```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: January 1, 2002
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Distributed Operating Systems (2301462) classnote.
 * Description: This sample server module illustrates remote host
 *           communication over standard TCP/IP connection.
 * History of Modification:
 *    Date: May 16, 2002 by Peraphon Sophatsathit
 *    This program, along with a directory search routine developed
 *    separately as an independent module, demonstrates some practical
 *    aspects of data exchange over TCP/IP socket as part of the
 *    Data Communcation I class (2301369).  The use of this code is
 *    free, provided that this information is kept intact.
 * Invocation:
 *    The server module must be invoked first to initiate the connection
 *    with default option, i.e.,
 *           $ server  0
 *    the client module is subsequently invoked to establish the
 *    communication in a 'simplex chat' manner, that is, a one-way
 *    talk session from client to server as follows:
 *           $ client  161.200.126.10
 * Input:
 *    at the prompt 'client>' from client side, two commands can be issued
 *    as follows:
 *           client>#get   .            (to download from server to client)
 *    or     client>#put   .            (to upload from client to server)
 * Features:
 *    add directory retrieval module to incorporate down/upload capabilities.
 *    from client to server.  As such, the main 'str_echo' routine underwent
 *    considerable change to accommodate these new features.  In addition,
 *    an adhoc exchange discipline was improvised to cope with some socket's
 *    idiosyncracies that hadn't been unraveled by the author.
 * Transmission exchange framework:
 *    1. set up read/write pair to ensure proper handshake (see download
 *       related values)
 *    2. use 'char *' buffer to transmit byte information over the socket
 *       rather than the usual read/write system calls syntax
 * Caveat:
 *    Different OSes use different 'BUFSIZ' which, in practice, results in
 *    unsynchronized byte exchange, hence the use of 'answer back' to keep
 *    both sides in sync.
 */
```

```c
#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>
#include     <errno.h>

/*
 * porting from BSD to SVR4
 */
#ifdef __USE_BSD
#  include   <machine/param.h>
#endif

#include     <sys/types.h>
#include     <sys/socket.h>
#include     <netinet/in.h>
#include     <signal.h>
#include     <unistd.h>
#include     <time.h>

/*
 * prototypes
 */
unsigned int alarm(unsigned int);
void         (*signal(int, void (*disp)(int))) (int);
void         handler(int);
void         t_out(int);
void         bzero(void *, size_t);
int          socket(int, int, int);
int          bind(int, const struct sockaddr *, socklen_t);
int          listen(int, int);
int          accept(int, struct sockaddr *, socklen_t *);
int          snprintf(char *, size_t, const char *, ...);
char         *strtok_r(char *, const char *, char **);

int          proc_loop(int);
int          driver(int, int, int, char *);
int          str_echo(FILE *, int);
int          parse_ln(char *, char *);
void         clear_buff(char *, int);
```

```
int             pdir(char *, int);
int             check_dw(char *, int);

/*
 * download related values
 */
#define             LISTENQ      1024
#define             SERV_PORT    9877
#define             Sep          " \t"
#define             Sep2         "()"
#define             Gets         "#get"
#define             Puts         "#put"
#define             Ready_put    "Rput"
#define             End_dw       "#End#"
#define             End_fl       "eof"
#define             Get_token    1000
#define             Put_token    1001
#define             Ready_token  1002
#define             Happy_open   "pass open"
#define             Rcv_mkdir    "received mkdir"
#define             Dw_done      "download completed!\n"
#define             Transfer_limit(1000)
#ifdef SURE
#define             Transfer_limit(BUFSIZ - 2)
#endif

/*
 * error return code
 */
#define             Normal       0
#define             Err_socket   1
#define             Err_bind     2
#define             Err_listen   3
#define             Err_accept   4
#define             Err_connect  5
#define             Err_write    6
#define             Err_read     7
#define             Err_IP       10
#define             Err_fork     88
#define             Err_usage    99
#define             DW_ok        887
#define             DW_failed    888
#define             DW_done      889
```

```
/*
 * globals and macros
 */
#define             EQ(a, b)     (strcmp(a, b) == 0)
#define             NE(a, b)     (strcmp(a, b) != 0)
#define             Small        20
#define             TRUE         1
#define             FALSE        0
#define             Null_char    '\0'

#define             rmode        "r"
#define             wmode        "w"
#define             Cur_dot      '.'
#define             Slash        '/'
#define             Dots         "."
#define             Dotts        ".."
#define             cmd1         "fopen"
#define             cmd3         "mkdir"
#define             Nul_str      ""

char         *desc[]      =
{
      "Description: Wait 0 second for timeout which is recommended.",
      "Any other values can be used as a precaution to prevent the",
      "process from running away, but will cause an abnormal",
      "termination.  However, too long a wait will have no effect",
      "if the process has already terminated.\n",
      ""
};

int          flag = FALSE;
FILE         *ffp;

/*
 * The purpose of signal calls employed in this program is to prevent
 * runaway processing.  The user may terminate (kill) the process any time
 * via user command (ctrl C) or timer.  The latter can be set to any
 * positive integer ranging from 0 to N (N is recommneded to be small to
 * have any effect).  All signals may appear to have no effect if control
 * is suspended by '(blocking) read'.  In which case, one must send a
 * message by typing from keyboard to get out of 'read' wait.
 * Note that in order for the signals to have an immediate effect,
 * non-blocking read must be set along with extra precaution to handle
 * any 'non-blocking' timing and synchronization idiosyncracies.
