4. Write a C/C++ program to create a file called file1 in blocking read-write mode and show how you can use fcntl api to modify its access control flags to non-blocking read-write mode.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   int accmode, val;

   // Check if the correct number of command-line arguments is provided
   if (argc != 2) {
      fprintf(stderr, "usage: %s <description>", argv[0]);
      exit(1);
   }

   // Use fcntl to get file access mode and file status flags
   val = fcntl(atoi(argv[1]), F_GETFL, 0);

   // Check if fcntl operation was successful
   if (val < 0) {
      perror("fcntl error for fd");
      exit(1);
   }

   // Extract file access mode using O_ACCMODE bitmask
   accmode = val & O_ACCMODE;

   // Print the file access mode
   if (accmode == O_RDONLY)
      printf("read only");
   else if (accmode == O_WRONLY)
      printf("Write only");
   else if (accmode == O_RDWR)
      printf("read write");
   else {
      fprintf(stderr, "unknown access mode");
      exit(1);
   }

   // Check and print additional file status flags
   if (val & O_APPEND)
      printf(", append");
```

```
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous write");

    putchar('\n');
    exit(0);
}
```

Note:
The fcntl (file control) function in C is used to perform various operations on open files or other file descriptors. It allows for the manipulation of file status flags, file descriptor duplication, file locking, and other file-related operations. The name "fcntl" stands for "file control." The function is typically used in Unix-like operating systems.

Syntax:
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */);

fd: The file descriptor on which the operation will be performed.
cmd: An integer representing the command or operation to be performed.
...: An optional argument, which depends on the specific operation (cmd). The argument may not be needed for some commands.
Common Commands (cmd values):
F_DUPFD:
Duplicates the file descriptor, creating a new file descriptor that refers to the same open file. The optional argument specifies the lowest acceptable file descriptor number for the new descriptor.
F_GETFL:
Gets the file access mode and file status flags.
F_SETFL:
Sets the file status flags.
F_SETLK, F_SETLKW, F_GETLK:
Operations for file locking. F_SETLK sets or clears a lock, F_SETLKW is similar but waits until the lock can be acquired, and F_GETLK retrieves information about a lock.
F_GETFD, F_SETFD:
Gets or sets the close-on-exec flag for the file descriptor.
F_GETOWN, F_SETOWN:
Gets or sets the process or process group ID that will receive signals when input is available or conditions change.
F_GETSIG, F_SETSIG:
Gets or sets the signal sent when input is available or conditions change.

File Descriptor:

Definition:

A file descriptor is a numeric identifier used by a process to access an open file, socket, or other I/O resource.
It is a small, non-negative integer that the operating system assigns when a file or resource is opened.

Difference between Inode number and File descriptor

- Inode numbers persist throughout the lifetime of a file, regardless of how many times the file is opened or closed.
- File descriptors are created when a file is opened and released when the file is closed.
- Inode numbers are used by the file system to manage file metadata.
- File descriptors are used by processes to perform I/O operations on open files or resources.

while both inode numbers and file descriptors are identifiers used in file systems, they serve different purposes and are used at different levels of the file access hierarchy. Inode numbers are associated with files or directories at the file system level, while file descriptors are used by processes to interact with open instances of files or resources.

Bitmask: A bitmask is a binary pattern that is used for bitwise operations, where each bit in the pattern represents a specific attribute or condition.

Example:
In the context of file access modes in Unix-like systems, O_ACCMODE is a bitmask that is used to extract the file access mode from the file status flags. The possible values for O_ACCMODE are:

O_RDONLY: Read-only mode (value 00 in octal).
O_WRONLY: Write-only mode (value 01 in octal).
O_RDWR: Read and write mode (value 02 in octal).
So, if a file is open in read-only mode, the value of O_ACCMODE will be O_RDONLY, and using the bitmask val & O_ACCMODE will result in O_RDONLY. In octal representation, O_RDONLY is 00, and performing the bitwise AND operation (val & O_ACCMODE) will preserve only the lower bits related to the access mode. The result will be the value 00 (read-only).

5.Write a C/C++ program to duplicate the file descriptor of a file Foo to standard input file descriptor.

