IT5403

OPERATING SYSTEMS

A DEMO REPORT

Submitted by Team Dynamic:

Gowtham Rajasekaran	2022506084
Aadhira R	2022506087
Ragul P	2022506054
Vicky Yashwa Anantharaj	2022506049
Arvinth	2022506316

B.Tech(4/8)

DEPARTMENT OF INFORMATION TECHNOLOGY

MADRAS INSTITUTE OFTECHNOLOGY

ANNA UNIVERSITY- CHENNAI

CHENNAI - 600044

May / June 2024

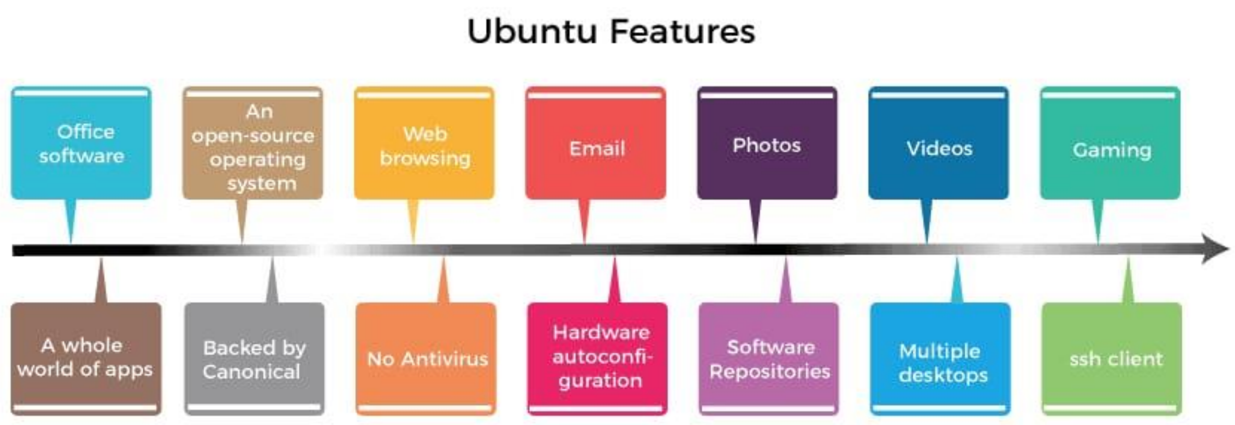
INDEX

INTRODUCTION

What is Ubuntu?

Ubuntu is a popular Linux distribution that is widely used for desktop, server, and cloud computing. It is known for its ease of use, regular updates, and strong community support. Developed by Canonical Ltd., Ubuntu is based on the Ubuntu architecture and is released with a new version every six months, with long-term support (LTS) versions released every two years. Ubuntu comes with a variety of pre-installed software and offers a user-friendly interface, making it accessible to both beginners and experienced users alike.

Key Features



1. **User-Friendly Interface**: Ubuntu offers a sleek and intuitive desktop environment, such as GNOME or Unity, making it easy for users to navigate and operate their systems.

2. **Open Source**: Ubuntu is built on open-source principles, meaning its source code is freely available for anyone to use, modify, and distribute.

3. **Regular Updates**: Canonical releases new versions of Ubuntu every six months, ensuring users have access to the latest features, security patches, and software updates.

4. **Long-Term Support (LTS)**: LTS releases are supported for five years on the desktop and server, providing stability and reliability for users who prefer not to upgrade frequently.

5. **Software Center**: Ubuntu Software Center provides a centralized location for users to discover, install, and manage applications, making it easy to find and install software.

6. **Community Support**: Ubuntu has a large and active community of users and developers who provide support, troubleshooting, and guidance through forums, wikis, and other online resources.

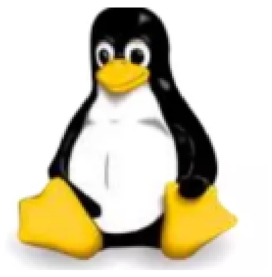
7. **Security**: Ubuntu prioritizes security and includes built-in security features such as AppArmor, which helps protect applications from security threats.

8. **Versatility**: Ubuntu is versatile and can be used for various purposes, including desktop computing, server hosting, cloud deployments, and IoT (Internet of Things) devices.

9. **Accessibility**: Ubuntu aims to be accessible to users with disabilities by providing features such as screen readers, magnification tools, and keyboard navigation options.

10. **Customization**: Users can customize Ubuntu to suit their preferences by choosing from a variety of desktop environments, themes, and extensions available in the software repositories.

Ubuntu
Debian
Linux



IPC- Shared memory

Shared memory is a mechanism provided by operating systems that allows multiple processes to access the same region of memory. It enables efficient communication and data sharing between processes running on the same system.

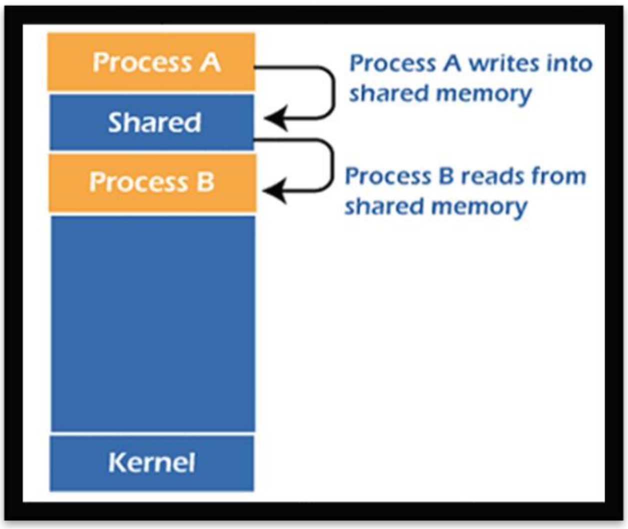
How Shared Memory Works?

1. **Memory Allocation**: Initially, a region of memory is allocated by one process using system calls like **shmget** or **shm_open**. This region is identified by a unique key or name.
2. **Attachment**: Processes that need to access this shared memory segment attach it to their address space using the **shmat** system call. This allows them to read from and write to the shared memory.
3. **Communication**: Processes can now communicate with each other by reading from and writing to the shared memory segment. They can use this shared memory region to exchange data in a fast and efficient manner.
4. **Synchronization**: Since multiple processes may be accessing the shared memory simultaneously, proper synchronization mechanisms like semaphores or mutexes are often used to ensure data consistency and avoid race conditions.
5. **Detachment**: When a process no longer needs to access the shared memory segment, it detaches from it using the **shmdt** system call. This removes the shared memory from the process's address space.
6. **Cleanup**: If the shared memory segment is no longer needed by any process, it can be removed from the system using **shmctl** or **shm_unlink**. This releases the allocated memory and other associated resources.

Shared memory provides a fast and efficient means of inter-process communication, especially when large amounts of data need to be exchanged between processes. However, it requires careful synchronization to avoid data corruption and ensure proper functioning of the cooperating processes.

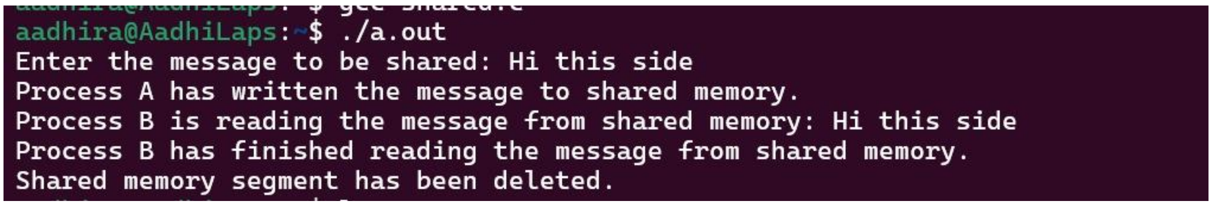
ALGORITHM

```
1. Create a shared memory segment using ftok() and shmget().
2. Attach to the shared memory segment using shmat().
3. Prompt the user to enter a message.
4. Write the entered message to the shared memory.
5. Pause execution for a specified time.
6. Fork a new process.
7. In the child process:
        a. Pause execution for a specified time.
        b. Attach to the shared memory segment using shmat().
        c. Read the message from the shared memory.
        d. Detach from the shared memory segment using shmdt().
8. In the parent process:
        a. Wait for the child process to finish.
        b. Pause execution for a specified time.
        c. Delete the shared memory segment using shmctl().
9. Exit the program.
```


Process A
Shared
Process B
Kernel
Process A writes into shared memory
Process B reads from shared memory

Representation of shared memory

Demo



```
aadhira@AadhiLaps:~$ ./a.out
Enter the message to be shared: Hi this side
Process A has written the message to shared memory.
Process B is reading the message from shared memory: Hi this side
Process B has finished reading the message from shared memory.
Shared memory segment has been deleted.
```

Scheduling Algorithm – CFS

The Completely Fair Scheduler (CFS) is a process scheduler introduced in the Linux kernel version 2.6.23. It was designed to address shortcomings of previous schedulers like the O(1) scheduler and aims to provide fairness, low latency, and scalability in CPU resource allocation.

How CFS Works?

1. **Virtual Runtime:** CFS maintains a notion of "virtual runtime" for each runnable process. The virtual runtime represents the amount of time a process has executed relative to its scheduling period. Processes with smaller virtual runtimes are given priority.
2. **Red-Black Tree:** CFS uses a red-black tree data structure to organize runnable processes based on their virtual runtimes. The tree is balanced to ensure efficient lookup and insertion of processes.
3. **Scheduling Decision:** When it's time to select the next process to run, CFS picks the leftmost node (i.e., the process with the smallest virtual runtime) from the red-black tree. This ensures that processes with the least amount of CPU time are given preference.
4. **Scheduling Period:** CFS operates on a concept of scheduling periods, typically measured in nanoseconds. During each scheduling period, processes accumulate virtual runtime based on their execution on the CPU.
5. **Fairness Guarantee:** CFS strives to provide fair CPU time allocation among processes. It ensures that each process gets its fair share of CPU time over a certain period, regardless of the number of processes or their CPU utilization patterns.
6. **Dynamic Priority Adjustment:** CFS adjusts the priority of processes dynamically based on their CPU usage history and the amount of time they have spent waiting in the run queue. This helps prevent starvation and ensures that no process hogs the CPU for an extended period.