 */
```

```
int
main(int ac, char **av)
{
	int		rt_code, i;
	unsigned int	sec;

	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	signal(SIGALRM, t_out);

	switch (ac)
	{
		case 2:
			sec = atoi(av[1]);
			if (sec > 0)
				alarm(sec);
			break;
		default:
			printf("\nUsage: %s   wait_sec\n\n", av[0]);
			printf("Example: %s   0  (no timer is set)\n", av[0]);
			printf("Example: %s   3  (3 seconds timeout)\n\n", av[0]);
			for (i = 0; NE(desc[i], Nul_str); i++)
				printf("%s\n", desc[i]);
			return Err_usage;
	}
	rt_code = proc_loop((int)sec);
	if (rt_code > Normal || flag == TRUE)
		printf("Abnornal termination of RPC loop\n");
	fflush(stdout);
	fflush(stderr);
	return Normal;
}

/*
 * set up standard TCP/IP connection
 */
int
proc_loop(int num)
{
	int			counter = 0;
	int			listenfd, connfd;
	char			buf[BUFSIZ];
	time_t			ticks;
	socklen_t		clilen;
	struct sockaddr_in	servaddr, cliaddr;
```

```
        /*
         * open a socket to accept incoming request from client(s)
         */
        if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        {
                return Err_socket;
        }
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family  = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port    = htons(SERV_PORT);

        if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
        {
                return Err_bind;
        }
        if (listen(listenfd, LISTENQ) < 0)
        {
                return Err_listen;
        }
        clilen = sizeof(cliaddr);
        if ((connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen)) < 0)
        {
                return Err_accept;
        }
        ticks  = time(NULL);
        snprintf(buf, sizeof(buf), "%.24s\n", ctime(&ticks));
        if (driver(counter, listenfd, connfd, buf) > 0)
        {
                perror("fork and exec failed");
        }
        close(connfd);
        return Normal;
}

/*
 * The driver function spawns a child process to start a TCP socket to
 * communicate with the client process.
 */
int
driver(int count, int listenfd, int connfd, char *sm)
{
        int     pid = 0;
        int     rt  = 0;
	pid = fork();
	if (pid == 0)
	{
		printf("begin child process\n");
		close(listenfd);
		rt = str_echo(stdin, connfd);
	}
	else if (pid > 0)
	{
		printf("parent: spawn succeeded!\n");
	}
	else
	{
		printf("fork failed: parent exiting...\n");
		rt = Err_fork;
	}
	return rt;
}

/*
 * read loop: first send prompt string to client and enter read/receive
 * message loop.  The process terminates when ctrl-D is received or
 * interrupts from pending signals.
 *
 * Change notes:
 * 'check_dw', etc., were added to intercept the input from client end.
 */
int
str_echo(FILE *fp, int sockfd)
{
	int	n, cflg;
	char	sline[BUFSIZ], rline[BUFSIZ];
	char	fn[BUFSIZ];

	strcpy(sline, "from server: begin typing message, ctrl-D to quit\n");
	n = strlen(sline);
	write(sockfd, sline, n);
	for (; flag == FALSE; )
	{
		clear_buff(rline, BUFSIZ);
		clear_buff(sline, BUFSIZ);
		if ((n = read(sockfd, rline, BUFSIZ)) == 0)
		{
			printf("connection closed by other end\n");
			break;
		}
```

```
                /*
                 * look for file transfer command from the incoming byte streams
                 */
                if ((cflg = check_dw(rline, sockfd)) == DW_ok)
                        continue;
                else if (cflg == DW_failed)
                {
                        printf("download failed\n");
                        break;
                }
                /*
                 * normal (chat) message and download commands from keyboard
                 */
                if ((cflg = parse_ln(rline, fn)) == Normal)
                {
                        write(sockfd, rline, n);
                        fprintf(stdout, "echo> %s", rline);
                }
                else
                {
                        /*
                         * client 'get' calls the module from this end.
                         */
                        if (cflg == Get_token || cflg == Ready_token)
                                pdir(fn, sockfd);
                        else
                        {
                                /*
                                 * client 'put' calls the module
                                 * at the other end (client side).
                                 */
                                sprintf(rline, "%s (%s) ", Ready_put, fn);
                                n = strlen(rline);
                                write(sockfd, rline, n);
                        }
                }
        }
        return Normal;
}

/*
 * check if input contains file transfer commands, i.e.,
 * 'fopen' and 'mkdir'
 * Note: strtok_r creates side-effect on original string, hence the use of 'tmp'
 */
```

```
int
check_dw(char *ln, int sockfd)
{
	int		num;
	char		size;
	char		*t, *q, *brk;
	char		tmp[BUFSIZ], answerb[Small];
	char		buf[BUFSIZ];
	unsigned char	buf2[BUFSIZ];
	int		n, rt;

	strcpy(tmp, ln);
	t = strtok_r(tmp, Sep, &brk);
	if (EQ(t, cmd1) || EQ(t, cmd3) || EQ(t, End_dw))
	{
		rt = DW_ok;
		/*
		 * open a new local file
		 */
		if (EQ(t, cmd1))
		{
			q = strtok_r(NULL, Sep2, &brk);
			strcpy(buf, q);
			printf("receiving file %s...", buf);
			if ((ffp = fopen(buf, wmode)) == NULL)
				return DW_failed;
			/*
			 * write something just to keep remote open happy
			 */
			strcpy(tmp, Happy_open);
			n = strlen(tmp);
			write(sockfd, tmp, n);
			/*
			 * file size is greater than zero (=FALSE)
			 */
			(void)read(sockfd, &size, 1);
			n = size - 48;
			if (n == FALSE)
			{
				/*
				 * make sure the buffer is cleared.  Any left
				 * over characters will result in error.
				 */
				clear_buff(tmp, BUFSIZ);
				num = Transfer_limit;
```

```
                /*
                 * transmission rate is too fast, hence the
                 * 'answer back' pause.