```
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>

int main(int argc, char **argv) {
   int fd, nfd;

   // Check if the correct number of command-line arguments is provided
   if (argc < 2) {
      printf("usage: %s pathname\n", argv[0]);
      exit(1);
   }

   // Open the file specified in the command-line argument for writing
   if ((fd = open(argv[1], O_WRONLY)) < 0) {
      perror("Problem in opening the file");
      exit(1);
   }

   // Duplicate the file descriptor using fcntl with F_DUPFD
   if ((nfd = fcntl(fd, F_DUPFD, 0)) == -1) {
      perror("Problem in duplicating fd");
      exit(1);
   }

   // Print a message indicating that the file descriptor has been duplicated
   printf("Fd %d duplicated with %d\n", fd, nfd);

   // Close the original and duplicated file descriptors
   close(fd);
   close(nfd);

   return 0;
}
```

Note: The fcntl system call returns -1 on failure, and additional information can be retrieved from the errno variable.

Use Cases:
File Descriptor Management: Duplication, modification, and retrieval of file descriptors.
File Locking: Preventing multiple processes from accessing or modifying a file simultaneously.
Changing File Status Flags: Modifying the behavior of file descriptors, such as enabling non-blocking mode.

- File descriptor duplication is commonly used for various purposes, including redirection, parallel processing, and interprocess communication. When duplicating file descriptors, it's important to manage the closing of descriptors appropriately to avoid resource leaks and unexpected behavior.

6.Write a C/C++ program to query and display the different attributes associated with a file.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<time.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
   struct stat sb;  // Declare a structure 'sb' of type 'struct stat' to store file attributes.

   if(argc != 2)
   {
      fprintf(stderr,"usage: %s <pathname>\n", argv[0]);  // Print an error message if the program is not called with exactly one argument.
      exit(EXIT_FAILURE);  // Terminate the program with a failure status.
   }

   if(stat(argv[1], &sb) == -1)
   {
      perror("stat");  // Print an error message if the 'stat' system call fails.
      exit(EXIT_FAILURE);  // Terminate the program with a failure status.
   }

   printf("file type:          ");
   switch(sb.st_mode & S_IFMT)  // Determine and print the file type based on the mode bits using a switch statement.
   {
      case S_IFBLK: printf("block device file\n");  // If the file is a block device, print accordingly.
```

```
                break;
        case S_IFCHR: printf("character device file\n");  // If the file is a character device, print accordingly.
                break;
        case S_IFDIR: printf("directory\n");  // If the file is a directory, print accordingly.
                break;
        case S_IFIFO: printf("FIFO/pipe\n");  // If the file is a FIFO/pipe, print accordingly.
                break;
        case S_IFLNK: printf("symlink\n");  // If the file is a symbolic link, print accordingly.
                break;
        case S_IFREG: printf("regular file\n");  // If the file is a regular file, print accordingly.
                break;
        case S_IFSOCK: printf("socket\n");  // If the file is a socket, print accordingly.
                 break;
        default:       printf("regular file\n");  // If the file type is not recognized, assume it's a regular file.
                 break;
    }

    printf("Inode number:   %ld\n", (long) sb.st_ino);  // Print the inode number of the file.
    printf("Mode:    %lo(octal)\n", (unsigned long) sb.st_mode);  // Print the file mode in octal representation.
    printf("Blocks allocated:    %lld\n", (long long) sb.st_blocks);  // Print the number of blocks allocated to the file.

    exit(EXIT_SUCCESS);  // Terminate the program with a success status.
}
```

Note:Syntax

int stat(const char *pathname, struct stat *statbuf);

