AUTHORS:

Anne Whitman   RUID: 042007629   netID: alh220

Michael Mazzola   RUID: 174006841   netID: mjm706

* *

Important Notes:

1. Our makefile can make all, make client, or make server. If you are only putting partial files on each machine, you can just make for one of the other by calling “make client” on the client side and “make server” on the server side. If you are going to have all files in the same directory, then you can just call “make”.

2. We did not make or turn in a netfileserver.h because the spec asked us to only turn in a netfileserver.c.

* *

NAME:

netfileserver.c; libnetfiles.c; netclient.c

SYNOPSIS:

./netfileserver ; ./netclient

DESCRIPTION:

netfileserver establishes a port and waits for connections. Once a connection has been made, it receives requests from the client computer to open, read, write, or close a file descriptor (using netopen(), netread(), netwrite(), and netclose(), respectively). If there is an error on the server side, the errno or h_errno is passed along to the client side.

netclient uses the libnetfiles.o (created from libnetfiles.c) library to access and use netopen(), netread(), netwrite(), and netclose() just like normal open(), read(), write(), and close() commands. By calling these commands, they receive back the same information they would from the regular commands.

Libnetfiles.c takes a client’s request to open, read, write, or close a file, opens a connection to the server, and passes along the command and other parameters the client passed in. When receiving a message back from the server, this program will then pass the info back to the client as if a regular open, read, write, or close had occurred. It will also set errno and h_errno where applicable.

FUNCTIONS:

netfileserver: Aside from main, this only has one function, the one that is called when a new thread is spawned:

clientService(void* args) – where void* args is a pointer to a sockfd.

/* This function parses through the message sent by the client, decodes what the message is asking for, executes the command, and then returns a message back to the client side. The messages come over in the format of <size>,<2-digit-command><command specific info>. Here are a few examples:

13,OP1./pathname => where 13 is the size after the comma, OP means open, 1 is the flag (O_RDONLY), and ./pathname is the pathname to open.

7,RD-6,10 => where 7 is the size after the first comma, RD means read, -6 is the file descriptor, and 10 is the number of bytes to read in from that file.

21,WR-6,well hello there => where 21 is the size after the first comma, WR is the command to write, -6 is the file descriptor, and "well hello there" is what is to be written into that file.

4,CL-6 => where 4 is the size after the first comma, CL is the command to close, and -6 is the file descriptor.

After parsing through the command to determine what the message is requesting, it then executes that command and returns a message to the client. The return message either begins with a Y (for success) or N (for failure). If it fails, then after the N will be a number that is used for tracking errno. If success, then here are examples of return messages based off of the examples above:

Open request:

Y-6 => where -6 is the file descriptor.

Read request:

Y10,here it is => where "here it is" is what was read from the file.

Write request:

Y21 => where 21 is how many bytes were written

Close request:

Y => there is no reason for more than Y because it was successful, there's no more information to give back.

After passing back the message, the thread closes the sockfd and then exits.*/

<u>Libnetfiles.c:</u> Along with the four required functions, this also has a function to open a socket.

```
int netopen(const char *pathname, int flags)
```

/*This function takes in a pathname and flags and attempts to open a file on a different machine. First, it checks to see if the connection between the machines has been initialized. If it has it will send a message to the other machine consisting of "OP" (to identify which function to execute), followed by either O_WRONLY, O_RDONLY or O_RDWR, and lastly the path name. The size of this is computed and concatenated to the front of the string followed by a comma. This message is sent and processed by the server and if it executes successfully the server returns "Y" followed by the negated file descriptor, otherwise the server returns "N" and the appropriate error. This function then either passes on the negated file descriptor the client or sets the errno and returns -1*/

```
ssize_t netread(int filedes, void* buf, size_t nbyte)
```

/*This function is given a file descriptor, a buffer, and a number representing how many bytes to read. It reads data from a file on a different machine. First, it checks if the connection has been initialized. If it has, it connects to a socket and creates a string to write to the server. The string consists of "RD" to indicate the function call, the file descriptor of the file to read from, and how many bytes to read. Also, the size of this is computed and concatenated to the beginning of the string followed by a comma. After the string is sent, the server reads it and attempts to perform what was sent and sends back a message. Following this, netread reads what was returned back from the server and if the returned message begins with "N" it indicates the process failed in some way and it will return an error number. If the first character is "Y" then the code was executed properly and the following characters up until a comma (used for a delimiter) represent the amount of bytes sent back. After the comma is the characters that were successfully read from the file and these characters are put copied into the buffer given to the function. The function then returns the amount of bytes read, or -1 on error.*/

```
ssize_t netwrite (int filedes, const void* buf, size_t nbyte)
```

```
    /*This function is given a file descriptor, a buffer, and a
    number representing how many bytes to write. It writes data to a
    file on a different machine. First, it checks if the connection
    has been initialized. If it has, it connects to a socket and
    creates a string to write to the server. The string consists of
    “WR” to indicate the function call, the file descriptor of the
    file to write to, how many bytes to write, and the string of what
    is to be written. Commas are added as delimiters. Also, the size
    of this is computed and concatenated to the beginning of the
    string followed by a comma.  After the string is sent, the server
    reads it and attempts to perform what was sent and sends back a
    message. Following this, netwrite reads what was returned back
    from the server and if the returned message begins with “N” it
    indicates the process failed in some way and it will return an
    error number. If the first character is “Y” then the code was
    executed properly and the following characters represent the
    amount of bytes sent back. After the comma is the characters that
    were successfully read from the file and these characters are put
    copied into the buffer given to the function. The function then
    returns the amount of bytes written or -1 on error.*/

int netclose (int fd)

    /*This function takes a file descriptor and attempts to close a
    previously opened file on a different machine. First, it checks
    to see if the connection is initialized. If it is it builds the
    instruction string to send to the other machine which consists of
    “CL” for the function call, and the file descriptor. The size of
    this is also concatenated to the beginning of the string followed
    by a comma. If the file is successfully closed it returns 0 and
    if there is an issue it returns -1 and sets the errno to the
    appropriate error.*/

int getSocket(struct sockaddr_in serverAI)

    /*This function takes a struct sockaddr_in and establishes a
    connection between the two machines. It fills in fields for the
    struct and returns a socket descriptor. */
```