CS211 Lab07
fa2022
streller/dutta

PROBLEM STATEMENT:

More on client, server communication via sockets.

CODE :

This lab deals with the creation of sockets and communications with them, sockets are used to communicate over a network between two computers for this lab you will only communicate between two processes running on the same computer.

The function socketpair synopsis is as follows:

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sv[2]);
```

The include file sys/types.h is required to define some C macro constants. The include file sys/socket.h is necessary to define the socketpair function prototype.

The socketpair function takes four arguments. They are

- The domain of the socket.
- The type of the socket.
- The protocol to be used.
- The pointer to the array that will receive file descriptors that reference the created sockets.

In detail, the four arguments:

For the socket pair function, however, always supply the C macro value AF_LOCAL for the *domain value.*

The *type* argument declares what type of socket you want to create. The choices for the socketpair function are

- SOCK_STREAM
- SOCK_DGRAM

For this lab, we'll simply use SOCK_STREAM for the type of the socket.

For the socketpair function, the *protocol* argument must be supplied as zero.

The argument *sv [2]* is a receiving array of two integer values that represent two sockets. Each file descriptor represents one socket (endpoint) and is otherwise indistinguishable from the other.
If the function is successful, the value zero is returned. Otherwise, a return value of -1 indicates that a failure has occurred, and that errno should be consulted for the specific reason.

NOTE

Always test the function return value for success or failure. The value errno should only be consulted when it has been determined that the function call has indicated that it failed. Only errors are posted to errno; it is never cleared to zero upon success.

Example socketpair

To demonstrate how the socketpair(2) function is used, the program in Listing1 is presented for your experimentation.

```
/*
* cs211_L7_listing1.c
*
* Example of socketpair(2) function:
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(int argc,char **argv)
{
  int z;                /* Status return code */
  int s[2];             /* Pair of sockets */

  /*
   * Create a pair of local sockets :
   */
  z = socketpair(AF_LOCAL,SOCK_STREAM,0,s);

  if ( z == -1 )
    {
      fprintf(stderr, %s: socketpair(AF_LOCAL,SOCK_STREAM,0)\n", strerror(errno));
      return 1;          /* Failed */
    }

  /*
   * Report the socket file descriptors returned:
   */
  printf("s[0] = %d;\n",s[0]);
  printf("s[1] = %d;\n",s[1]);

  system("netstat --unix -p");

  return 0;
}
```

Create the program and run it. Record the output only for the program itself.

s[0]	s[1]	Program Name	I-node #	PID

Sockets can be written to and read from just like any opened file. You are going to demonstrate this firsthand for yourself. For the sake of completeness however, let's review the function synopsis for the calls read, write, and close before we put them to work:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

These are Linux input/output functions. The function read returns input that is available from the file descriptor fd, into your supplied buffer buf of a maximum size of count bytes. The return value represents the number of bytes read. A return count of zero represents end-of-file.

The write function writes data to your file descriptor fd, from your supplied buffer buf for a total of count bytes. The returned value represents the actual number of bytes written. Normally, this should match the supplied count argument.

Finally, close returns zero if the unit was closed successfully.

A return value of -1 for any of these functions indicates that an error occurred, and that the reason for the error is posted to the external variable errno. To make this value accessible, include the file errno.h within the source module that needs it.

Example 2:

Look at the following code:for cs211_L7_listing2.c (included in this lab as separate file)

```
/* Listing2:
 *
 * Example performing I/O on a Socket Pair :
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(int argc,char **argv)
{
  int z;              /* Status return code */
  int s[2];           /* Pair of sockets */
  char *cp;           /* A work pointer */
  char buf[80];       /* Work buffer */

  /*
   * Create a pair of local sockets :
   */
  z = socketpair(AF_LOCAL,SOCK_STREAM,0,s);

  if ( z == -1 )
    {
      fprintf(stderr, "%s: socketpair(AF_LOCAL,SOCK_STREAM," "0)\n", strerror(errno));
      return 1;           /* Failed */
    }

  /*
   * Write a message to socket s[1] :
   */
  cp = "PLACE YOUR INITIAL MESSAGE HERE";
  z = write(s[1],cp,strlen(cp));
  if ( z < 0 )
    {
      fprintf(stderr, "%s: write(%d,\"%s\",%d)\n", strerror(errno),s[1],cp,strlen(cp));
```

```
    return 2;   /* Failed write */
  }

 printf("Wrote message '%s' on s[1]\n",cp);

 /*
  * Read from socket s[0] :
  */
 z = read(s[0],buf,sizeof buf);
 if ( z < 0 )
  {
    fprintf(stderr, "%s: read(%d,buf,%d)\n", strerror(errno),s[0],sizeof buf);
    return 3;   /* Failed read */
  }

 /*
  * Report received message :
  */
 buf[z] = 0;    /* NUL terminate */
 printf("Received message '%s' from socket s[0]\n", buf);

 /*
  * Send a reply back to s[1] from s[0] :
  */
 cp="PLACE YOUR RESPONSE MESSAGE HERE";
 z = write(s[0],cp,strlen(cp));
 if ( z < 0 )
  {
    fprintf(stderr, "%s: write(%d,\"%s\",%d)\n", strerror(errno),s[0],cp,strlen(cp));
    return 4;   /* Failed write */
  }

 printf("Wrote message '%s' on s[0]\n",cp);