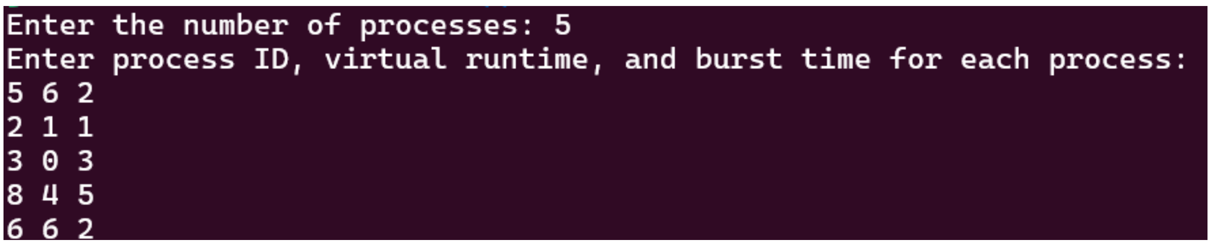
Overall, the Completely Fair Scheduler aims to provide fairness in CPU time allocation while maximizing system throughput and responsiveness. It achieves this by maintaining a balanced red-black tree of processes and making scheduling decisions based on each process's virtual runtime.
Time Complexity:

1. Search – O(log n)
2. Insert – O(log n)
3. Delete – O(log n)

ALGORITHM

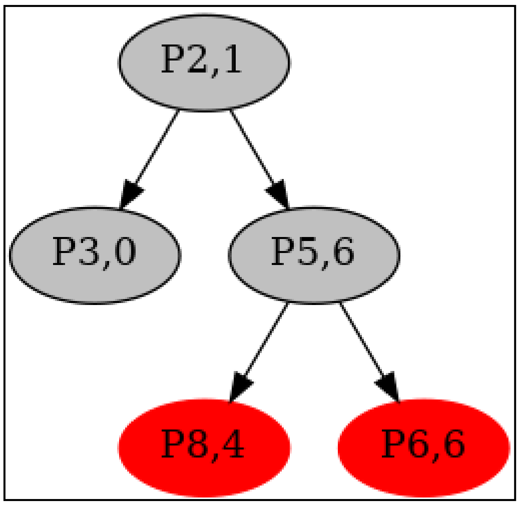
1. Step: Define necessary structures and enums:
    a. Define enum Color with values RED and BLACK.
    b. Define struct Node with process_id, vruntime, color, parent, left, and right pointers.
2. Step: Implement the RedBlackTree class:
    a. Define private member variables, including the root pointer.
    b. Implement rotateLeft() method to perform a left rotation in the tree.
    c. Implement rotateRight() method to perform a right rotation in the tree.
    d. Implement fixViolation() method to fix any violations of the red-black tree properties after insertion.
    e. Implement transplant() method to replace one subtree as a child of its parent with another subtree.
    f. Implement deleteFix() method to fix any violations of the red-black tree properties after deletion.
    g. Implement public methods insert(), minimum(), deleteNode(), getRoot(), inorderHelper(), and inorder().
3. Step: Implement the main function:
    a. Create an instance of RedBlackTree, a queue for process IDs, and a map for burst times.
    b. Prompt the user to enter the number of processes.
    c. Prompt the user to enter process ID, virtual runtime, and burst time for each process, inserting them into the Red-Black Tree.
    d. Print the inorder traversal of the Red-Black Tree.
    e. Dequeue leftmost nodes from the Red-Black Tree and enqueue their process IDs.
    f. Print the order of execution with process IDs, start times, and burst times.
4. Step: Exit the program.

Demo

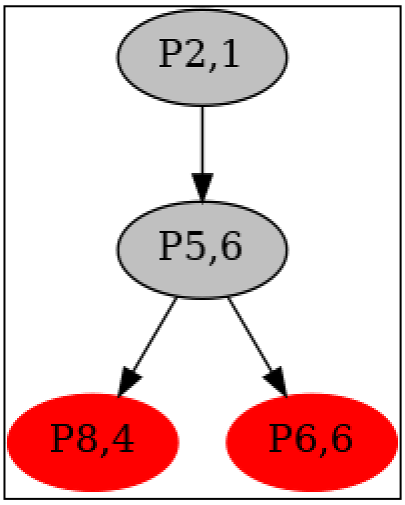


```
Enter the number of processes: 5
Enter process ID, virtual runtime, and burst time for each process:
5 6 2
2 1 1
3 0 3
8 4 5
6 6 2
```

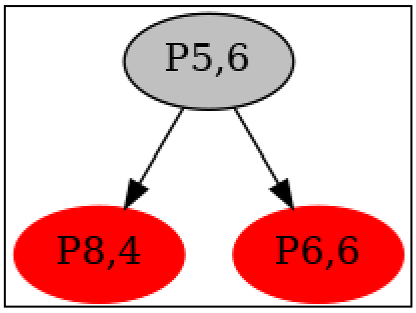
Initial tree



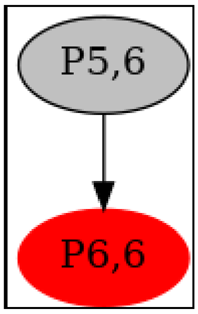
Process P3 scheduled to CPU



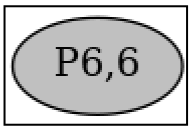
Process P2 scheduled to CPU



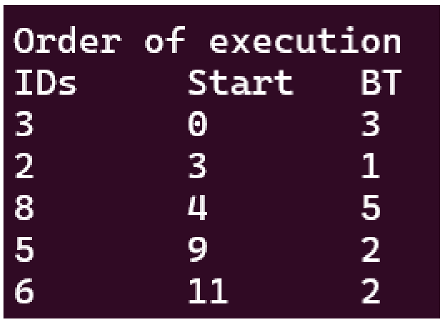
Process P8 scheduled to CPU



Process P5 scheduled to CPU



Output



```
Order of execution
IDs       Start    BT
3         0        3
2         3        1
8         4        5
5         9        2
6         11       2
```

FILE ALLOCATION AND PROTECTION

The ext4 (fourth extended filesystem) is a widely used file system in Linux, known for its robustness, scalability, and performance. Here's an overview of how it allocates space on the disk:

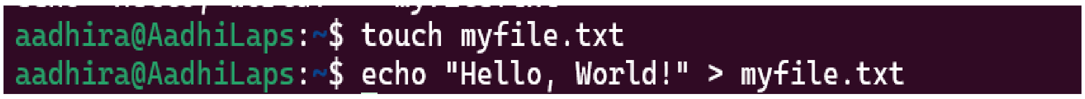
Space Allocation in ext4

1. **Inodes and Data Blocks**: ext4 uses inodes to store metadata about files (e.g., permissions, ownership, timestamps) and data blocks to store the actual file data.
2. **Block Groups**: The disk is divided into block groups, which contain both inodes and data blocks. This division helps in reducing fragmentation and improving performance.
3. **Extent-based Storage**: Unlike the older block mapping method, ext4 uses extents (a range of contiguous blocks) to map large files, reducing fragmentation and improving efficiency.
4. **Delayed Allocation**: ext4 delays the allocation of blocks until data is actually written to disk, which helps in optimizing the layout of the data on disk.
5. **Multi-block Allocator**: This allocator enhances the filesystem's ability to allocate multiple blocks at once, improving performance.

File Manipulation in ext4

Creating, Deleting, and Manipulating Files

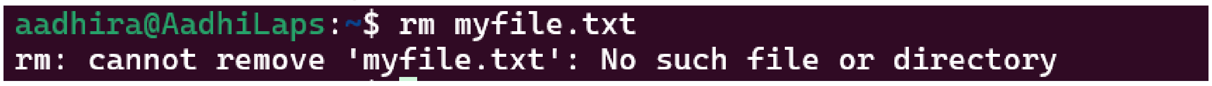
To create, delete, and manipulate files on an ext4 file system in Ubuntu, you can use standard Unix commands.



```
aadhira@AadhiLaps:~$ touch myfile.txt
aadhira@AadhiLaps:~$ echo "Hello, World!" > myfile.txt
```

Deleting Files

You can delete files using the **rm** command.



```
aadhira@AadhiLaps:~$ rm myfile.txt
rm: cannot remove 'myfile.txt': No such file or directory
```

Manipulating Files

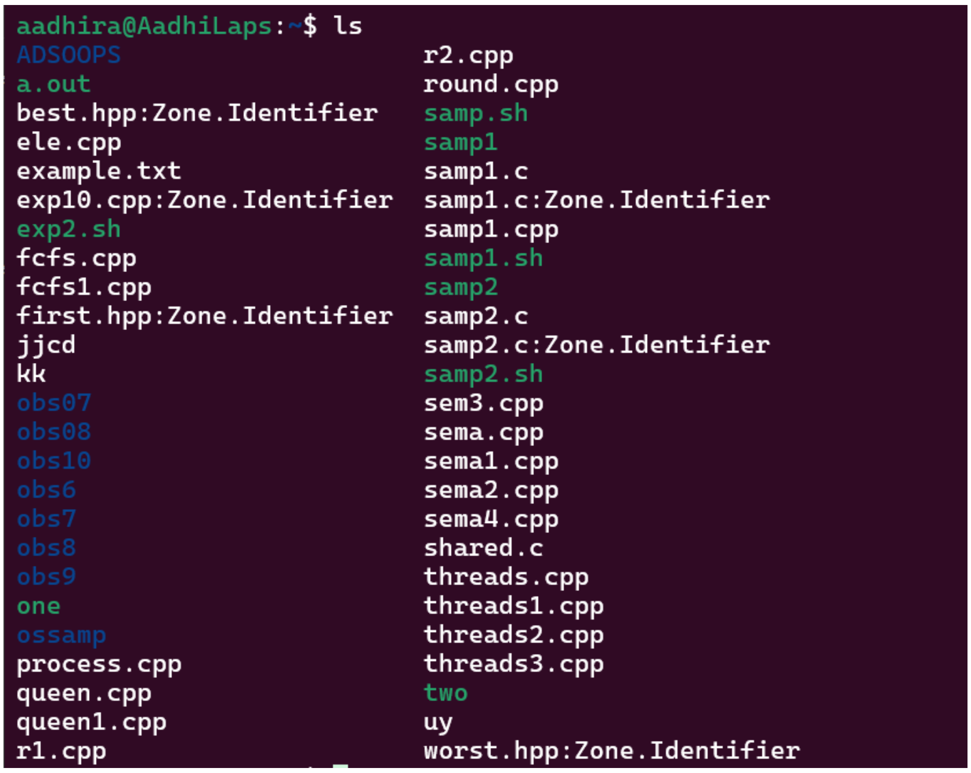
Common commands to manipulate files include **cp** for copying, **mv** for moving or renaming, and **cat** or **less** for viewing contents.

```
# Copy a file
cp source.txt destination.txt

# Move or rename a file
mv oldname.txt newname.txt

# View file content
cat file.txt
less file.txt
```

Listing Files and Directories

Command: **ls**

Explanation: This command lists the files and directories in the current directory. Adding options such as **-l** (long format) or **-a** (including hidden files) can provide more detailed or comprehensive listings.