                 */
                while ((n = read(sockfd, (char *)buf2, num)) == num)
                {
                        fwrite((void *)buf2, (size_t)n, 1, ffp);
                        fflush(ffp);
                        answerb[0] = Null_char;
                        write(sockfd, answerb, 1);
                }
                if (n > 0)
                {
                        fwrite((void *)buf2, (size_t)n, 1, ffp);
                        fflush(ffp);
                }
        }
        fclose(ffp);
        /*
         * signal end of (writing) target file
         */
        n = strlen(End_fl);
        write(sockfd, End_fl, n);
        printf("done\n");
}
else if (EQ(t, cmd3))
{
        /*
         * write something just to keep remote 'snd_dir' happy
         */
        q = strtok_r(NULL, Sep2, &brk);
        strcpy(buf, Rcv_mkdir);
        n = strlen(buf);
        write(sockfd, buf, n);
        clear_buff(buf, BUFSIZ);
        sprintf(buf, "%s %s", t, q);
        /*
         * check directory existence
         */
        if (system(buf) != 0)
        {
                sprintf(buf, "test -d %s", q);
                if (system(buf) != 0)
                        return DW_failed;
        }
}
```

```
		/*
		 * end download string
		 */
		else
		{
			strcpy(tmp, Dw_done);
			n = strlen(tmp);
			write(sockfd, tmp, n);
		}
	}
	else
	{
		rt = Normal;
	}
	return rt;
}

/*
 * parse input to locate download command
 * Note: strtok_r creates side-effect on original string, hence the use of 'tmp'
 */
int
parse_ln(char *ln, char *fn)
{
	int	rt;
	char	*t, *brk;
	char	tmp[BUFSIZ];

	strcpy(tmp, ln);
	t = strtok_r(tmp, Sep, &brk);
	if (EQ(t, Gets) || EQ(t, Puts) || EQ(t, Ready_put))
	{
		if (EQ(t, Gets))
			rt = Get_token;
		else if (EQ(t, Puts))
			rt = Put_token;
		else
			rt = Ready_token;
		t = strtok_r(NULL, Sep2, &brk);
		strcpy(fn, t);
		return rt;
	}
	return Normal;
}
```

```
/*
 * clear R/W buffer to null
 */
void
clear_buff(char *line, int n)
{
	register int i;

	for (i = 0; i < n; i++)
		line[i] = Null_char;
	return;
}

/*
 * time out by alarm clock
 */
void
t_out(int sig)
{
	signal(SIGALRM, t_out);
	flag = TRUE;
	printf("timeout by ALARM signal\n");
	return;
}

/*
 * interrupt and kill signals
 */
void
handler(int sig)
{
	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	flag = TRUE;
	printf("receiving INT/QUIT signal\n");
	return;
}
```

```c
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: May 15, 2002
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Written for use as a sample classnote.
 * Description: This sample directory retrieval module accesses and
 *             retrieves files and sub-dir in the specified directory
 *             within the user authorization.  When coupled with
 *             the network communication modules, it serves as a
 *             file download utility.
 * History of change:
 *     Date: May 20, 2002 by Peraphon Sophatsathit
 *     To make it callable from the above 'semi-chat' programs, the
 *     'main' module was changed to 'pdir' to accommodate directory
 *     search and retrieval.
 *
 *     Date: June 3, 2002 by Peraphon Sophatsathit
 *     It was discovered that all bytes transmitted over socket should
 *     be made 'unsigned char' rather than 'char' to cope with 8-bit
 *     byte pattern, hence the use of 'buf2' as an adhoc fix.
 */

#include       <dirent.h>

#ifdef __USE_BSD
#include       <sys/param.h>
extern long    int    telldir(DIR *);
#else
#define        __USE_BSD
#include       <sys/stat.h>
#endif

/*
 * system functions
 */
DIR            *opendir(const char *);
struct dirent  *readdir(DIR *);
int            closedir(DIR *);
int            lstat(const char *, struct stat *);
```

```
/*
 * unused system functions, but might come in handy later.
 */
#ifdef FUTURE_USED
int             chdir(const char *);
int             fchdir(int);
void            seekdir(DIR *, long);
void            rewinddir(DIR *);
#endif

/*
 * prototypes
 */
int             dir_loop(char *, int);
int             check_dir(char *);
char            *stripping(char *);
int             snd_file(char *, int, int);
int             snd_dir(char *, int);
int             loc_open(char *, int);
int             wrong_name(char *);

/*
 * error conditions
 */
#define                 Illegal         1
#define                 U_defn          2
#define                 File_error      20
#define                 Errors          99

/*
 * main driver to retrieve directory information
 */
int
pdir(char *av, int sockfd)
{
        int     rt_code, pos, n;
        char    *cur_name, *p;
        char    dir_ptr[BUFSIZ];
        char    tmp[BUFSIZ];

        p = av;
        n = strlen(p);
        p[n-1] = Null_char;
        cur_name = stripping(p);
        if (*cur_name == Null_char)
        {
                strcpy(dir_ptr, Dots);
        }
        else if (NE(cur_name, p))
                strcpy(dir_ptr, cur_name);
        else
                strcpy(dir_ptr, p);
        pos = check_dir(dir_ptr);
        if (pos == Normal)
        {
                rt_code = dir_loop(dir_ptr, sockfd);
                /*
                 * send end_download string to the other end
                 */
                strcpy(tmp, End_dw);
                n = strlen(tmp);
                write(sockfd, tmp, n);
        }
        else
        {
                rt_code = Errors;
                printf("%s directory path is not permitted\n", dir_ptr);
        }
        return rt_code;
}

/*
 * recursively traverses the directory tree
 * Since modern UNIX permits embeded blanks in file name, special
 * 'delimiters' were employed to avoid conventional 'whitespace'
 * delimiters, e.g., tab, blank, and newline.  The 'parenthesis'
 * was chosen to internally delimit the entire file name.  As such,
 * any source file having 'parenthesis' pairs will not be transmitted
 * to avoid conflict (see 'wrong_name' block).
