINIX OS to use the SCAN or C-SCAN disk scheduling algorithm is its efficiency in reducing disk arm movement and minimizing seek time. Disk I/O operations can be a significant bottleneck in system performance, and minimizing seek time can lead to faster response times and improved overall system throughput. By prioritizing disk requests based on their proximity to the current disk head position, SCAN and C-SCAN algorithms optimize the sequence in which disk operations are performed, resulting in efficient disk utilization and reduced latency.

CRITICAL SECTION

In MINIX OS, managing critical sections is crucial to ensure that concurrent processes or threads do not lead to race conditions, data corruption, or inconsistencies. Critical sections are portions of the code that must be executed by only one thread or process at a time to prevent concurrent access to shared resources. Below are the critical section rules and solutions implemented in MINIX OS:

Critical Section Rules

Mutual Exclusion: Only one process or thread should be allowed to execute in the critical section at any given time.

Progress: If no process is in the critical section, and some processes wish to enter it, one of those processes must be allowed to enter the critical section.

Bounded Waiting: There should be a limit on the number of times other processes are allowed to enter the critical section after a process has made a request to enter it and before that request is granted.

No Assumptions about Speeds or Number of CPUs: The solution should work regardless of the number of CPUs or the relative speeds of processes.

Solutions for critical section in MINIX

Semaphores: Semaphores are used to signal and control access to resources. MINIX uses both binary and counting semaphores.

Mutexes: Mutexes are used to enforce mutual exclusion by locking critical sections so that only one thread can execute them at a time.

Condition Variables: Used with mutexes to allow threads to wait for certain conditions to be met before proceeding.

\*\*\*\*\*