

Week1.**1. Implement the following forms of IPC.****A) Pipes B) FIFO****Algorithm**

Step 1 : Store any message in one character array (`char *msg="Hello world"`)

Step 2 : Declare another character array

Step 3 : Create a pipe by using `pipe()` system call

Step 4 : Create another process by executing `fork()` system call

Step 5 : In parent process use system call `write()` to write message from one process to another process.

Step 6 : In child process display the message.

Week2.**Implement file transfer using Message Queue form of IPC****Algorithm**

Step 1 : Initialize character array with string.

Step 2 : Create a message Queue by using `msgget()` system call.

Step 3 : send a message with `msgsnd()` system call.

Step 4 : Receive the message by using `msgrcv()` system call.

Step 5 : Print the message.

Step 6 : kill the message queue using `msgctl()` system call.

Message queues are implemented as linked lists of data stored in shared memory. The message queue itself contains a series of data structures, one for each message, each of which identifies the address, type, and size of the message plus a pointer to the next message in the queue.

To allocate a queue, a program uses the `msgget()` system call. Messages are placed in the queue by `msgsnd()` system calls and retrieved by `msgrcv()` . Other operations related to managing a given message queue are performed by the `msgctl()` system call.

Week3.

Write a program to create an integer variable using shared memory concept and increment the variable

Simultaneously by two processes. Use semaphores to avoid race conditions.

Algorithm

Step 1 : Create a shared memory using shmget().

Step 2 : store integer value in shared memory. (shmat())

Step 3 : create a child process using fork().

Step 3 :get a semaphore on shared memory using semget().

Step 5 : increase the value of shared variable

Step 7 : release the semaphore

Step 8 : repeat step 4,5,6 in child process also.

Step 9 : remove shared memory.

Week4.

Design TCP iterative Client and server application to reverse the given input sentence.

Algorithm

1. Client sends message to server using send functions.
2. Server receives all the messages, server ignores all the consonants in the message.
3. All the vowels in the message are converted into upper case.
4. Server returns the entire message to clients (with toggled vowel cases).
5. For example: "This is a test and sample message." to server will be sent back to client as "ThIs Is A tEst And sAmPlE mEssAgE."

When client closes the connection server should close the communication with that client (socket). And once again wait for new clients to connect. Server program never exits.

Using fork function rewrite the programs, such that this server can handle multiple client connections at one time. To test this you need to run simultaneously multiple copies of client executions. Please log on server machine number of clients it is handled at **this** time.