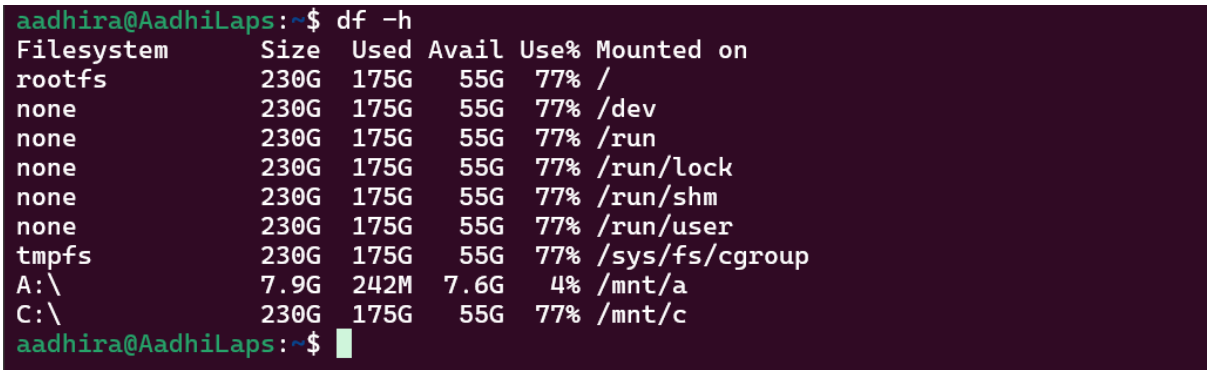
```
aadhira@AadhiLaps:~$ ls
ADSOOPS                     r2.cpp
a.out                       round.cpp
best.hpp:Zone.Identifier    samp.sh
ele.cpp                     samp1
example.txt                 samp1.c
exp10.cpp:Zone.Identifier   samp1.c:Zone.Identifier
exp2.sh                     samp1.cpp
fcfs.cpp                    samp1.sh
fcfs1.cpp                   samp2
first.hpp:Zone.Identifier   samp2.c
jjcd                        samp2.c:Zone.Identifier
kk                          samp2.sh
obs07                       sem3.cpp
obs08                       sema.cpp
obs10                       sema1.cpp
obs6                        sema2.cpp
obs7                        sema4.cpp
obs8                        shared.c
obs9                        threads.cpp
one                         threads1.cpp
ossamp                      threads2.cpp
process.cpp                 threads3.cpp
queen.cpp                   two
queen1.cpp                  uy
r1.cpp                      worst.hpp:Zone.Identifier
```

Disk Space Management

ext4 uses several mechanisms to manage disk space efficiently:

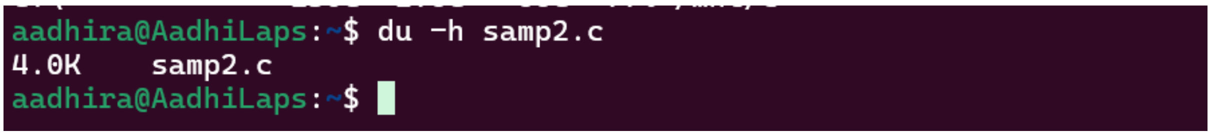
1. **Journal**: ext4 maintains a journal to keep track of changes that have not yet been committed to the main file system, providing crash recovery capabilities.
2. **Quota Management**: Admins can set disk quotas to limit the amount of disk space or the number of inodes that users or groups can use.
3. **Filesystem Checks**: Tools like **fsck** are used to check and repair ext4 file systems, ensuring integrity and fixing errors.

1. **Checking Disk Usage**:
   - **Command**: **df -h**
   - **Explanation**: This command displays the disk space usage for all mounted filesystems in a human-readable format (**-h**). It shows information such as total size, used space, available space, and the filesystem type.



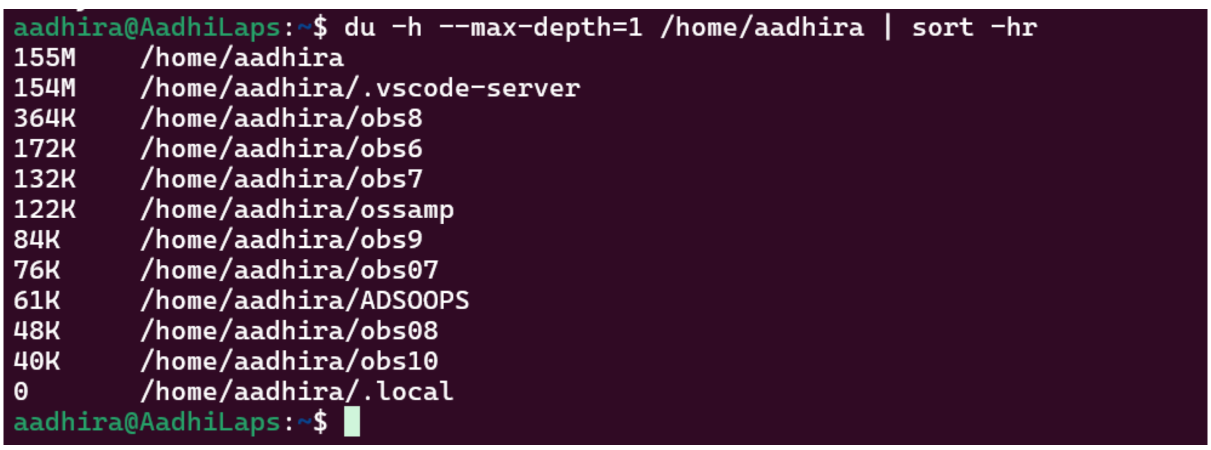
```
aadhira@AadhiLaps:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
rootfs          230G  175G   55G  77% /
none            230G  175G   55G  77% /dev
none            230G  175G   55G  77% /run
none            230G  175G   55G  77% /run/lock
none            230G  175G   55G  77% /run/shm
none            230G  175G   55G  77% /run/user
tmpfs           230G  175G   55G  77% /sys/fs/cgroup
A:\             7.9G  242M  7.6G   4% /mnt/a
C:\             230G  175G   55G  77% /mnt/c
aadhira@AadhiLaps:~$
```

2. **Checking File and Directory Sizes**:
   - **Command**: **du -h filename/directory**
   - **Explanation**: This command displays the disk space usage of the specified file or directory in a human-readable format. It provides information on the size of individual files or directories.



```
aadhira@AadhiLaps:~$ du -h samp2.c
4.0K    samp2.c
aadhira@AadhiLaps:~$
```

3. **Monitoring Disk Usage**:
    - **Command**: **du -h --max-depth=1 /path | sort -hr**
    - **Explanation**: This command shows disk usage for each directory within a specified path, sorted by size in a human-readable format. It helps identify which directories are consuming the most space.



```
aadhira@AadhiLaps:~$ du -h --max-depth=1 /home/aadhira | sort -hr
155M    /home/aadhira
154M    /home/aadhira/.vscode-server
364K    /home/aadhira/obs8
172K    /home/aadhira/obs6
132K    /home/aadhira/obs7
122K    /home/aadhira/ossamp
84K     /home/aadhira/obs9
76K     /home/aadhira/obs07
61K     /home/aadhira/ADSOOPS
48K     /home/aadhira/obs08
40K     /home/aadhira/obs10
0       /home/aadhira/.local
aadhira@AadhiLaps:~$
```

These commands and utilities allow users to create, manipulate, and manage files and directories, as well as monitor and manage disk space usage effectively in Ubuntu.

