IMPLEMENTING SYSTEM CALL

System calls serve as a link between user apps and low-level kernel functions in contemporary operating systems. A useful tool that enables applications to map files or devices into memory is mmap(), a system call of this type. This report describes the purpose, operation, and my experience implementing mmap(), which was initially meant for the Inferno OS but was finally implemented on Ubuntu because of installation issues.

What is mmap()?

The system call "memory map" (mmap()) is offered by Linux and other Unix-like operating systems. Its primary function is to directly map memory objects, devices, or files into a process's address space.

With mmap(), a file or device can be accessed as though it were a part of the program's memory, without the need for conventional file I/O (read(), write()). Performance is enhanced by this, particularly for jobs requiring a lot of memory or huge files.

How mmap()works?

The operating system creates a memory area in the process's virtual address space when mmap() is used. Because this memory area is connected to a file or device, accessing it will cause the file to be read from or written to.

Function Prototype

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
```

Main Parameters

1.addr: Preferred starting address for the mapping (usually set to NULL)

2,length: Number of bytes to map

3.prot: Memory protection (e.g., PROT_READ, PROT_WRITE)

4.flags: Mapping type (MAP_PRIVATE, MAP_SHARED)

5.fd: File descriptor of the file to be mapped

6.offset: Starting point within the file

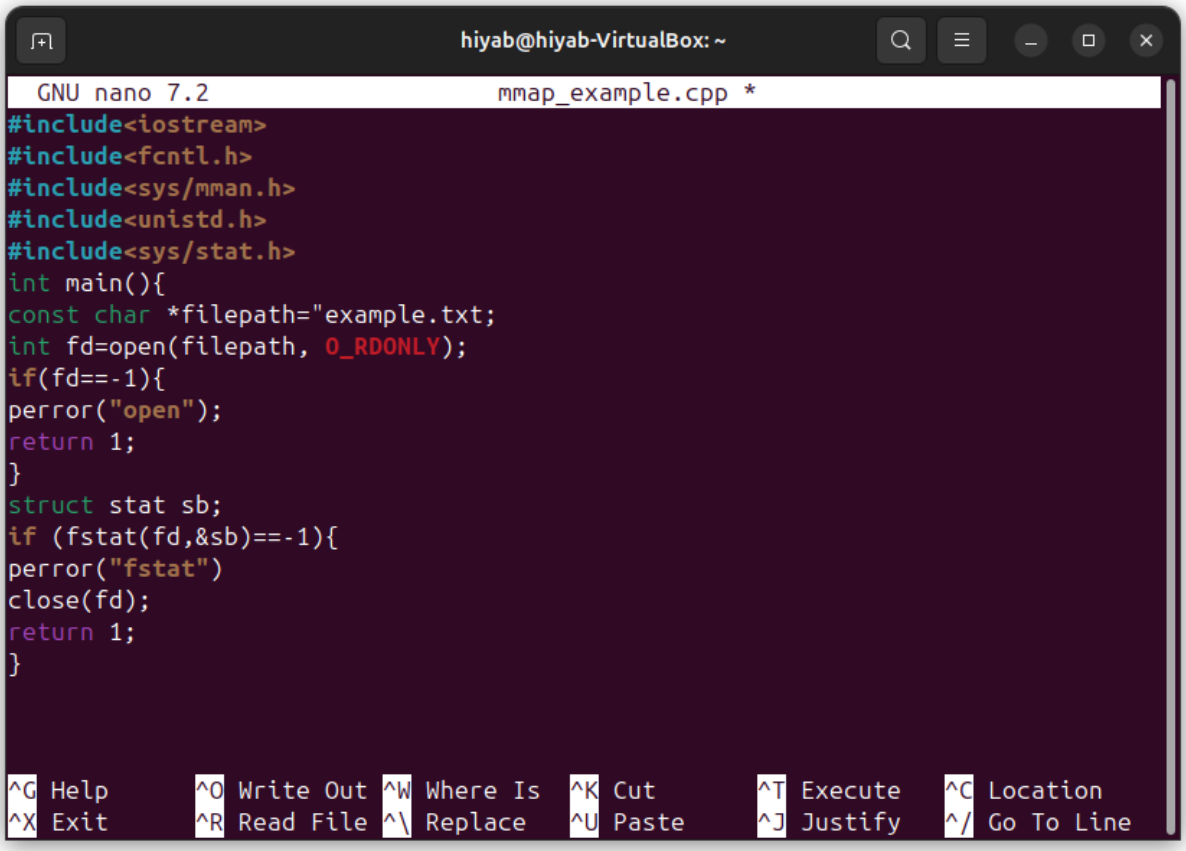
My implementation experience

I was given the task of implementing the mmap() system call on Bell Labs' Inferno OS, an operating system renowned for its portability and lightweight design. However, I ran into a number of serious issues when I tried to install and set up Inferno in a virtual environment

- Lack of proper ISO support for easy VirtualBox installation
- Complex and outdated setup instructions
- Minimal hardware support and community documentation are some of challenges I faced

Because of the challenges above, I chose to use Ubuntu Linux, a popular and POSIX-compliant operating system, to implement and test the mmap() system call.

Implementation on Ubuntu



```
hiyab@hiyab-VirtualBox: ~
  GNU nano 7.2                    mmap_example.cpp *
perror("open");
return 1;
}
struct stat sb;
if (fstat(fd,&sb)==-1){
perror("fstat")
close(fd);
return 1;
}
char *mapped=(char *)mmap(NULL,sb.st_size,PROT_READ,MAP_PRIVATE,fd,0);
if(mapped==MAP_FAILED){
perror("mmap);
close(fd);
return 1;
}
write(STDOUT_FILEND,mapped,sb.st_size);
munmap(mapped,sb.st_size);
close(fd);
return 0;
}

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^/ Go To Line
```

What the Code Does:

- Opens a text file (example.txt)
- Uses fstat() to get the file size
- Calls mmap() to map the file into memory
- Reads the content from memory and prints it
- Cleans up with munmap() and closes the file

Why mmap()used?

- ✓ permits direct memory access, avoiding read/write system calls.
- ✓ A memory-mapped file can be shared by several programs.
- ✓ allows for quicker file processing, particularly when working with binary files or big information.
- ✓ Useful in virtual memory management, caching, and IPC (inter-process communication).Useful in virtual memory management, caching, and IPC (inter-process communication).

Conclusion

Although my initial task was to adapt the Inferno OS's mmap() system call, I ran into issues with installation and usability. I switched to Ubuntu Linux in order to finish the assignment and still achieve the learning objectives, and I was able to successfully implement and test mmap() there.

I now have a better grasp of memory mapping's operation and the reasons it is such a potent component of contemporary operating systems thanks to this procedure. By acting as a link between memory access and file I/O, mmap() provides speed and adaptability for sophisticated system-level programming.