COEN146lab3

Monday, April 13, 2020 12:26 PM

Goal:

1. Copy a binary and text file using the **C function** and **system calls**.
2. Understand and run the UDP client-server application.
3. Modify the UDP client-server application such that the client sends a file (binary/text) to the server, and the server saves it.
4. Bonus (+10 points): Modify the a client-server application using TCP to transfer file.

Part 1:

1. Read and write text and binary files using C functions.
2. Read and write text and binary files using system calls.

Libraries needed for part 1:

#include<sys/types.h>
#include<sys/stat.h>
#include <fcntl.h>

What are the standard binary and text file extensions?

- Binary file: jpg, png, bmp, tiff etc.
- Text file: txt, html, xml, css, json etc.

Note:

- C Function connects the C code to file using I/O stream, while system call connects C code to file using file descriptor.
  - File descriptor is integer that uniquely identifies an open file of the process.
  - I/o stream sequence of bytes of data.
- Stream provide high level interface, while File descriptor provide a low-level interface.
- Streams are represented as FILE * object , while File descriptors are represented as objects of type int

C Functions to open and close a binary/text file.

- **fopen(): C Functions to open a binary/text file.**
  FILE *fopen(const char *file_name, const char *mode_of_operation);
  where:
  - file_name: file to open
  - mode_of_operation: refers to the mode of the file access
    - For example:- r: read , w: write , a: append etc
  - fopen() return a pointer to FILE if success, else NULL is returned

- **fclose(): C Functions to close a binary/text file.**
  fclose(FILE *file_name);
  Where:
  - file_name: file to close
  - fclose () function returns zero on success, or EOF if there is an error

C Functions to read and write a binary file.

- **fread(): C function to read binary file.**
  fread (void * ptr, size_t size, size_t count, FILE * stream);
  where:
  - ptr- it specifies the pointer to the block of memory with a size of at least (size*count) bytes to store the objects.
  - size - it specifies the size of each objects in bytes.
  - count: it specifies the number of elements, each one with a size of size bytes.
  - stream - This is the pointer to a FILE object that specifies an input stream.
  - Returns the number of items read

- **fwrite(): C function to write binary file.**
  fwrite (void *ptr, size_t size, size_t count, FILE *stream);
  - Returns number of items written

  *arguments of fwrite are similar to fread. Only difference is of read and write.

For example:

To open "lab3.jpg" file in read mode then function would be:

```
FILE* demo;                                  // demo is a pointer of type FILE
char buffer[100];                            // block of memory (ptr)
demo= fopen("lab3.jpg", "r");        // open lab3.jpg in read mode
fread(&buffer, sizeof(buffer), 1, demo);   // read 1 element of size = size of buffer (100)
fclose(demo);                                // close the file
```

C Functions to read and write the text file.

- **fscanf(): C function to read text file.**
  fscanf(FILE *ptr, const char *format, ...)
  - Reads formatted input from the stream.
  - Ptr: File from which data is read.
  - format: format of data read.
  - returns the number of input items successfully matched and assigned, zero if failure

- **fprintf(): C function to write a text file.**
  fprintf(FILE *ptr, const char *format, ...);
  *arguments similar to fscanf()

For example:

```
FILE *demo;                                  // demo is a pointer of type FILE
demo= FILE *fopen("lab3.txt", "r");          // open lab3.txt in read mode
/* Assuming that lab3.txt has content in below format
    CITY
    hyderbad
   */
char buf[100];                               // block of memory
fscanf(demo, "%s", buf);                     // to read a text file
fclose(demo);                                // close the file
```

***to read whole file use while loop**

System call to open, close , read and write a text/binary file.

- **Open(): System call to open a binary/text file.**
  open (const char* Path, int flags [, int mode]);
  - Path :- path to file
  - flags :- O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write, O_CREAT: create file if it doesn't exist, O_EXCL: prevent creation if it already exists
  - Open() system call return file descriptor used on success and -1 upon failure

- **Close(): System call to close a binary/text file.**
  close(int fd);
  - fd : file descriptor which uniquely identifies an open file of the process
  - Close () system call return 0 on success and -1 on error.