 */
int
dir_loop(char *name, int sockfd)
{
        struct stat   st;
        struct dirent *diry;
        DIR           *dp;
        uid_t         uid, euid;
        gid_t         gid;
        mode_t        md;
        int           rt = Normal;
        int           cnt, fsiz;
        char          fname[BUFSIZ];
        char          tmp[BUFSIZ];
```

```
/*
 * open directory
 */
if ((dp = opendir(name)) == NULL)
{
      rt = U_defn;
      perror(name);
      return rt;
}
while ((diry = readdir(dp)) != NULL)
{
      if (EQ(diry->d_name, Dots) || EQ(diry->d_name, Dotts))
            continue;
      else if (wrong_name(diry->d_name) == TRUE)
      {
            printf("File name containing illegal characters <%s>, ", diry->d_name);
            printf("skip to next entry\n");
            continue;
      }
      sprintf(fname, "%s/%s", name, diry->d_name);
      if (lstat(fname, &st) < 0)
      {
            perror(fname);
            rt = Illegal;
            break;
      }
      else
      {
            uid  = getuid();
            euid = geteuid();
            gid  = getgid();
            fsiz = st.st_size == 0 ? TRUE : FALSE;
            if ((st.st_uid == uid || st.st_uid == euid) && st.st_gid == gid)
            {
                  md = st.st_mode & S_IFMT;
                  if (md == S_IFREG || md == S_IFDIR)
                  {
                        if (md == S_IFDIR)
                        {
                              if (snd_dir(fname, sockfd) == Normal)
                                    printf("sending dir %s completed.\n", fname);
                              else
                                    printf("error sending dir %s\n", fname);
                              (void)dir_loop(fname, sockfd);
                        }
                        else
                        {
                              if (snd_file(fname, sockfd, fsiz) == Normal)
```

```
                                                {
                                                        printf("completed.\n");
                                                        cnt = read(sockfd, tmp, Small);
                                                }
                                                else
                                                {
                                                        printf("error.\n");
                                                        /* modify */
                                                        rt = Illegal;
                                                        break;
                                                }
                                        }
                                }
                                /*
                                 * the entry is neither file nor directory
                                 */
                                else
                                {
                                        continue;
                                }
                        }
                        else
                        {
                                /*
                                 * not the owner
                                 */
                                rt = Illegal;
                                break;
                        }
                }
        }
        closedir(dp);
        return rt;
}

/*
 * send file content to remote host with a T/F byte to signal premature end if the
 * file size is zero, where T denotes zero file size and F denotes otherwise.
 */
int
snd_file(char *fn, int sockfd, int fsiz)
{
        int             num, i, k;
        FILE            *fp;
        char            new_file[BUFSIZ];
        char            answerb[Small];
        unsigned char   buf2[BUFSIZ];
        sprintf(new_file, "%s (%s) %s ", cmd1, fn, wmode);
        printf("downloading file %s, please wait....", fn);
        if ((fp = fopen(fn, rmode)) == NULL || (loc_open(new_file, sockfd)) == 0)
                return File_error;
        sprintf((char *)buf2, "%1d", fsiz);
        write(sockfd, (char *)buf2, 1);
        if (fsiz == TRUE)
                return Normal;
        /*
         * caveat: byte transfer does not work well in real world
         * transfmission, hence buffered sent (and answer back for
         * reason stated above).
         */
        i = 0;
        k = 0;
        while ((num = fgetc(fp)) != EOF)
        {
                buf2[i++] = num;
                if (i == Transfer_limit)
                {
                        buf2[i] = Null_char;
                        write(sockfd, (char *)buf2, Transfer_limit);
                        read(sockfd, answerb, 1);
                        k += i;
                        i = 0;
                }
        }
        fclose(fp);
        buf2[i] = Null_char;
        write(sockfd, (char *)buf2, i);
        return Normal;
}

/*
 * request remote file open
 */
int
loc_open(char *fn, int sockfd)
{
        int     rp;
        int     n;
        char    tmp[BUFSIZ];

        n = strlen(fn);
        write(sockfd, fn, n);
        rp = read(sockfd, tmp, BUFSIZ);
        return rp;
}
```

```
/*
 * request remote create directory
 */
int
snd_dir(char *fn, int sockfd)
{
        int     n;
        char    tmp[BUFSIZ];

        sprintf(tmp, "%s (%s) ", cmd3, fn);
        n = strlen(tmp);
        write(sockfd, tmp, n);
        (void)read(sockfd, tmp, BUFSIZ);
        return Normal;
}

/*
 * filter out '.' and '..' entries, as well as extra '/'
 */
int
check_dir(char *cur_name)
{
        if (cur_name[0] == Slash ||
            (cur_name[0] == Cur_dot && cur_name[1] == Cur_dot))
                return Illegal;
        return Normal;
}

/*
 * skip extra './' pair
 */
char *
stripping(char *cur_name)
{
        char    *p;

        p = cur_name;
        while (*p != Null_char && *p == Cur_dot)
        {
                if (*(p+1) == Slash)
                        p = p + 2;
                else
                        break;
        }
        return p;
}
```

```
/*
 * look for illegal file name.  If 'Sep2' grows, change the size of 'cstr'.
 */
int
wrong_name(char *name)
{
	int	i;
	char	cstr[Small];

	strcpy(cstr, Sep2);
	for (i = 0; cstr[i] != Null_char; i++)
	{
		if (strchr(name, cstr[i]) != NULL)
			return TRUE;
	}
	return FALSE;
}
```

```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: January 2, 2002
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Distributed Operating Systems (2301462) classnote.
 * Description: This sample client module illustrates remote host
 *           communication over standard TCP/IP connection.