Linux File Permissions

Linux file permissions control the access level for files and directories using a combination of read (r), write (w), and execute (x) permissions. Permissions are assigned to three classes of users:

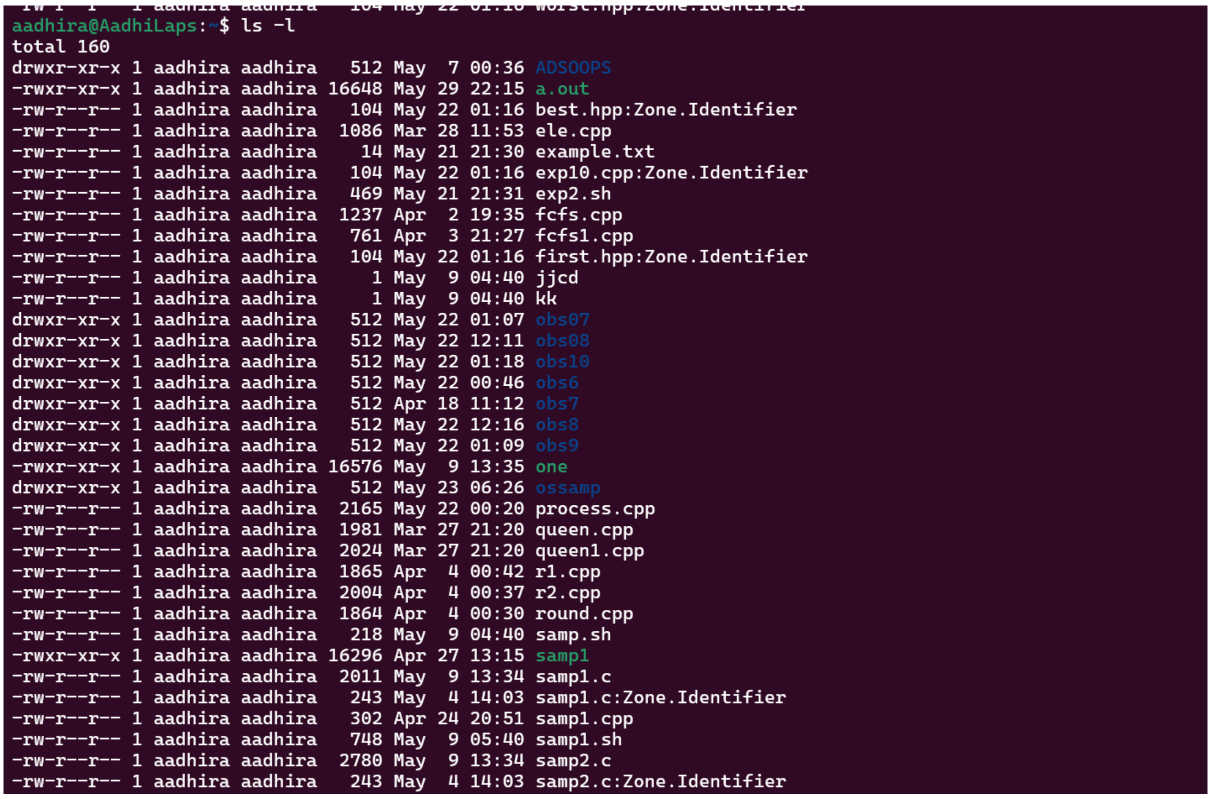
- **Owner**: The user who owns the file.
- **Group**: A group of users who share access to the file.
- **Others**: All other users.

Permissions are displayed as a 10-character string (e.g., **-rwxr-xr--**):

- The first character indicates the file type (- for a regular file, **d** for a directory).
- The next nine characters represent the permissions for the owner, group, and others, respectively.

LISTING FILES WITH DETAILED PERMISSIONS

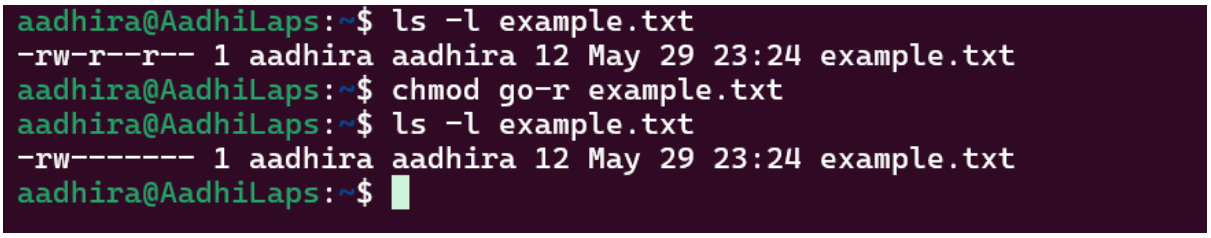
To list files with detailed permissions in the current directory:



```
aadhira@AadhiLaps:~$ ls -l
total 160
drwxr-xr-x 1 aadhira aadhira   512 May  7 00:36 ADSOOPS
-rwxr-xr-x 1 aadhira aadhira 16648 May 29 22:15 a.out
-rw-r--r-- 1 aadhira aadhira   104 May 22 01:16 best.hpp:Zone.Identifier
-rw-r--r-- 1 aadhira aadhira  1086 Mar 28 11:53 ele.cpp
-rw-r--r-- 1 aadhira aadhira    14 May 21 21:30 example.txt
-rw-r--r-- 1 aadhira aadhira   104 May 22 01:16 exp10.cpp:Zone.Identifier
-rw-r--r-- 1 aadhira aadhira   469 May 21 21:31 exp2.sh
-rw-r--r-- 1 aadhira aadhira  1237 Apr  2 19:35 fcfs.cpp
-rw-r--r-- 1 aadhira aadhira   761 Apr  3 21:27 fcfs1.cpp
-rw-r--r-- 1 aadhira aadhira   104 May 22 01:16 first.hpp:Zone.Identifier
-rw-r--r-- 1 aadhira aadhira     1 May  9 04:40 jjcd
-rw-r--r-- 1 aadhira aadhira     1 May  9 04:40 kk
drwxr-xr-x 1 aadhira aadhira   512 May 22 01:07 obs07
drwxr-xr-x 1 aadhira aadhira   512 May 22 12:11 obs08
drwxr-xr-x 1 aadhira aadhira   512 May 22 01:18 obs10
drwxr-xr-x 1 aadhira aadhira   512 May 22 00:46 obs6
drwxr-xr-x 1 aadhira aadhira   512 Apr 18 11:12 obs7
drwxr-xr-x 1 aadhira aadhira   512 May 22 12:16 obs8
drwxr-xr-x 1 aadhira aadhira   512 May 22 01:09 obs9
-rwxr-xr-x 1 aadhira aadhira 16576 May  9 13:35 one
drwxr-xr-x 1 aadhira aadhira   512 May 23 06:26 ossamp
-rw-r--r-- 1 aadhira aadhira  2165 May 22 00:20 process.cpp
-rw-r--r-- 1 aadhira aadhira  1981 Mar 27 21:20 queen.cpp
-rw-r--r-- 1 aadhira aadhira  2024 Mar 27 21:20 queen1.cpp
-rw-r--r-- 1 aadhira aadhira  1865 Apr  4 00:42 r1.cpp
-rw-r--r-- 1 aadhira aadhira  2004 Apr  4 00:37 r2.cpp
-rw-r--r-- 1 aadhira aadhira  1864 Apr  4 00:30 round.cpp
-rw-r--r-- 1 aadhira aadhira   218 May  9 04:40 samp.sh
-rwxr-xr-x 1 aadhira aadhira 16296 Apr 27 13:15 samp1
-rw-r--r-- 1 aadhira aadhira  2011 May  9 13:34 samp1.c
-rw-r--r-- 1 aadhira aadhira   243 May  4 14:03 samp1.c:Zone.Identifier
-rw-r--r-- 1 aadhira aadhira   302 Apr 24 20:51 samp1.cpp
-rw-r--r-- 1 aadhira aadhira   748 May  9 05:40 samp1.sh
-rw-r--r-- 1 aadhira aadhira  2780 May  9 13:34 samp2.c
-rw-r--r-- 1 aadhira aadhira   243 May  4 14:03 samp2.c:Zone.Identifier
```

Changing File Permissions

The file **example.txt** initially had permissions **-rw-r--r--**, allowing read and write access for the owner (**aadhira**), and read-only access for the group and others.

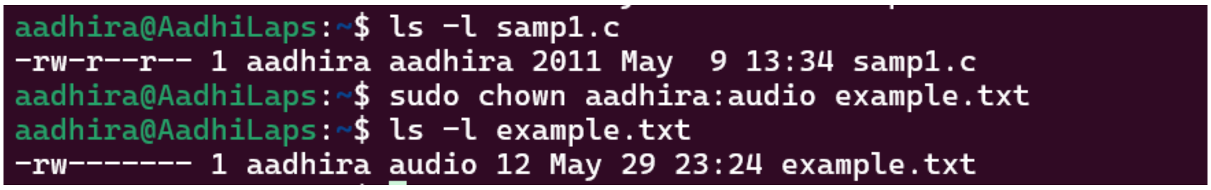


```
aadhira@AadhiLaps:~$ ls -l example.txt
-rw-r--r-- 1 aadhira aadhira 12 May 29 23:24 example.txt
aadhira@AadhiLaps:~$ chmod go-r example.txt
aadhira@AadhiLaps:~$ ls -l example.txt
-rw------- 1 aadhira aadhira 12 May 29 23:24 example.txt
aadhira@AadhiLaps:~$
```

Using the command **chmod go-r example.txt**, read permissions for the group and others were removed.
The new permissions became **-rw-------**, meaning only the owner (**aadhira**) can read and write to the file, while the group and others have no permissions.

Changing File Ownership

Change the ownership of **round.cpp** to user **username** and group **groupname**



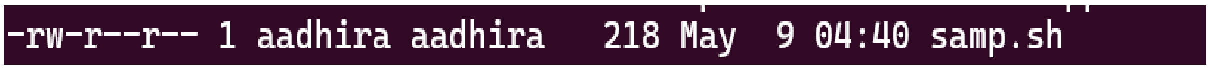
```
aadhira@AadhiLaps:~$ ls -l samp1.c
-rw-r--r-- 1 aadhira aadhira 2011 May  9 13:34 samp1.c
aadhira@AadhiLaps:~$ sudo chown aadhira:audio example.txt
aadhira@AadhiLaps:~$ ls -l example.txt
-rw------- 1 aadhira audio 12 May 29 23:24 example.txt
```

sudo chown new_owner:new_group example.txt

Special Permission Bits (SUID, SGID, Sticky Bit)

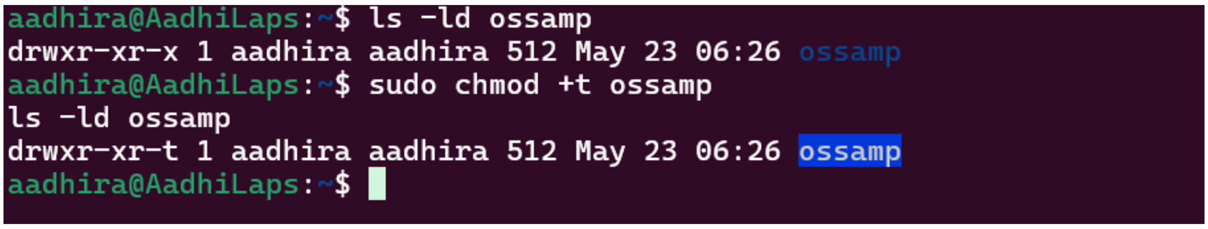
A file with **SUID** always executes as the user who owns the file, regardless of the user passing the command. If the file owner doesn't have execute permissions, then use an uppercase **S** here.