- **Read(): System call to read a binary/text file.**
  read (int fd, void* buf, size_t len);
  - Return:
    - return 0 on reaching end of file
    - return -1 on error
    - return -1 on signal interrupt
  - fd: file descriptor
  - buf: buffer to read data from
  - len: length of buffer

- **Write(): System call to write a binary/text file.**
  write (int fd, void* buf, size_t len);
  *arguments and return of write are similar to read. Only difference is of read and write.

For example:

```
int fd1 = open("demo.txt", O_RDONLY | O_CREAT);   //if file not in directory then file is created.
Close(fd1);
```

Part 2: UDP client-server application

UDP Server:

Step1: Declare socket file descriptor.
  int sockfd; // file descriptor of type int

Step2: Open UDP socket using socket() function.
  Socket(int domain, int type, int protocol)
  Where:
  - Domain: AF_INET for IPv4 and AF_INET6 for IPv6
  - Type: Type of socket to be created
    - SOCK_STREAM for TCP
    - SOCK_DGRAM for UDP
  - Protocol: Protocol to be used by socket.
    - 0 means use default protocol for the address family.
  - Return socket file descriptor and <0 on failure

So, if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
  {
    perror("Failure to setup an endpoint socket");
    exit(1);
  }

Step3: Declare and define server and client address.
  *Server has family(IPv4 or IPv6), port address, IP address
  struct sockaddr_in servAddr, clienAddr; // serAddr: server address, cleinAddr: client address

  servAddr.sin_family = AF_INET; // IPv4
  servAddr.sin_port = htons(5000); //Port 5000 is assigned to server
  *htons() makes sure numbers are stored in memory in network byte order, which is with the most significant byte (MSB) first

  servAddr.sin_addr.s_addr = INADDR_ANY; //Local IP address of any interface is assigned

Step4: bind socket with port number using bind() function.
  bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
  Where:
  sockfd – File descriptor of socket
  addr – Structure in which address to be binded to is specified
  addrlen – Size of *addr* structure

Step5: client receives message from server using recvfrom() function.
  recvfrom(int sockfd, void *buf, size_t len, int flag, struct sockaddr *clen_addr, socklen_t *addrlen)
  sockfd – File descriptor of socket
  buf – Application buffer in which to receive data
  len – Size of *buf* application buffer
  flags – Bitwise OR of flags to modify socket behavior (pass it as 0)
  clen_addr – Structure containing client address is returned
  addrlen – Variable in which size of clen_*addr* structure is returned

  while (1) // loop so that server can continuously receive data
    {
      int nr = recvfrom(sockfd, rbuf, 1024, 0, (struct sockaddr *)&clienAddr, &addrLen);
      rbuf[nr] = '\0';
      write(file, rbuf, nr);   //writes received data to destination file
    }

UDP client:

Step1: Declare socket file descriptor.
Step2: Open UDP socket using socket() function.
Step3: Declare and define server and client address.
  struct hostent *host; // pre-defined structure in library
  host = (struct hostent *)gethostbyname("localhost"); // Converts domain names into numerical IP addresses via DNS
  *gethostbyname(): retrieves host information corresponding to a host name (localhost) from a host database and returns a structure of type hostent

  servAddr.sin_family = AF_INET; // IPv4
  servAddr.sin_port = htons(5000); // server port number
  servAddr.sin_addr = *((struct in_addr *)host->h_addr); //

Step4: send data to server using sendto() function.

  sendto(int sockfd, const void *buf, size_t len, int flags,const struct sockaddr *dest_addr, socklen_t addrlen)
  - sockfd – File descriptor of socket
  - buf – Application buffer containing the data to be sent
  - len – Size of *buf* application buffer
  - flags – Bitwise OR of flags to modify socket behavior (pass it a 0)
  - dest_addr – Structure containing address of server
  - addrlen – Size of *dest_addr* structure

  while(1)
  {
  printf("Client: Type a message to send to Server\n");
  scanf("%s", sbuf);
  sendto(sockfd, sbuf, strlen(sbuf), 0, (struct sockaddr *)&servAddr, sizeof(struct sockaddr));
  }

Part 3:

Send a file from client to server using UDP.

Bonus: Send a file from client to server using TCP.