 * History of Modification:
 *    Date: May 16, 2002 by Peraphon Sophatsathit
 *    This program, along with a directory search routine developed
 *    separately as an independent module, demonstrates some practical
 *    aspects of data exchange over TCP/IP socket as part of the
 *    Data Communcation I class (2301369).  The use of this code is
 *    free, provided that this information is kept intact.
 * Invocation:
 *    The server module must be invoked first to initiate the connection
 *    with default option, i.e.,
 *           $ server  0
 *    the client module is subsequently invoked to establish the
 *    communication in a 'simplex chat' manner, that is, a one-way
 *    talk session from client to server as follows:
 *           $ client  161.200.126.10
 * Input:
 *    at the prompt 'client>' from client side, two commands can be issued
 *    as follows:
 *           client>#get   .            (to download from server to client)
 *    or     client>#put   .            (to upload from client to server)
 * Features:
 *    add directory retrieval module to incorporate down/upload capabilities.
 *    from client to server.  As such, the main 'str_echo' routine underwent
 *    considerable change to accommodate these new features.  In addition,
 *    an adhoc exchange discipline was improvised to cope with some socket's
 *    idiosyncracies that hadn't been unraveled by the author.
 * Transmission exchange framework:
 *    1. set up read/write pair to ensure proper handshake (see download
 *       related values)
 *    2. use 'char *' buffer to transmit byte information over the socket
 *       rather than the usual read/write system calls syntax
 * Caveat:
 *    Different OSes use different 'BUFSIZ' which, in practice, results in
 *    unsynchronized byte exchange, hence the use of 'answer back' to keep
 *    both sides in sync.
 */
```

```
#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>
#include     <errno.h>

/*
 * porting from BSD to SVR4
 */
#ifdef __USE_BSD
#  include   <machine/param.h>
#endif

#include     <sys/types.h>
#include     <sys/socket.h>
#include     <netinet/in.h>
#include     <arpa/inet.h>
#include     <signal.h>
#include     <unistd.h>
#include     <time.h>

/*
 * prototypes
 */
unsigned int alarm(unsigned int);
void         (*signal(int, void (*disp)(int))) (int);
void         handler(int);
void         t_out(int);
void         bzero(void *, size_t);
void         *memcpy(void *dest, const void *str, size_t nbytes);
int          socket(int, int, int);
int          connect(int, const struct sockaddr *, socklen_t);
int          inet_pton(int, const char *, void *);
int          inet_aton(const char *, struct in_addr *);
char         *strtok_r(char *, const char *, char **);

int          proc_loop(char *);
int          str_cli(FILE *, int);
int          parse_ln(char *, char *);
void         clear_buff(char *, int);

int          pdir(char *, int);
int          check_dw(char *, int);
```

```
/*
 * download commands
 */
#define             SERV_PORT    9877
#define             Null_char    '\0'
#define             Sep          " \t"
#define             Sep2         "()"
#define             Gets         "#get"
#define             Puts         "#put"
#define             Ready_put    "Rput"
#define             End_dw       "#End#"
#define             End_fl       "eof"
#define             Get_token    1000
#define             Put_token    1001
#define             Ready_token  1002
#define             Prompts      "client"
#define             Happy_open   "pass open"
#define             Rcv_mkdir    "received mkdir"
#define             Dw_done      "download completed!\n"
#define             Transfer_limit(1000)
#ifdef SURE
#define             Transfer_limit(BUFSIZ - 2)
#endif

/*
 * error return code
 */
#define             Normal       0
#define             Err_socket   1
#define             Err_bind     2
#define             Err_listen   3
#define             Err_accept   4
#define             Err_connect  5
#define             Err_write    6
#define             Err_read     7
#define             Err_IP       10
#define             Err_fork     88
#define             Err_usage    99
#define             DW_ok        887
#define             DW_failed    888
#define             DW_done      889

/*
 * globals
 */
#define             EQ(a, b)     (strcmp(a, b) == 0)
#define             NE(a, b)     (strcmp(a, b) != 0)
#define             Small        20
```

```
#define             TRUE         1
#define             FALSE        0

#define             rmode        "r"
#define             wmode        "w"
#define             Cur_dot      '.'
#define             Slash        '/'
#define             Dots         "."
#define             Dotts        ".."
#define             cmd1         "fopen"
#define             cmd3         "mkdir"
#define             Nul_str      ""

char         *desc[]      =
{
       "Description: Wait 0 second for timeout which is recommended.",
       "Any other values can be used as a precaution to prevent the",
       "process from running away, but will cause an abnormal",
       "termination.  However, too long a wait will have no effect",
       "if the process has already terminated.\n",
       ""
};

int          flag = FALSE;
FILE         *ffp;

/*
 * The purpose of signal calls employed in this program is to prevent
 * runaway processing.  The user may terminate (kill) the process any time
 * via user command (ctrl C) or timer.  The latter can be set to any
 * positive integer ranging from 0 to N (N is recommneded to be small to
 * have any effect).  All signals may appear to have no effect if control
 * is suspended by '(blocking) read'.  In which case, one must send a
 * message by typing from keyboard to get out of 'read' wait.
 * Note that in order for the signals to have an immediate effect,
 * non-blocking read must be set along with extra precaution to handle
 * any 'non-blocking' timing and synchronization idiosyncracies.