*** server program ***

```
#include "unistd.h"
#include "errno.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netdb.h"

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "strings.h"
#include "sys/wait.h"

//Function Prototypes
void myabort(char *);

//Some Global Variables
int serverport = 3000;
char * eptr = NULL;
int listen_socket, client_socket;
struct sockaddr_in Server_Address, Client_Address;
int result,i;
socklen_t csize;
pid_t processid;
int childcount = 0;

//main()
int
main(int argc, char **argv){

char buf[100];
char tmp[100];
char * ptr;
int n, sent, length;

//Step 0: Process Command Line
if (argc > 2){
myabort("Usage: server ");
}
if (argc == 2){
serverport = (int) strtol(argv[1], &eptr, 10);
if (*eptr != '\0') myabort("Invalid Port Number!");
}

//Step 1: Create a socket
```

```
listen_socket = socket(PF_INET, SOCK_STREAM, 0);
if (listen_socket == -1) myabort("socket()");

//Step 2: Setup Address structure
bzero(&Server_Address, sizeof(Server_Address));
Server_Address.sin_family = AF_INET;
Server_Address.sin_port = htons(serverport);
Server_Address.sin_addr.s_addr = INADDR_ANY;

//Step 3: Bind the socket to the port
result = bind(listen_socket, (struct sockaddr *) &Server_Address, sizeof(Server_Address));
if (result == -1) myabort("bind()");

//Step 4: Listen to the socket
result = listen(listen_socket, 1);
if (result == -1) myabort("listen()");

printf("\nThe forkserver :%d\n", ntohs(Server_Address.sin_port));
fflush(stdout);
//Step 5: Setup an infinite loop to make connections
while(1){

//Accept a Connection
csize = sizeof(Client_Address);
client_socket = accept( listen_socket, (struct sockaddr *) &Client_Address, &csize);
if (client_socket == -1) myabort("accept()");

printf( "\nClient Accepted!\n" );

//fork this process into a child and parent
processid = fork();

//Check the return value given by fork(), if negative then error,
//if 0 then it is the child.
if ( processid == -1){
myabort("fork()");
}else if (processid == 0){
/*Child Process*/

close(listen_socket);
//loop until client closes
while (1){
```

```
//read string from client
bzero(&buf, sizeof(buf));
do{
bzero(&tmp, sizeof(tmp));
n = read(client_socket,(char *) &tmp, 100);
//cout << "server: " << tmp;
tmp[n] = '\0';
if (n == -1) myabort("read()");
if (n == 0) break;
strncat(buf, tmp, n-1);
buf[n-1] = ' ';
} while (tmp[n-1] != '\n');

buf[ strlen(buf) ] = '\n';

printf( "From client: %s",buf);

if (n == 0) break;


//write string back to client
sent = 0;
ptr = buf;
length = strlen(buf);

//the vowels in the message are converted into upper case.
for( i = 0; ptr[ i ]; i++)
{
if( ptr[i]=='a' || ptr[i]=='e' || ptr[i]=='i' || ptr[i]=='o' || ptr[i]=='u' )
ptr[ i ] = toupper( ptr[ i ] );
else
ptr[ i ] = ptr[ i ] ;
}

printf( "To client: %s",ptr);
while (sent < length ){
n = write(client_socket, ptr, strlen(ptr) );
if ( n == -1) myabort("write()");
sent += n;
ptr += n;
}
} //end inner while

close(client_socket);

//Child exits
exit(0);
```

```
}

//Parent Process

printf("\nChild process spawned with id number: %d",processid );
//increment the number of children processes
childcount++;
while(childcount){
processid = waitpid( (pid_t) - 1, NULL, WNOHANG );
if (processid < 0) myabort("waitpid()");
else if (processid == 0) break;
else childcount--;
}

}
close(listen_socket);

exit(0);

}

void myabort(char * msg){
printf("Error!: %s" , msg);
exit(1);
}
```

Week5.

Design TCP iterative Client and server application to reverse the given input sentence

/* client */

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "sys/socket.h"
#include "sys/types.h"
#include "netinet/in.h"
#include "strings.h"
#include "arpa/inet.h"

#define BUFFER 1024
```

```
main(int argc, char **argv)
{
    struct sockaddr_in serv;
    int sock;
    char in[BUFFER];
    char out[BUFFER];
    int len;

    if((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(-1);
    }

    serv.sin_family = AF_INET;
    serv.sin_port = htons(atoi(argv[2]));
    serv.sin_addr.s_addr = inet_addr(argv[1]);
    bzero(&serv.sin_zero, 8);

    printf("\nThe TCP client %d\n", ntohs(serv.sin_port));
    fflush(stdout);

    if((connect(sock, (struct sockaddr *)&serv,
        sizeof(struct sockaddr_in))) == -1)
    {
        perror("connect");
        exit(-1);
    }

    while(1)
    {
        printf("\nInput: ");

        fgets(in, BUFFER, stdin);
        send(sock, in, strlen(in), 0);

        len = recv(sock, out, BUFFER, 0);
        out[len] = '\0';
        printf("Output: %s\n", out);
    }

    close(sock);
}
```

Week6.**Design TCP client and server application to transfer file****Synopsis**