Stat -Data type :Structure

So, the Structure definition is:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* Inode number */
    mode_t    st_mode;    /* File type and mode */
    nlink_t   st_nlink;   /* Number of hard links */
    uid_t     st_uid;     /* User ID of owner */
    gid_t     st_gid;     /* Group ID of owner */
    dev_t     st_rdev;    /* Device ID (if special file) */
    off_t     st_size;    /* Total size, in bytes */
    blksize_t st_blksize; /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;  /* Number of 512B blocks allocated */
```

```
    time_t    st_atime;   /* Time of last access */
    time_t    st_mtime;   /* Time of last modification */
    time_t    st_ctime;   /* Time of last status change */
};
```

8.Write a C/C++ program to demonstrate masking of read/write/execute permission of a specified input file for user group and others category.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main() {
    mode_t oldMask, newMask;

    // Get the current file mode creation mask
    oldMask = umask((mode_t)0);
    printf("\n Old mask = %o\n", (int)oldMask);

    // Check and modify group permissions in the mask
    if (oldMask & S_IRGRP) {
        printf("\nChanging group read permission from Masked to unmasked.\n");
        oldMask = (oldMask ^ S_IRGRP);
    }

    // Set new mask with group write and execute permissions unmasked
    newMask = (oldMask | S_IWGRP | S_IXGRP);
    umask(newMask);
    printf("\nNew Mask = %o\n\n", (int)newMask);

    // Display the changes in the file mode creation mask
    printf("The file mode creation mask now specifies:\n");
    printf("  Group read permission  UNMASKED\n");
    printf("  Group write permission  MASKED\n");
    printf("  Group execute permission  MASKED\n");

    // Repeat the process for user permissions
    oldMask = umask((mode_t)0);
    printf("\n Old mask = %o\n", (int)oldMask);

    if (oldMask & S_IRUSR) {
        printf("\nChanging user read permission from Masked to unmasked.\n");
```

```
    oldMask = (oldMask ^ S_IRUSR);
  }

  newMask = (oldMask | S_IWUSR | S_IXUSR);
  umask(newMask);
  printf("\nNew Mask = %o\n\n", (int)newMask);

  printf("The file mode creation mask now specifies:\n");
  printf("  User read permission  UNMASKED\n");
  printf("  User write permission  MASKED\n");
  printf("  User execute permission  MASKED\n");

  // Repeat the process for other (world) permissions
  oldMask = umask((mode_t)0);
  printf("\n Old mask = %o\n", (int)oldMask);

  if (oldMask & S_IROTH) {
    printf("\nChanging Other read permission from Masked to unmasked.\n");
    oldMask = (oldMask ^ S_IROTH);
  }

  newMask = (oldMask | S_IWOTH | S_IXOTH);
  umask(newMask);
  printf("\nNew Mask = %o\n\n", (int)newMask);

  printf("The file mode creation mask now specifies:\n");
  printf("  Other read permission  UNMASKED\n");
  printf("  Other write permission  MASKED\n");
  printf("  Other execute permission  MASKED\n");

  return 0;
}
```

Note :

umask : System Call: At the system level, which interacts with the kernel.
When a program calls the umask function, it eventually leads to the execution of the umask system call to change the file mode creation mask for the process.
function in Unix-like operating systems, the term "mask" refers to a set of bits that determines the default permissions of newly created files. The umask function is used to set or retrieve this file mode creation mask.

The file mode creation mask is a 9-bit value that represents the default permissions of a new file. These 9 bits correspond to the permission bits for the file owner, group, and others. The mask is applied to the default permissions specified when creating a file to determine the final permissions.

Here's a breakdown of the 9 bits in the file mode creation mask:

Owner Permissions:

S_IRUSR (Read permission for owner)
S_IWUSR (Write permission for owner)
S_IXUSR (Execute permission for owner)
Group Permissions:

S_IRGRP (Read permission for group)
S_IWGRP (Write permission for group)
S_IXGRP (Execute permission for group)
Other (World) Permissions:

S_IROTH (Read permission for others)
S_IWOTH (Write permission for others)
S_IXOTH (Execute permission for others)