 */
int
main(int ac, char **av)
{
       char         buf[Small];
       int          rt_code, i;
       unsigned int sec = 0;

       signal(SIGINT, handler);
       signal(SIGQUIT, handler);
       signal(SIGALRM, t_out);
	switch (ac)
	{
		case 3:
			sec = atoi(av[2]);
			if (sec > 0)
				alarm(sec);
		case 2:
			strcpy(buf, av[1]);
			break;
		default:
			printf("\nUsage: %s  IPaddress          [ wait_sec ]\n\n", av[0]);
			printf("Example: %s  161.200.192.17     (default to no timeout)\n", av[0]);
			printf("Example: %s  161.200.192.17  0  (same as default)\n", av[0]);
			printf("Example: %s  161.200.192.17  5  (timeout in 5 seconds)\n\n", av[0]);
			for (i = 0; NE(desc[i], Nul_str); i++)
				printf("%s\n", desc[i]);
			return Err_usage;
	}
	rt_code = proc_loop(buf);
	if (rt_code > Normal || flag == TRUE)
		printf("Abnornal termination of RPC loop\n");
	fflush(stdout);
	fflush(stderr);
	return Normal;
}

/*
 * setup C/S connection
 */
int
proc_loop(char *adr)
{
	int			sockfd;
	int			rt;
	struct sockaddr_in	servaddr;

	/*
	 * open client socket to communicate with the server via
	 * standard TCP connection.
	 */
	if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
	{
		return Err_socket;
	}
	bzero(&servaddr, sizeof(servaddr));
	servaddr.sin_family  = AF_INET;
	servaddr.sin_port    = htons(SERV_PORT);
```

```
        if (inet_pton(AF_INET, adr, &servaddr.sin_addr) <= 0)
        {
                printf("invalid IP address: <%s>\n", adr);
                return Err_IP;
        }

        if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
        {
                return Err_connect;
        }
        rt = str_cli(stdin, sockfd);
        return rt;
}

/*
 * returned value:
 *    -1    af does not contain a valid address family
 *     0    src does not contain a character string representing a valid network address
 *     1    network address is successfully converted
 */
int
inet_pton(int family, const char *strptr, void *addrptr)
{
        struct in_addr      in_val;
        int                 rt;

        if (family == AF_INET)
        {
                if (inet_aton(strptr, &in_val) == 1)
                {
                        memcpy(addrptr, &in_val, sizeof(struct in_addr));
                        rt = 1;
                }
                else
                {
                        rt = 0;
                }
        }
        else
        {
                errno = EAFNOSUPPORT;
                rt = -1;
        }
        return rt;
}
```

```
/*
 * send and receive messages and download commands (get/put)
 *
 * Change notes:
 * 'check_dw', etc., were added to intercept the input from server end.
 */
int
str_cli(FILE *fp, int sockfd)
{
      int    cflg;
      int    n, m;
      char   *t;
      char   sline[BUFSIZ], rline[BUFSIZ];
      char   fn[BUFSIZ];

      /*
       * read prompt string from host (this must be changed if
       * different handshake protocol is used
       */
      n = read(sockfd, rline, BUFSIZ);
      rline[n] = Null_char;
      fputs(rline, stdout);
      clear_buff(sline, BUFSIZ);
      /*
       * read from stdin and send it over to server.  Echo the info
       * getting back from server.
       */
      printf("%s>", Prompts);
      t = fgets(sline, BUFSIZ, fp);
      m = strlen(sline);
      write(sockfd, sline, m);
      clear_buff(sline, m);
      while (flag == FALSE && t != NULL)
      {
            clear_buff(rline, BUFSIZ);
            if ((n = read(sockfd, rline, BUFSIZ)) == 0)
            {
                  printf("connection closed by server\n");
                  break;
            }
            /*
             * look for file transfer command from the incoming byte streams
             */
            if ((cflg = check_dw(rline, sockfd)) == DW_ok)
                  continue;
            else if (cflg == DW_failed)
            {
                  printf("download failed\n");
```

```
                        break;
                }
                /*
                 * normal (chat) message and download commands from keyboard
                 */
                if ((cflg = parse_ln(rline, fn)) == Normal)
                {
                        printf("%s>", Prompts);
                        t = fgets(sline, BUFSIZ, fp);
                        m = strlen(sline);
                        write(sockfd, sline, m);
                        clear_buff(sline, BUFSIZ);
                }
        }
        return Normal;
}

/*
 * check if input contains file transfer commands, i.e.,
 * 'fopen' and 'mkdir'
 * Note: strtok_r creates side-effect on original string, hence the use of 'tmp'
 */
int
check_dw(char *ln, int sockfd)
{
        int             num;
        char            size;
        char            *t, *q, *brk, *v;
        char            tmp[BUFSIZ];
        char            buf[BUFSIZ], answerb[Small];
        unsigned char   buf2[BUFSIZ];
        int             n, rt;

        strcpy(tmp, ln);
        t = strtok_r(tmp, Sep, &brk);
        if (EQ(t, cmd1) || EQ(t, cmd3) || EQ(t, End_dw) || EQ(t, Ready_put))
        {
                rt = DW_ok;
                /*
                 * open a new local file
                 */
                if (EQ(t, cmd1))
                {
                        q = strtok_r(NULL, Sep2, &brk);
                        strcpy(buf, q);
                        printf("receiving file %s...", buf);
                        if ((ffp = fopen(buf, wmode)) == NULL)
                                return DW_failed;
        /*
         * write something just to keep remote open happy
         */
        strcpy(tmp, Happy_open);
        n = strlen(tmp);
        write(sockfd, tmp, n);
        /*
         * file size is greater than zero (=FALSE)
         */
        (void)read(sockfd, &size, 1);
        n = size - 48;
        if (n == FALSE)
        {
                /*
                 * make sure the buffer is cleared.  Any left
                 * over characters will result in error.
                 */
                clear_buff(tmp, BUFSIZ);
                num = Transfer_limit;
                /*
                 * transmission rate is too fast, hence the
                 * 'answer back' pause.