```
#include "msock.h"
```

```
int ServerSocket(u_short port,int max_servers);
int ClientSocket(char *netaddress,u_short port);
int sockGets(int sockfd,char *str,size_t count);
int sockRead(int sockfd,char *str,size_t count);
int sockPuts(int sockfd,char *str);
int sockWrite(int sockfd,char *str,size_t count);
int getPeerInfo(int sockfd,char *cli_host,char *cli_ip,u_short *cli_port);
int atoport(char *service,char *proto);
struct in_addr *atoaddr(char *address);
```

There are several routines available for SSL if compiled with -DUSE_SSL=1 flag. You will need [SSLey](#). SSLey is a free implementation of Netscape's Secure Socket Layer - the software encryption protocol behind the Netscape Secure Server and the Netscape Navigator Browser. The routines are:

```
int sockReadSSL(SSL *ssl,char *buf,size_t n);
int sockWriteSSL(SSL *ssl,char *str,size_t count);
int sockGetsSSL(SSL *ssl,char *str,size_t count);
int sockPutsSSL(SSL *ssl,char *str);
```

int ServerSocket(u_short port,int max_servers)

This function listens on a port and returns connections. The connection is returned as the socket file descriptor. The socket is of type SOCK_STREAM and AF_INET family. The function will create a new process for every incoming connections, so in the listening process, it will never return. Only when a connection comes in and a new process for it is created, the function will return. This means, the caller should never loop.

The parameters are as follows:

u_short port The port to listens to (host byte order)
int max_servers The maximum number of connection to queue up before having them rejected automatically.

The function returns the socked file descriptor (a positive number) on success and -1 on failure.

NOTE: _NEVER_ convert port to network byte order by calling htons(). It will be converted by the function. The options SO_REUSEADDR option is set on the socket file

descriptor. Read the Chapter 7 of Unix Network Programming (2nd Ed) by Stevens for details on the socket options.

int ClientSocket(char *netaddress,u_short port)

This function makes a connection to a given server's stream socket. The parameters are as follows:

char *netaddress The host to connect to. The netaddress can be the hostname or the IP address (dot separated 8 octets).

u_short port The port to connect to

The function returns the socked file descriptor (a positive number) on success and -1 on failure.

NOTE: `_NEVER_` convert port to network byte order by calling `htons()`. It will be converted by the function.

int sockGets(int sockfd,char *str,size_t count)

This function reads from a socket until it receives a line feed character. It fills the buffer "str" up to the maximum size "count".

The parameters are as follows:

int sockfd The socket to read from.

char *str The buffer to fill up.

size_t count The maximum number of bytes to stuff to str.

This function returns the number of bytes read from the socket or -1 if the connection is closed unexpectedly.

WARNING: If a single line exceeds the length of count, the data will be read and discarded.

int sockRead(int sockfd,char *str,size_t count)

This function reads the "count" number of bytes and stuffs to str. str must have enough space. This routine is like `read()` system call except that it makes sure all the requested number of bytes are read. Note, str must have enough space to hold the "count" number of bytes.

The function returns ≥ 0 if succeeds and -1 if there is any error.

int sockPuts(int sockfd,char *str)

This function writes a character string out to a socket.

The function returns number of bytes written to the socket or -1 if the connection is closed while it is trying to write.

int sockWrite(int sockfd,char *str,size_t count)

This function is like the `write()` system call, except that it will make sure all data is transmitted.

The function returns number of bytes written to the socket and -1 in case of any error.

int getPeerInfo(int sockfd,char *cli_host,char *cli_ip,u_short *cli_port)

This function gets information about the host connected to the socket. The parameters are as follows:

int sockfd The connected socket

char *cli_host The hostname connected to the socket (returns)

char *cli_ip The IP address of the host connected (returns)

u_short *cli_port The client side port of the host (returns)

Note, cli_host, cli_ip must have enough space to hold the info.

int atoport(char *service, char *proto)

This function takes a service name and a service type and returns a port number. If the service name is not found, it tries it as a decimal number. The number returned is byte ordered for the network.

The function returns a positive number if succeeds and -1 if fails.

struct in_addr *atoaddr(char *address)

This function converts ASCII text to **in_addr** struct. NULL is returned if the address can not be found.

Compile