Set the SUID bit on **samp.sh**



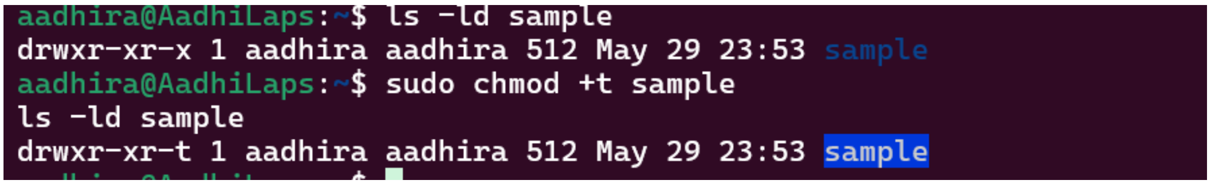
```
-rw-r--r-- 1 aadhira aadhira   218 May  9 04:40 samp.sh
```

Set the **SGID bit** on a directory (assuming there's a directory named **ossamp**):



```
aadhira@AadhiLaps:~$ ls -ld ossamp
drwxr-xr-x 1 aadhira aadhira 512 May 23 06:26 ossamp
aadhira@AadhiLaps:~$ sudo chmod +t ossamp
ls -ld ossamp
drwxr-xr-t 1 aadhira aadhira 512 May 23 06:26 ossamp
aadhira@AadhiLaps:~$
```

Set the **Sticky Bit** on a directory:



```
aadhira@AadhiLaps:~$ ls -ld sample
drwxr-xr-x 1 aadhira aadhira 512 May 29 23:53 sample
aadhira@AadhiLaps:~$ sudo chmod +t sample
ls -ld sample
drwxr-xr-t 1 aadhira aadhira 512 May 29 23:53 sample
```

Memory Management

In Ubuntu, memory management is a crucial aspect handled by the operating system's kernel, which employs various strategies and algorithms to efficiently allocate, manage, and monitor system memory. Here's a summarized overview of memory management in Ubuntu:

Memory Allocation Algorithms:

Ubuntu's kernel utilizes various memory allocation algorithms, including the buddy system, to manage physical memory effectively.
Other algorithms such as First Fit (FF), Best Fit (BF), and Worst Fit (WF) may also be used in certain contexts, particularly in user-space memory allocation libraries.

Kernel Memory Management:

The Linux kernel in Ubuntu manages both physical and virtual memory, handling tasks such as page allocation, swapping, and memory protection.
It employs the buddy system algorithm for managing physical memory pages, dividing them into power-of-two-sized blocks and merging adjacent free blocks to optimize memory utilization.

Utilities and Tools:

Ubuntu provides various utilities and tools for monitoring and managing memory usage, including command-line tools like top, htop, and free.
Advanced tools such as valgrind and gdb can be used for memory debugging, profiling, and analysis.

Memory Allocation Libraries:

User-space programs in Ubuntu typically use memory allocation functions provided by the GNU C Library (glibc), such as malloc, calloc, and realloc.
These libraries may internally use different memory allocation algorithms, including the buddy system, depending on the implementation and configuration.

Tuning and Configuration:

Ubuntu allows tuning and configuration of memory management parameters through kernel parameters and system settings.
Users can adjust parameters related to memory allocation, swapping behavior, and cache management to optimize system performance based on workload requirements.

Overall, memory management in Ubuntu involves a combination of kernel-level functionality, user-space libraries, and system utilities to efficiently allocate, manage, and monitor system memory usage. The buddy system, along with other memory management techniques, helps ensure optimal memory utilization and performance in Ubuntu and other Linux-based operating systems.

The buddy system is a memory allocation algorithm commonly used in operating systems to manage physical memory efficiently. In Ubuntu, like many other Linux distributions, the buddy system is employed by the kernel for memory allocation.
Here's how the buddy system works:

Memory Partitioning:

The available physical memory is divided into fixed-size blocks, typically powers of 2, such as 4KB, 8KB, 16KB, etc.
Each memory block is further split into smaller blocks, known as "buddies," which are halves of the original block size.

Binary Tree Structure:

The memory blocks are organized in a binary tree structure, where each node represents a memory block and its children represent its buddies.
The root of the tree represents the entire memory space, and each level of the tree represents a different block size.

Allocation:

When a memory allocation request is received, the buddy system searches the binary tree for the smallest free block that can accommodate the requested size.
If the found block is larger than needed, it is split into smaller blocks until an appropriately sized block is obtained.

Deallocation:

When memory is deallocated, the buddy system merges adjacent free blocks to form larger free blocks.
It uses the binary tree structure to efficiently locate and merge buddies.
The merging process continues recursively until it reaches the root of the tree or encounters a non-free buddy.

Fragmentation Handling:
The buddy system aims to minimize both internal and external fragmentation. Internal fragmentation occurs when allocated memory is larger than needed, leaving unused space within allocated blocks. The buddy system minimizes internal fragmentation by splitting blocks into smaller sizes only when necessary.

External fragmentation happens when there are many small free memory blocks scattered throughout the memory. The buddy system helps mitigate external fragmentation by merging adjacent free blocks into larger ones during deallocation.

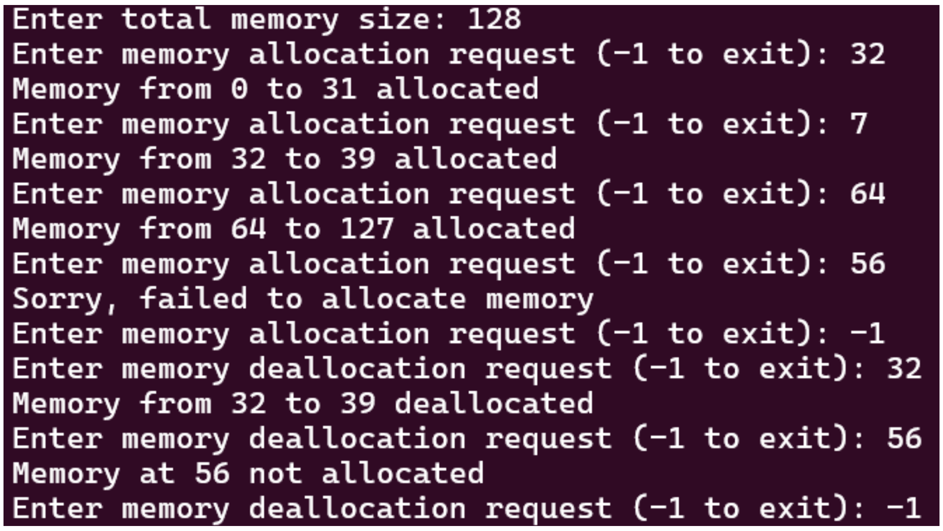
In Ubuntu, the buddy system plays a significant role in managing physical memory, providing efficient memory allocation and deallocation to ensure optimal system performance. It is an essential component of the kernel's memory management subsystem, contributing to effective utilization of system resources.

Buddy System Algorithm

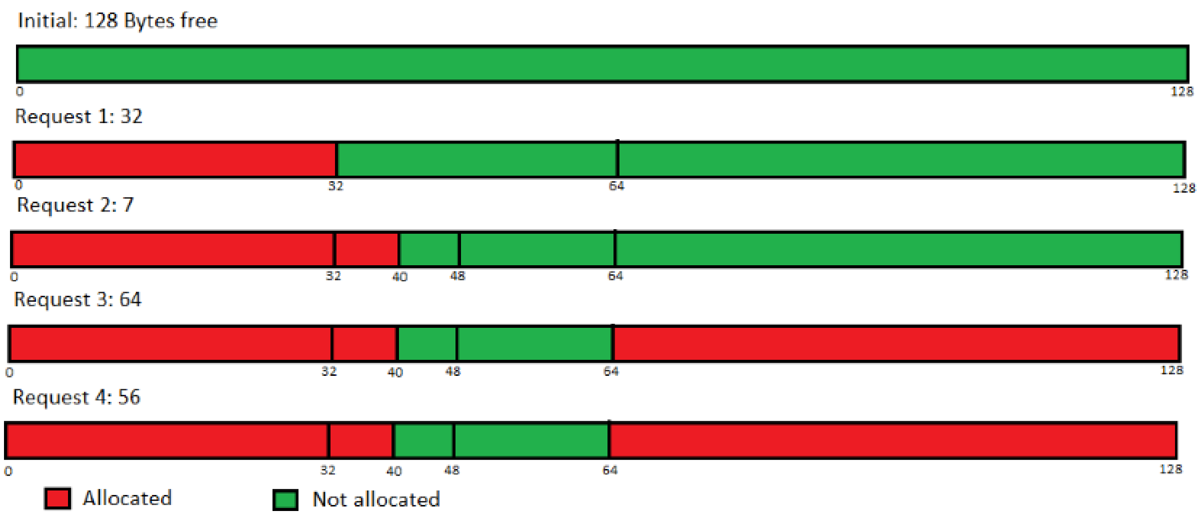
1. Initialization: a. Prompt the user to enter the total memory size. b. Calculate the maximum number of powers of 2 that can fit into the given size. c. Initialize the free list with one large block representing the total available memory.
2. Memory Allocation: a. Prompt the user to enter the memory allocation request size. b. Calculate the appropriate block size (smallest power of 2 greater than or equal to the requested size). c. If a block of the required size is available in the free list, allocate it:
    - Remove the block from the free list.
    - Map the starting address of the block to its size.
    - Print the allocation details. d. If no block of the required size is available:
    - Search for the next larger block in the free list.
    - If a larger block is found:
        - Split the larger block into two halves until the required block size is obtained.
        - Allocate the required block and map its starting address to its size.
        - Print the allocation details.
    - If no larger block is found, print an error message indicating failure to allocate memory. e. Continue accepting memory allocation requests until the user exits.