                 */
                while ((n = read(sockfd, (char *)buf2, num)) == num)
                {
                        fwrite((void *)buf2, (size_t)n, 1, ffp);
                        fflush(ffp);
                        answerb[0] = Null_char;
                        write(sockfd, answerb, 1);
                }
                if (n > 0)
                {
                        fwrite((void *)buf2, (size_t)n, 1, ffp);
                        fflush(ffp);
                }
        }
        fclose(ffp);
        /*
         * signal end of (writing) target file
         */
        n = strlen(End_fl);
        write(sockfd, End_fl, n);
        printf("done\n");
}
else if (EQ(t, cmd3))
{
```

```
                        /*
                         * write something just to keep remote 'snd_dir' happy
                         */
                        q = strtok_r(NULL, Sep2, &brk);
                        strcpy(buf, Rcv_mkdir);
                        n = strlen(buf);
                        write(sockfd, buf, n);
                        clear_buff(buf, BUFSIZ);
                        sprintf(buf, "%s %s", t, q);
                        /*
                         * check directory existence
                         */
                        if (system(buf) != 0)
                        {
                                sprintf(buf, "test -d %s", q);
                                if (system(buf) != 0)
                                        return DW_failed;
                        }
                }
                else if (EQ(t, Ready_put))
                {
                        q = strtok_r(NULL, Sep2, &brk);
                        strcpy(tmp, q);
                        if ((v = strchr(tmp, '\n')) != NULL)
                                *v = Null_char;
                        pdir(tmp, sockfd);
                }
                /*
                 * end download string
                 */
                else
                {
                        strcpy(tmp, Dw_done);
                        n = strlen(tmp);
                        write(sockfd, tmp, n);
                }
        }
        else
        {
                rt = Normal;
        }
        return rt;
}
```

```
/*
 * parse input to locate download command
 * Note: strtok_r creates side-effect on original string, hence the use of 'tmp'
 */
int
parse_ln(char *ln, char *fn)
{
	int	rt;
	char	*t, *brk;
	char	tmp[BUFSIZ];

	strcpy(tmp, ln);
	t = strtok_r(tmp, Sep, &brk);
	if (EQ(t, Gets) || EQ(t, Puts) || EQ(t, Ready_put))
	{
		if (EQ(t, Gets))
			rt = Get_token;
		else if (EQ(t, Puts))
			rt = Put_token;
		else
			rt = Ready_token;
		t = strtok_r(NULL, Sep2, &brk);
		strcpy(fn, t);
		return rt;
	}
	return Normal;
}

/*
 * clear R/W buffer to null
 */
void
clear_buff(char *line, int n)
{
	register int	i;

	for (i = 0; i < n; i++)
		line[i] = Null_char;
	return;
}
```

```
/*
 * time out by alarm clock
 */
void
t_out(int sig)
{
	signal(SIGALRM, t_out);
	flag = TRUE;
	printf("timeout by ALARM signal\n");
	return;
}

/*
 * interrupt and kill signals
 */
void
handler(int sig)
{
	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	flag = TRUE;
	printf("receiving INT/QUIT signal\n");
	return;
}
```

```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: May 15, 2002
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Written for use as a sample classnote.
 * Description: This sample directory retrieval module accesses and
 *              retrieves files and sub-dir in the specified directory
 *              within the user authorization.  When coupled with
 *              the network communication modules, it serves as a
 *              file download utility.
 * History of change:
 *      Date: May 20, 2002 by Peraphon Sophatsathit
 *      To make it callable from the above 'semi-chat' programs, the
 *      'main' module was changed to 'pdir' to accommodate directory
 *      search and retrieval.
 *
 *      Date: June 3, 2002 by Peraphon Sophatsathit
 *      It was discovered that all bytes transmitted over socket should
 *      be made 'unsigned char' rather than 'char' to cope with 8-bit
 *      byte pattern, hence the use of 'buf2' as an adhoc fix.
 */

#include        <dirent.h>

#ifdef __USE_BSD
#include        <sys/param.h>
extern long     int     telldir(DIR *);
#else
#define         __USE_BSD
#include        <sys/stat.h>
#endif

/*
 * system functions
 */
DIR             *opendir(const char *);
struct dirent   *readdir(DIR *);
int             closedir(DIR *);
int             lstat(const char *, struct stat *);
```

```
/*
 * unused system functions, but might come in handy later.
 */
#ifdef FUTURE_USED
int          chdir(const char *);
int          fchdir(int);
void         seekdir(DIR *, long);
void         rewinddir(DIR *);
#endif

/*
 * prototypes
 */
int          dir_loop(char *, int);
int          check_dir(char *);
char         *stripping(char *);
int          snd_file(char *, int, int);
int          snd_dir(char *, int);
int          loc_open(char *, int);
int          wrong_name(char *);

/*
 * error conditions
 */
#define             Illegal      1
#define             U_defn       2
#define             File_error   20
#define             Errors       99

/*
 * main driver to retrieve directory information
 */
int
pdir(char *av, int sockfd)
{
      int    rt_code, pos, n;
      char   *cur_name, *p;
      char   dir_ptr[BUFSIZ];
      char   tmp[BUFSIZ];

      p = av;
      cur_name = stripping(p);
      if (*cur_name == Null_char)
      {
             strcpy(dir_ptr, Dots);
      }
      else if (NE(cur_name, p))
             strcpy(dir_ptr, cur_name);
	else
		strcpy(dir_ptr, p);
	pos = check_dir(dir_ptr);
	if (pos == Normal)
	{
		rt_code = dir_loop(dir_ptr, sockfd);
		/*
		 * send end_download string to the other end
		 */
		strcpy(tmp, End_dw);
		n = strlen(tmp);
		write(sockfd, tmp, n);
	}
	else
	{
		rt_code = Errors;
		printf("%s directory path is not permitted\n", dir_ptr);
	}
	return rt_code;
}

/*
 * recursively traverses the directory tree
 * Since modern UNIX permits embeded blanks in file name, special
 * 'delimiters' were employed to avoid conventional 'whitespace'
 * delimiters, e.g., tab, blank, and newline.  The 'parenthesis'
 * was chosen to internally delimit the entire file name.  As such,
 * any source file having 'parenthesis' pairs will not be transmitted
 * to avoid conflict (see 'wrong_name' block).