```
$ gunzip <
$ cd libmsock
$ ./configure
$ make
```

If make succeeds the library libmsock.a will be created. To install it copy **libmsock.a** to /usr/local/lib and **msock.h** in /usr/local/include. Go to the examples and its sub-directories, type make to compile an example.

Week7.

Design a TCP concurrent server to convert a given text into upper case using multiplexing system call "select"

```
#include "unp.h"
```

```
Int main(int argc, char **argv)
{

int i, maxi, maxfd, listenfd, connfd, sockfd;
int ready, client[FD_SETSIZE];
ssize_t n;
fd_set rset, allset;
char line[MAXLINE];
socklen_t clilen;
struct sockaddr_in cliaddr, servaddr;

listenfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

maxfd = listenfd; /* initialize */
maxi = -1; /* index into client[] array */
for (i = 0; i <>
    client[i] = -1; /* -1 indicates available entry */
    FD_ZERO(&allset);
    FD_SET(listenfd, &allset);
/* end fig01 */

/* include fig02 */
for ( ; ; ) {
    rset = allset; /* structure assignment */
    nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
    if (FD_ISSET(listenfd, &rset))
    {
        /* new client connection */
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
#ifdef NOTDEF
        printf("new client: %s, port %d\n",
            Inet_ntop(AF_INET, &cliaddr.sin_addr, 4, NULL),
            ntohs(cliaddr.sin_port));
#endif
        for (i = 0; i <>
            if (client[i] <>
                client[i] = connfd; /* save descriptor */
                break;
        }
        if (i == FD_SETSIZE)
            err_quit("too many clients");
        FD_SET(connfd, &allset);

        /* add new descriptor to set */
        if (connfd > maxfd)
            maxfd = connfd; /* for select */
        if (i > maxi)
```

```

    maxi = i;    /* max index in client[] array */

    if (--nready <= 0)
        continue;    /* no more readable descriptors */
}

for (i = 0; i <= maxi; i++) {    /* check all clients for data */
    if ( (sockfd = client[i]) <>
        continue;
    if (FD_ISSET(sockfd, &rset)) {
        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)

    {
        /*connection closed by client */
        Close(sockfd);
        FD_CLR(sockfd, &allset);
        client[i] = -1;
    } else
        Writen(sockfd, line, n);
    if (--nready <= 0)
        break;    /* no more readable descriptors */
    }
}
}

```

Week8.

Design a TCP concurrent server to echo given set of sentences using poll functions

```

#include "unp.h"
#include "limits.h"    /* for OPEN_MAX */

int main(int argc, char **argv)
{
    int i, maxi, listenfd, connfd, sockfd;
    int nready;
    ssize_t n;
    char line[MAXLINE];
    socklen_t clien;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    client[0].fd = listenfd;
    client[0].events = POLLRDNORM;
    for (i = 1; i <>
        client[i].fd = -1;    /* -1 indicates available entry */

```

```

        maxi = 0;                                /* max index into client[]
array */
/* end fig01 */

/* include fig02 */
    for ( ; ; ) {
        nready = Poll(client, maxi+1, INFTIM);

        if (client[0].revents & POLLRDNORM) { /* new client connection */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
#ifdef NOTDEF
            printf("new client: %s\n", Sock_ntop((SA *) &cliaddr,
clilen));
#endif

            for (i = 1; i <>
                if (client[i].fd <>
                    client[i].fd = connfd; /* save descriptor
*/
                    break;
                }
            if (i == OPEN_MAX)
                err_quit("too many clients");

            client[i].events = POLLRDNORM;
            if (i > maxi)
                maxi = i;                                /* max
index in client[] array */

            if (--nready <= 0)
                continue;                                /* no more
readable descriptors */
        }

        for (i = 1; i <= maxi; i++) { /* check all clients for data */
            if ( (sockfd = client[i].fd) <>
                continue;
            if (client[i].revents & (POLLRDNORM | POLLERR)) {
                if ( (n = readline(sockfd, line, MAXLINE)) <>
                    if (errno == ECONNRESET) {
                        /*4connection reset by
client */
#ifdef NOTDEF
                        printf("client[%d] aborted
connection\n", i);
#endif
                        Close(sockfd);
                        client[i].fd = -1;
                    } else
                        err_sys("readline error");
                } else if (n == 0) {
                    /*4connection closed by client */
#ifdef NOTDEF
                    printf("client[%d] closed connection\n",
i);
#endif
                    Close(sockfd);
                    client[i].fd = -1;
                } else

```