3. Memory Deallocation: a. Prompt the user to enter the starting address of the memory block to deallocate. b. If the starting address is not found in the map, print an error message indicating that the memory was not allocated. c. If the starting address is found:
    - Retrieve the size of the allocated block.
    - Attempt to merge the block with its buddy block (if available and free):
        - Calculate the buddy block's address.
        - Check if the buddy block is free and of the same size.
        - If so, merge the blocks and update the starting address and size for the next iteration.
    - Add the final (possibly merged) block back to the free list.
    - Print the deallocation details. d. Continue accepting memory deallocation requests until the user exits.
4. Program Termination: a. Exit the program after processing all allocation and deallocation requests.

OUTPUT



```
Enter total memory size: 128
Enter memory allocation request (-1 to exit): 32
Memory from 0 to 31 allocated
Enter memory allocation request (-1 to exit): 7
Memory from 32 to 39 allocated
Enter memory allocation request (-1 to exit): 64
Memory from 64 to 127 allocated
Enter memory allocation request (-1 to exit): 56
Sorry, failed to allocate memory
Enter memory allocation request (-1 to exit): -1
Enter memory deallocation request (-1 to exit): 32
Memory from 32 to 39 deallocated
Enter memory deallocation request (-1 to exit): 56
Memory at 56 not allocated
Enter memory deallocation request (-1 to exit): -1
```



A concise comparison between the buddy system and the First Fit algorithm in memory management:

Buddy System:

- Divides memory into fixed-size blocks, typically powers of 2, and maintains them in a binary tree structure.
- Allocates memory by searching for the smallest free block that can accommodate the request and splitting it if necessary.
- Deals with fragmentation by merging adjacent free blocks during deallocation.
- Minimizes both internal and external fragmentation.

First Fit:

- Searches memory sequentially for the first available block that meets the allocation request.
- Typically does not perform merging of free blocks during deallocation.
- May suffer from fragmentation, especially external fragmentation, due to its allocation strategy.
- In essence, the buddy system offers a structured approach to memory allocation and management, aiming to minimize fragmentation and improve efficiency compared to the simpler, sequential allocation strategy of First Fit.

Disk Scheduling –CFQ

Overview

The Completely Fair Queuing (CFQ) scheduler is an I/O scheduler designed to allocate disk access time to processes in a fair and efficient manner. It organizes I/O requests into per-process queues and assigns access time based on each process's I/O priority. This prioritization ensures that higher priority tasks receive more immediate attention while still maintaining overall fairness.

Key Components

- **Per-Process I/O Queues:** Each process has its own I/O queue, allowing the scheduler to manage requests individually.
- **I/O Priority:** Requests are prioritized based on the process's I/O priority, influencing how access time is distributed.
- **Batched Queues for Asynchronous Requests:** Asynchronous requests from all processes are grouped into fewer queues according to their I/O priority.

Scheduling Classes

The CFQ scheduler divides processes into three main classes, each with different priority levels:

1. **Real-Time (Highest Priority):**
   - Reserved for time-sensitive processes that require immediate attention. o Subdivided into eight priority levels (0 to 7), with 0 being the highest and 7 the lowest.
2. **Best Effort:**
   - The default class for most processes. o Also subdivided into eight priority levels (0 to 7), with 0 being the highest and 7 the lowest.
   - Default priority within this class is 4.
3. **Idle (Lowest Priority):**
   - Assigned to processes that do not need immediate I/O access.
   - Only serviced when there are no pending requests from the Real-Time **or Best Effort classes.**

Scheduling Mechanism

- **Priority Enforcement:** Real-Time class processes are serviced before Best Effort class processes, which in turn are serviced before Idle class processes. This hierarchical structure ensures that high-priority tasks receive the required processor time, potentially causing lower priority tasks to experience delays or "starvation."
- **Idle Class Servicing:** Idle class processes are only serviced when no other I/O requests are pending, making it crucial to assign this class only to **processes that can tolerate delays without affecting system performance.**

Example of Operation

1. A process in the Real-Time class with priority 0 submits an I/O request. This request is handled before any Best Effort or Idle class requests.
2. If no Real-Time class requests are pending, the scheduler then handles Best Effort class requests, starting with the highest priority within this class.
3. Idle class requests are only serviced if there are no pending requests from the other two classes.

Conclusion

The CFQ scheduler effectively balances fairness and efficiency by organizing I/O requests into per-process queues and assigning them access time based on priority. This approach ensures that critical tasks receive prompt attention while maintaining overall system performance. The use of multiple scheduling classes and priority levels allows for granular control over I/O resource allocation, making CFQ suitable for a variety of workloads and system requirements.

How CFQ Works

1. **Queue Management**: CFQ maintains multiple queues, one for each process or thread. Each queue represents the I/O requests of a single process.
2. **Time Slices**: Each queue is allocated a time slice during which it can issue I/O requests. The length of these time slices can be adjusted dynamically based on the process's I/O patterns.
3. **Priority Classes**: CFQ classifies requests into different priority classes, typically including real-time, best-effort, and idle classes. Real-time processes get the highest priority, followed by best-effort, and then idle.
4. **Request Dispatch**: Within each priority class, CFQ distributes time slices among the processes, ensuring that each process gets a fair chance to access the disk. This distribution is done in a round-robin fashion.
5. **Seek Minimization**: CFQ attempts to minimize disk head movement by servicing requests in an order that reduces seek times, while still adhering to the fairness constraints.

Optimization Techniques

- **Adaptive Time Slices**: CFQ adjusts the time slices based on the I/O behavior of processes. For instance, a process performing sequential reads may get longer time slices compared to one performing random reads.
- **Anticipatory Scheduling**: CFQ incorporates elements of anticipatory scheduling by waiting briefly after servicing a request, anticipating that the next request from the same process may be nearby on the disk.
- **Bandwidth Allocation**: CFQ dynamically adjusts bandwidth allocation to balance throughput and latency. For instance, high-priority tasks can get more bandwidth during high load periods.

Benefits of CFQ

- **Fairness**: Ensures that all processes get a fair share of disk access time, preventing any single process from monopolizing the disk.
- **Performance**: Balances between achieving high throughput and maintaining low latency, making it suitable for a wide range of workloads.
- **Flexibility**: Can handle different types of I/O patterns and dynamically adapts to changing workloads.

Limitations of CFQ

- **Complexity**: CFQ is more complex to implement and manage compared to simpler algorithms like FIFO or Round Robin.
- **Workload Sensitivity**: Performance can vary significantly depending on the nature of the workload. For example, CFQ may not perform as well with highly random I/O patterns.

Page Replacement Algorithms

Introduction

Page replacement algorithms are essential in managing the limited physical memory of a system. When the memory is full and a new page needs to be loaded, the system must decide which page to evict. Common page replacement algorithms include FIFO (First-In, First-Out), LRU (Least Recently Used), and Clock.

FIFO (First-In, First-Out)

FIFO is one of the simplest page replacement algorithms. It evicts the oldest page in memory, which is the page that has been in memory the longest.

- **Mechanism**: Maintains a queue of pages in the order they were loaded. When a page needs to be replaced, the page at the front of the queue (the oldest) is evicted.
- **Advantages**: Simple to implement and understand.
- **Disadvantages**: Does not consider how frequently or recently a page has been accessed, which can lead to suboptimal performance.

LRU (Least Recently Used)

LRU evicts the page that has not been used for the longest time, approximating the optimal algorithm which evicts the page that will not be used for the longest time in the future.

- **Mechanism**: Tracks the access history of pages and uses this information to determine which page has been least recently used.
- **Advantages**: Generally provides better performance than FIFO because it takes into account the usage patterns of pages.
- **Disadvantages**: Can be complex to implement efficiently due to the need to maintain and update access information.

Clock Algorithm

The Clock algorithm is a more efficient approximation of LRU. It maintains a circular list (or "clock") of pages and uses a reference bit to track page usage.

How Clock Works

1. **Reference Bit**: Each page entry in the clock has an associated reference bit that is set when the page is accessed.
2. **Pointer**: A "clock hand" or pointer moves through the circular list of pages.
3. **Eviction Process**: When a page needs to be replaced, the algorithm checks the reference bit of the page at the clock hand's position:
o If the reference bit is 0, the page is evicted. o If the reference bit is 1, the bit is cleared (set to 0) and the clock hand moves to the next page.
4. **Iteration**: This process repeats until a page with a reference bit of 0 is found.

Advantages of Clock

- **Efficiency**: The Clock algorithm is more efficient than LRU because it requires less frequent updates to the reference bits and uses a simpler data structure.
- **Performance**: Provides a good balance between simplicity and performance, often performing nearly as well as LRU but with lower overhead.

Page Replacement Algorithms: Second Chance (Clock) Policy

Apart from LRU, OPT, and FIFO page replacement policies, we also have the Second Chance/Clock page replacement policy. In the Second Chance page replacement policy, the candidate pages for removal are considered in a round-robin manner, and a page that has been accessed between consecutive considerations will not be replaced. The page replaced is the one that, when considered in a round-robin manner, has not been accessed since its last consideration.

It can be implemented by adding a "second chance" bit to each memory frame. Every time the frame is considered (due to a reference made to the page inside it), this bit is set to 1, which gives the page a second chance. When we consider the candidate page for replacement, we replace the first one with this bit set to 0 (while zeroing out bits of the other pages we see in the process). Thus, a page with the "second chance" bit set to 1 is never replaced during the first consideration and will only be replaced if all the other pages deserve a second chance too!