 */
int
dir_loop(char *name, int sockfd)
{
	struct stat	st;
	struct dirent	*diry;
	DIR		*dp;
	uid_t		uid, euid;
	gid_t		gid;
	mode_t		md;
	int		rt = Normal;
	int		cnt, fsiz;
	char		fname[BUFSIZ];
	char		tmp[BUFSIZ];
```

```
/*
 * open directory
 */
if ((dp = opendir(name)) == NULL)
{
      rt = U_defn;
      perror(name);
      return rt;
}
while ((diry = readdir(dp)) != NULL)
{
      if (EQ(diry->d_name, Dots) || EQ(diry->d_name, Dotts))
            continue;
      else if (wrong_name(diry->d_name) == TRUE)
      {
            printf("File name containing illegal characters <%s>, ", diry->d_name);
            printf("skip to next entry\n");
            continue;
      }
      sprintf(fname, "%s/%s", name, diry->d_name);
      if (lstat(fname, &st) < 0)
      {
            perror(fname);
            rt = Illegal;
            break;
      }
      else
      {
            uid  = getuid();
            euid = geteuid();
            gid  = getgid();
            fsiz = st.st_size == 0 ? TRUE : FALSE;
            if ((st.st_uid == uid || st.st_uid == euid) && st.st_gid == gid)
            {
                  md = st.st_mode & S_IFMT;
                  if (md == S_IFREG || md == S_IFDIR)
                  {
                        if (md == S_IFDIR)
                        {
                              if (snd_dir(fname, sockfd) == Normal)
                                    printf("sending dir %s completed.\n", fname);
                              else
                                    printf("error sending dir %s\n", fname);
                              (void)dir_loop(fname, sockfd);
                        }
                        else
                        {
                              if (snd_file(fname, sockfd, fsiz) == Normal)
```

```
                                                {
                                                        printf("completed.\n");
                                                        cnt = read(sockfd, tmp, Small);
                                                }
                                                else
                                                {
                                                        printf("error.\n");
                                                        /* modify */
                                                        rt = Illegal;
                                                        break;
                                                }
                                        }
                                }
                                /*
                                 * the entry is neither file nor directory
                                 */
                                else
                                {
                                        continue;
                                }
                        }
                        else
                        {
                                /*
                                 * not the owner
                                 */
                                rt = Illegal;
                                break;
                        }
                }
        }
        closedir(dp);
        return rt;
}

/*
 * send file content to remote host with a T/F byte to signal premature end if the
 * file size is zero, where T denotes zero file size and F denotes otherwise.
 */
int
snd_file(char *fn, int sockfd, int fsiz)
{
        int             num, i, k;
        FILE            *fp;
        char            new_file[BUFSIZ], answerb[Small];
        unsigned char   buf2[BUFSIZ];
        sprintf(new_file, "%s (%s) %s ", cmd1, fn, wmode);
        printf("downloading file %s, please wait....", fn);
        if ((fp = fopen(fn, rmode)) == NULL || (loc_open(new_file, sockfd)) == 0)
                return File_error;
        sprintf((char *)buf2, "%1d", fsiz);
        write(sockfd, (char *)buf2, 1);
        if (fsiz == TRUE)
                return Normal;
        /*
         * caveat: byte transfer does not work well in real world
         * transfmission, hence buffered sent (and answer back for
         * reason stated above).
         */
        i = 0;
        k = 0;
        while ((num = fgetc(fp)) != EOF)
        {
                buf2[i++] = num;
                if (i == Transfer_limit)
                {
                        buf2[i] = Null_char;
                        write(sockfd, (char *)buf2, Transfer_limit);
                        read(sockfd, answerb, 1);
                        k += i;
                        i = 0;
                }
        }
        fclose(fp);
        buf2[i] = Null_char;
        write(sockfd, (char *)buf2, i);
        return Normal;
}

/*
 * request remote file open
 */
int
loc_open(char *fn, int sockfd)
{
        int     rp;
        int     n;
        char    tmp[BUFSIZ];

        n = strlen(fn);
        write(sockfd, fn, n);
        rp = read(sockfd, tmp, BUFSIZ);
        return rp;
}
```

```
/*
 * request remote create directory
 */
int
snd_dir(char *fn, int sockfd)
{
       int    n;
       char   tmp[BUFSIZ];

       sprintf(tmp, "%s (%s) ", cmd3, fn);
       n = strlen(tmp);
       write(sockfd, tmp, n);
       (void)read(sockfd, tmp, BUFSIZ);
       return Normal;
}

/*
 * filter out '.' and '..' entries, as well as extra '/'
 */
int
check_dir(char *cur_name)
{
       if (cur_name[0] == Slash ||
          (cur_name[0] == Cur_dot && cur_name[1] == Cur_dot))
              return Illegal;
       return Normal;
}

/*
 * skip extra './' pair
 */
char *
stripping(char *cur_name)
{
       char   *p;

       p = cur_name;
       while (*p != Null_char && *p == Cur_dot)
       {
              if (*(p+1) == Slash)
                     p = p + 2;
              else
                     break;
       }
       return p;
}
```

```c
/*
 * look for illegal file name.  If 'Sep2' grows, change the size of 'cstr'.
 */
int
wrong_name(char *name)
{
	int	i;
	char	cstr[Small];

	strcpy(cstr, Sep2);
	for (i = 0; cstr[i] != Null_char; i++)
	{
		if (strchr(name, cstr[i]) != NULL)
			return TRUE;
	}
	return FALSE;
}
```