```
        Writen(sockfd, line, n);

        if (--nready <= 0)
            break;                                /* no more
readable descriptors */
    }
}
}
```

Week9.

Design UDP Client and server application to reverse the given input sentence

Server

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "stdio.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"

#define BUFLen 512
#define NPACK 10
#define PORT 9930

void diep(char *s)
{
    perror(s);
    exit(1);
}

int main(void)
{
    struct sockaddr_in si_me, si_other;
    int s, i, slen=sizeof(si_other);
    char buf[BUFLen];

    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))== -1)
        diep("socket");

    memset((char *) &si_me, 0, sizeof(si_me));
    si_me.sin_family = AF_INET;
    si_me.sin_port = htons(PORT);
    si_me.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, &si_me, sizeof(si_me))== -1)
        diep("bind");

    for (i=0; i
        if (recvfrom(s, buf, BUFLen, 0, &si_other, &slen)== -1)
            diep("recvfrom()");
        printf("Received packet from %s:%d\nData: %s\n\n",
            inet_ntoa(si_other.sin_addr), ntohs(si_other.sin_port), buf);
    }
```

```
    close(s);  
    return 0;  
}
```

Week10

Design UDP Client server to transfer a file

The server

```
#include "sys/types.h"  
  
#include "sys/socket.h"  
  
#include "unistd.h"  
  
#include "netinet/in.h"  
  
#include "stdio.h"  
  
#include "error.h"  
  
struct request {  
    float oper1, oper2;  
    char op;  
};  
  
struct response {  
    float resval;  
  
    int status;  
};  
  
int main()  
{
```

```
char buf[5];

struct request req;

struct response res;

int lisfd,sesfd,ser_addlen,cli_addlen,ret;

struct sockaddr_in server_address,client_address;

sesfd=socket(AF_INET,SOCK_DGRAM,0);

server_address.sin_family=AF_INET;

server_address.sin_port=4700;

server_address.sin_addr.s_addr=inet_addr("127.0.0.1");

ser_addlen=sizeof(server_address);

cli_addlen=sizeof(client_address);

bind(sesfd,(struct sockaddr*)&server_address,ser_addlen);

printf("Server executed listen...\n");

while(1)

{

ret=recvfrom(sesfd,&req,sizeof(req),0,(struct sockaddr *)&client_address,&cli_addlen);

printf("after recvfrom...%d\n",ret);

perror("Recverror:");

switch(req.op)

{

case '+':

res.resval=req.oper1+req.oper2;

res.status=1;

break;

case '-':
```

```
res.resval=req.oper1-req.oper2;

res.status=1;

break;

case '*':

res.resval=req.oper1*req.oper2;

res.status=1;

break;

case '/':

res.resval=req.oper1/req.oper2;

res.status=1;

break;

default:

res.resval=-999;

res.status=0;

}

ret=sendto(sesfd,&res,sizeof(res),0,(struct sockaddr*)&client_address,cli_addlen);

printf("After sendto....%d\n",ret);

}

close(sesfd);

return(0);

}
```