Traditionally, Second Chance and Clock are believed to be less efficient than LRU (having a higher miss ratio). Recent research from Carnegie Mellon University finds that Second Chance and Clock are more efficient than LRU with a lower miss ratio. Because Second Chance and Clock are faster and more scalable than LRU, this means LRU has no advantage over Second Chance and Clock anymore.

Example – Let’s say the reference string is 0, 4, 1, 4, 2, 4, 3, 4, 2, 4, 0, 4, 1, 4, 2, 4, 3, 4, and we have 3 frames. Let’s see how the algorithm proceeds by tracking the second chance bit and the pointer.
Initially, all frames are empty, so after the first 3 passes they will be filled with {0, 4, 1}, and the second chance array will be {0, 0, 0} as none has been referenced yet. Also, the pointer will cycle back to 0.

Pass-4: Frame = {0, 4, 1}, second_chance = {0, 1, 0} [4 will get a second chance], pointer = 0 (No page needed to be updated so the candidate is still the page in frame 0), pf = 3 (No increase in page fault number).

Pass-5: Frame = {2, 4, 1}, second_chance = {0, 1, 0} [0 replaced; its second chance bit was 0, so it didn’t get a second chance], pointer = 1 (updated), pf = 4.

Pass-6: Frame = {2, 4, 1}, second_chance = {0, 1, 0}, pointer = 1, pf = 4 (No change).

Pass-7: Frame = {2, 4, 3}, second_chance = {0, 0, 0} [4 survived but its second chance bit became 0], pointer = 0 (as the element at index 2 was finally replaced), pf = 5.

Pass-8: Frame = {2, 4, 3}, second_chance = {0, 1, 0} [4 referenced again], pointer = 0, pf = 5.

Pass-9: Frame = {2, 4, 3}, second_chance = {1, 1, 0} [2 referenced again], pointer = 0, pf = 5.

Pass-10: Frame = {2, 4, 3}, second_chance = {1, 1, 0}, pointer = 0, pf = 5 (no change).

Pass-11: Frame = {2, 4, 0}, second_chance = {0, 0, 0}, pointer = 0, pf = 6 (2 and 4 got second chances).

Pass-12: Frame = {2, 4, 0}, second_chance = {0, 1, 0}, pointer = 0, pf = 6 (4 will again get a second chance).

Pass-13: Frame = {1, 4, 0}, second_chance = {0, 1, 0}, pointer = 1, pf = 7 (pointer updated, pf updated).

Pass-14: Frame = {1, 4, 0}, second_chance = {0, 1, 0}, pointer = 1, pf = 7 (No change).

Pass-15: Frame = {1, 4, 2}, second_chance = {0, 0, 0}, pointer = 0, pf = 8 (4 survived again due to 2nd chance!).

Pass-16: Frame = {1, 4, 2}, second_chance = {0, 1, 0}, pointer = 0, pf = 8 (2nd chance updated).

Pass-17: Frame = {3, 4, 2}, second_chance = {0, 1, 0}, pointer = 1, pf = 9 (pointer, pf updated).

Pass-18: Frame = {3, 4, 2}, second_chance = {0, 1, 0}, pointer = 1, pf = 9 (No change).

In this example, the second chance algorithm does as well as the LRU method, which is much more expensive to implement in hardware.

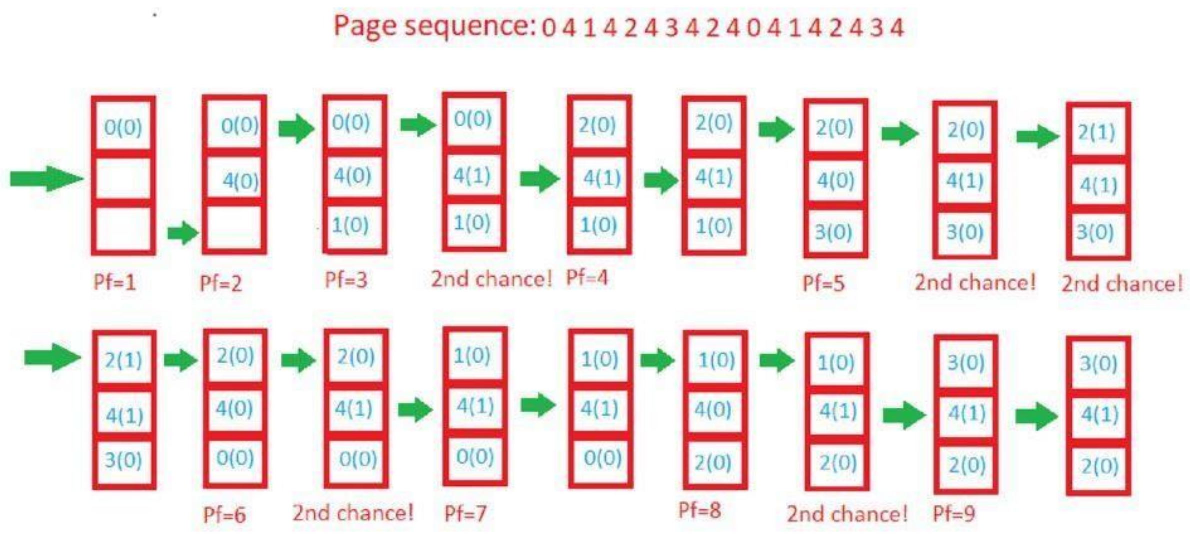
More Examples –

- **Input:** 2, 5, 10, 1, 2, 2, 6, 9, 1, 2, 10, 2, 6, 1, 2, 1, 6, 9, 5, 1
  - **Frames:** 3
  - **Output:** 14
- **Input:** 2, 5, 10, 1, 2, 2, 6, 9, 1, 2, 10, 2, 6, 1, 2, 1, 6, 9, 5, 1
  - **Frames:** 4 o **Output:** 11

Algorithm

1. Create an array frames to track the pages currently in memory and another Boolean array second_chance to track whether that page has been accessed since its last replacement (that is, if it deserves a second chance or not) and a variable pointer to track the target for replacement.
2. Start traversing the array. If the page already exists, simply set its corresponding element in second_chance to true and return.
3. If the page doesn't exist, check whether the space pointed to by pointer is empty (indicating cache isn't full yet). If so, put the element there and return. Otherwise, traverse the array arr one by one (cyclically using the value of pointer), marking all corresponding second_chance elements as false, until finding one that's already false. That is the most suitable page for replacement, so do so and return.
4. Finally, report the page fault count.

This implementation ensures efficient page replacement with minimal overhead, making it a practical choice for modern operating systems.



Virtual memory & TLB

What is a TLB?

The Translation Lookaside Buffer (TLB) is a specialized cache used to improve the speed of virtual address translation. It is part of the memory management unit (MMU) in a computer's CPU.

TLB and Ubuntu:

In Ubuntu, the Translation Lookaside Buffer (TLB) operates in conjunction with the Linux kernel and hardware MMU (Memory Management Unit) to optimize virtual memory management.
The TLB is a specialized cache used to store recent translations of virtual addresses to physical addresses, which significantly speeds up memory access. By caching these address translations, the TLB reduces the need for frequent and time-consuming page table lookups, thus improving overall system performance.

TLB in Virtual Memory Management on Ubuntu

The Translation Lookaside Buffer (TLB) is an essential component in Ubuntu's virtual memory management system. It helps optimize the translation of virtual addresses to physical addresses, significantly enhancing performance.

Role of TLB in Ubuntu 1. Address Translation

Ubuntu, like other Linux distributions, uses virtual memory to allow processes to use a larger address space than the physical memory. The TLB helps speed up this address translation process.

Example

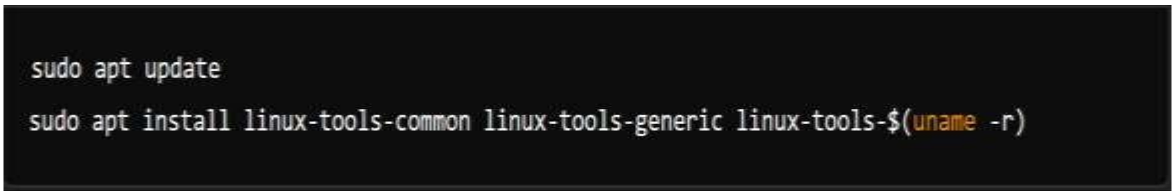
1. **Without TLB**: Every memory access requires a page table lookup:
   - A process accesses virtual address 0xBEEF1234. oThe CPU must traverse the page tables to find the physical address 0xDEAD5678. oThis lookup involves multiple memory accesses, causing delays.
2. **With TLB**: The TLB caches recent translations:

A process accesses virtual address 0xBEEF1234. oThe CPU checks the TLB and finds the cached translation to 0xDEAD5678. oThe physical address is used immediately, skipping the slow page table lookup

Performance Monitoring with perf

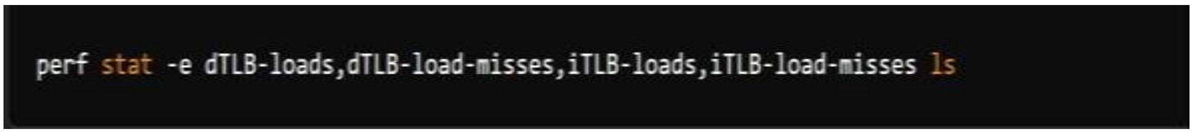
Ubuntu provides tools to monitor TLB performance, such as perf. This tool can measure the number of TLB loads and misses, helping diagnose performance issues.

Installing perf



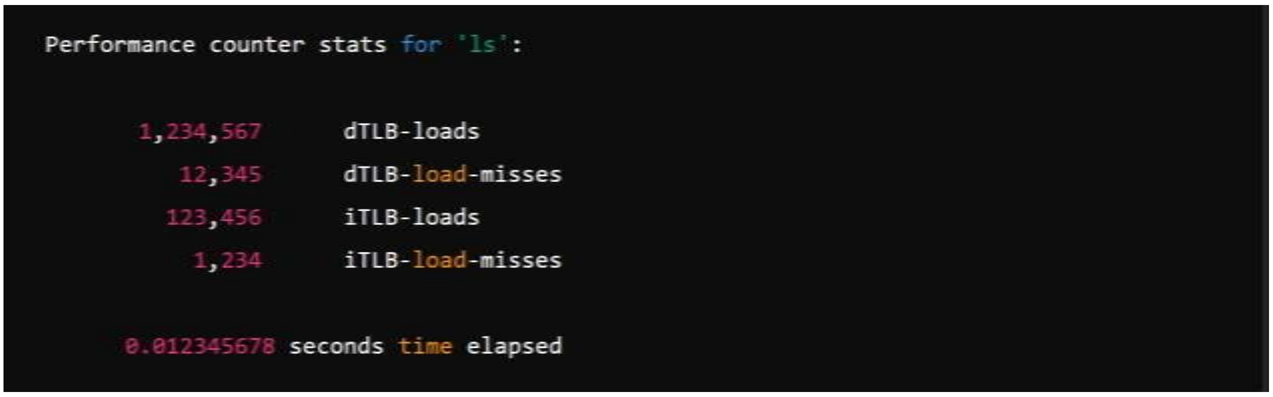
```
sudo apt update
sudo apt install linux-tools-common linux-tools-generic linux-tools-$(uname -r)
```

Using perf to Monitor TLB Performance



```
perf stat -e dTLB-loads,dTLB-load-misses,iTLB-loads,iTLB-load-misses ls
```

Example Output



```
Performance counter stats for 'ls':


       1,234,567      dTLB-loads
          12,345      dTLB-load-misses
         123,456      iTLB-loads
           1,234      iTLB-load-misses


      0.012345678 seconds time elapsed
```

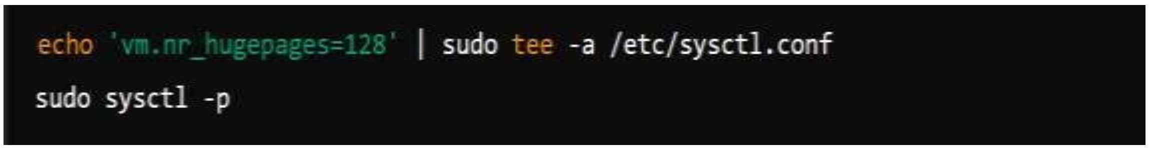
- **dTLB-loads:** Number of data TLB loads.
- **dTLB-load-misses:** Number of data TLB load misses.
- **iTLB-loads:** Number of instruction TLB loads.
- **iTLB-load-misses:** Number of instruction TLB load misses.

Reducing TLB Misses with HugePages

HugePages can help reduce TLB misses by using larger memory pages, thereby reducing the number of entries needed in the TLB.

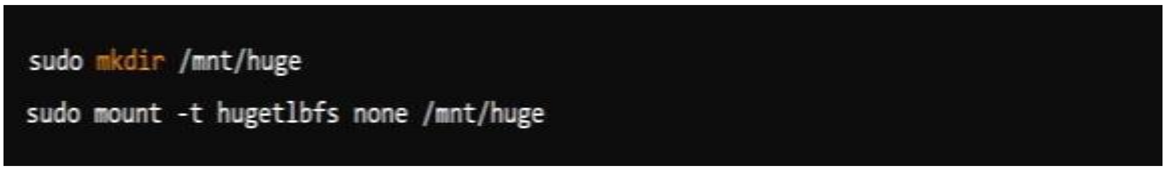
Configuring HugePages on Ubuntu

1. **Enable HugePages:**



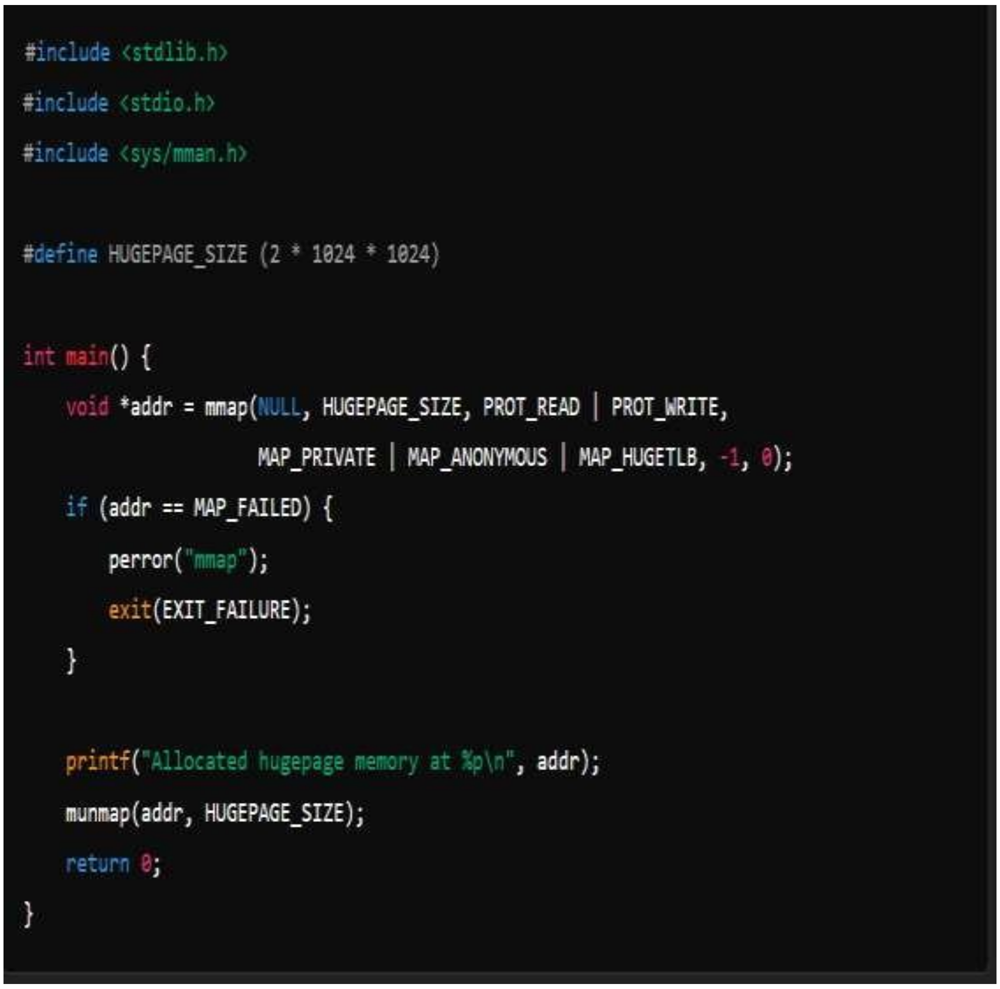
```
echo 'vm.nr_hugepages=128' | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

2. **Mount HugePages:**



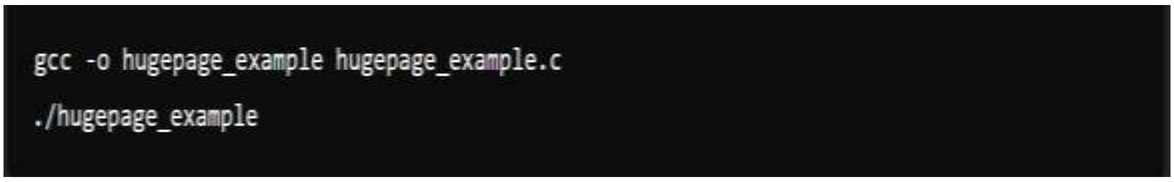
```
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs none /mnt/huge
```

Example Program Using HugePages:



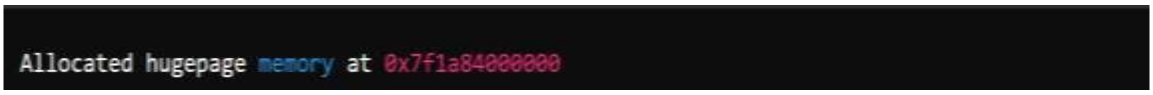
```
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>

#define HUGEPAGE_SIZE (2 * 1024 * 1024)

int main() {
    void *addr = mmap(NULL, HUGEPAGE_SIZE, PROT_READ | PROT_WRITE,
                      MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    printf("Allocated hugepage memory at %p\n", addr);
    munmap(addr, HUGEPAGE_SIZE);
    return 0;
}
```

Compile and Run:



```
gcc -o hugepage_example hugepage_example.c
./hugepage_example
```

Output



```
Allocated hugepage memory at 0x7f1a84000000
```

Impact of TLB on Performance in Ubuntu

Improved Memory Access Time

- TLB Hits: When the TLB contains the required translation, memory access is almost instantaneous.
- TLB Misses: If the translation is not in the TLB, the CPU must perform a slower page table lookup.

CPU Efficiency

- Reduced Overhead: TLB hits reduce the overhead of address translation, allowing the CPU to perform other tasks more efficiently.
- Context Switching: During context switches, TLB entries may become invalid. Techniques like TLB tagging with process identifiers help mitigate this issue.

Program Behavior

- Locality of Reference: Programs with good locality of reference (frequently accessing a small set of memory locations) benefit more from TLB, resulting in higher TLB hit rates.
- Using Large Pages: Configuring applications to use HugePages can reduce TLB misses and improve performance.

Summary

In Ubuntu, the TLB is critical for efficient virtual memory management. It reduces the time required for address translation and enhances overall system performance. Tools like perf can be used to monitor TLB performance, and configuring HugePages can further optimize memory access.

Conclusion

This report has explored key aspects of operating systems, focusing on IPC through shared memory, CFS scheduling, file allocation and protection, memory management, CFQ disk scheduling, page replacement, and virtual memory with TLB management. These components collectively ensure efficient process communication, fair resource allocation, secure file handling, optimized memory usage, and effective disk and page management.

Understanding these mechanisms highlights the importance of each element in maintaining the performance, security, and reliability of modern operating systems. Continuous advancements in these areas are essential to meet the evolving demands of contemporary computing environments.