The client

```
#include "sys/types.h"

#include "sys/socket.h"

#include "unistd.h"
```

```
#include "netinet/in.h"

#include "stdio.h"

struct request {

float oper1,oper2;

char op;

};

struct response {

float resval;

int status;

};

int main()

{

char buf[5];

struct request req;

struct response res;

int sesfd,ser_addlen,cli_addlen,ret;

struct sockaddr_in server_address,client_address;

sesfd=socket(AF_INET,SOCK_DGRAM,0);

server_address.sin_family=AF_INET;

server_address.sin_port=4700;

server_address.sin_addr.s_addr=inet_addr("127.0.0.1");

ser_addlen=sizeof(server_address);

printf("Operator is:");

scanf("%c",&req.op);

printf("Operands are..");
```

```
scanf("%f %f",&req.oper1,&req.oper2);

sendto(sesfd,&req,sizeof(req),0,(struct sockaddr *)&server_address,ser_addlen);

printf("Before recvfrom..\n");

recvfrom(sesfd,&res,sizeof(res),0,(struct sockaddr *)&server_address,&ser_addlen);

printf("Status is %d\n",res.status);

printf("ResVal is %f\n",res.resval);

close(sesfd);

return(0);

}
```

The client

```
#define SRV_IP "999.999.999.999"

/* diep(), #includes and #defines like in the server */

int main(void)

{

    struct sockaddr_in si_other;

    int s, i, slen=sizeof(si_other);
```

```
char buf[BUFLEN];
```

```
if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
```

```
    diep("socket");
```

```
memset((char *) &si_other, 0, sizeof(si_other));
```

```
si_other.sin_family = AF_INET;
```

```
si_other.sin_port = htons(PORT);
```

```
if (inet_aton(SRV_IP, &si_other.sin_addr)==0) {
```

```
    fprintf(stderr, "inet_aton() failed\n");
```

```
    exit(1);
```

```
}
```

```
for (i=0; i
```

```
    printf("Sending packet %d\n", i);
```

```
    sprintf(buf, "This is packet %d\n", i);
```

```
    if (sendto(s, buf, BUFLen, 0, &si_other, slen)==-1)
```

```
    diep("sendto()");  
  
}
```

```
close(s);  
  
return 0;  
  
}
```

Week12

Design a RPC application to add and subtract a given pair of integers

rpc server

```
/*  
  
* This is sample code generated by rpcgen.  
  
* These are only templates and you can use them  
  
* as a guideline for developing your own functions.  
  
*/
```

```
#include "stdio.h"

#include "errno.h"

#include "string.h"

#include "signal.h"

#include "rpcps.h"

int isnewline(char c);

#define isnewline(c) ((c) == '\n' || (c) == '\r' || (c) == '\f')

void chlldie(int sig)

{

wait(0);

}

int ConvertString(char *String)

{ int cols=0;

char *srch, *done;

srch = done = String;

while ( *srch != 0 )

{

if ( isnewline(*srch) )

{

cols++;

*done++ = 0;

break;

}

else if ( srch[0] == ' ' && srch[1] == ' ' )

{
```

```
cols++;

while ( *srch == ' ' )

srch++;

*done++ = 0;

}

else

*done++ = *srch++;

}

*done = 0;

return cols;

}

TableResult *

getps_1_svc(char **argp, struct svc_req *rqstp)

{ int i,j, cols=10;

FILE *fp;

TRow row;

TTable table;

char TempS[1024], *str;

static TableResult result;

signal(SIGCHLD, chlddie);

strncpy(TempS, "ps ", sizeof(TempS));

strncat(TempS, *argp, sizeof(TempS));

if ( (fp = popen(TempS, "r")) != NULL )

{

xdr_free((xdrproc_t)xdr_TableResult, (void*)&result);
```

```
table = result.TableResult_u.Table = calloc(1, sizeof(TTable*));

while ( fgets(TempS, sizeof(TempS), fp) != NULL )

{

row = table->Row = calloc(1, sizeof(TRow*));

cols = ConvertString(TempS);

str = TempS;

for ( j = 0; j <

{

row->Field = strdup(str);

while ( *str++ != 0 );

if ( j <

{

row->Next = calloc(1, sizeof(TRow*));

row = row->Next;

}

}

if ( !feof(fp) )

{

table->Next = calloc(1, sizeof(TTable*));

table = table->Next;

}

}

result.Err = 0;

return &result;

}
```

```
result.Err = errno;  
return &result;  
}
```

NMP LAB