 /*
  * Read from socket s[1] :
  */
 z = read(s[1],buf,sizeof buf);
 if ( z < 0 )
  {
    fprintf(stderr, "%s: read(%d,buf,%d)\n", strerror(errno),s[1],sizeof buf);
    return 3;   /* Failed read */
  }

 /*
  * Report message received by s[0] :
  */
 buf[z] = 0;    /* NUL terminate */
 printf("Received message '%s' from socket s[1]\n", buf);

/*
  * Close the sockets :
  */
 close(s[0]);
 close(s[1]);

 puts("Done.");
 return 0;
}
```

For this program you will not have to type it in. See included listing2.c. Edit the code for your messages i.e. PLACE YOUR MESSAGE INITIAL MESSAGE HERE and PLACE OUR RESPONSE MESSAGE HERE, compile and run it.

A pair of sockets can be easily created and some elementary input and output can be performed using those sockets. These sockets could be closed in the same manner that files are with the use of the close function call. It's now time that you learn what is implied by the closing of a socket.

The receiving end of a socket will receive an end-of-file indication when the other endpoint (socket) has been closed.

A problem develops when the local process wants to signal to the remote endpoint that there is no more data to be received. If the local process closes its socket, this much will be accomplished. However, if it needs to receive a confirmation from the remote end, it cannot, because its socket is now closed. Situations like these require a means to half close a socket.

The shutdown Function

The following shows the function synopsis of the shutdown function:

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

The function shutdown requires two arguments. They are

- Socket descriptor s specifies the socket to be partially shut down.
- Argument how indicates how this socket should be shut down.

The returned value is zero if the function call succeeded. A failure is indicated by returning a value of -1, and the reason for the failure is posted to errno.

The permissible values for how are shown in the Table.

Table: Permissible Values of the shutdown how Argument

Value	Macro	Description
0	SHUT_RD	No further reads will be allowed on the specified socket.
1	SHUT_WR	No further writes will be allowed on the specified socket.
2	SHUT_RDWR	No further reads or writes will be allowed on the specified socket.

Notice that when the how value is supplied as 2, this function call become almost equivalent to a close call.

Shutting Down Writing to a Socket

The following code shows how to indicate that no further writes will be performed upon the local socket:

```
int z;
int s; /* Socket */

z = shutdown(s, SHUT_WR);
if ( z == -1 )
 perror("shutdown()");
```

Shutting down the writing end of a socket solves a number of thorny problems. They are

- Flushes out the kernel buffers that contain any pending data to be sent. Data is buffered by the kernel networking software to improve performance.
- Sends an end-of-file indication to the remote socket. This tells the remote reading process that no more data will be sent to it on this socket.
- Leaves the partially shutdown socket open for reading. This makes it possible to receive confirmation messages after the end-of-file indication has been sent on the

socket.

- Disregards the number of open references on the socket. Only the last close on a socket will cause an end-of-file indication to be sent.

The last point requires a bit of explanation, which is provided in the next section.

Dealing with Duplicated Sockets

If a socket's file descriptor is duplicated with the help of a dup or a dup2 function call, then only the last outstanding close call actually closes down the socket. This happens because the other duplicated file descriptors are still considered to be in use. This is demonstrated in the following code:

```
int s; /* Existing socket */
int d; /* Duplicated socket */

d = dup(s); /* duplicate this socket */
close(s); /* nothing happens yet */
close(d); /* last close, so shutdown socket */
```

In the example, the first close call would have no effect. It would make no difference which socket was closed first. Closing either s or d first would still leave one outstanding file descriptor for the same socket. Only when closing the last surviving file descriptor for that socket would a close call have any effect. In the example, the close of the d file descriptor closes down the socket.

The shutdown function avoids this difficulty. Repeating the example code, the problem is solved using the shutdown function:

```
int s; /* Existing socket */
int d; /* Duplicated socket */

d = dup(s); /* duplicate this socket */
shutdown(s, SHUT_RDWR); /* immediate shutdown */
```

Even though the socket s is also open on file unit d, the shutdown function immediately causes the socket to perform its shutdown duties as requested. This naturally affects both the open file descriptors s and d because they both refer to the same socket.

Another way this problem is manifested is after a fork function has been called upon. Any sockets that existed prior to a fork operation would be duplicated in the child process.

Use the shutdown function instead of the close function whenever immediate or partial shutdown action is required. Duplicated file descriptors from dup, dup2, or fork operations can prevent a close function from initiating any shutdown action until the last outstanding descriptor is closed.

Shutting Down Reading from a Socket

Shutting down the read side of the socket causes any pending read data to be silently ignored. If more data is sent from the remote socket, it too is silently ignored. Any attempt by the process to read from that socket, however, will have an error returned to it.

Knowing When Not to Use shutdown

The shutdown function is documented to return the errors shown in the Table.

Table: Possible errors returned by shutdown

Error	Description
EBADF	Given socket is not a valid file descriptor.
ENOTSOCK	Given file descriptor is not a socket.
ENOTCONN	The specified socket is not connected.

From the Table you can see that you should only call shutdown for connected sockets. Otherwise, the error code ENOTCONN is returned.

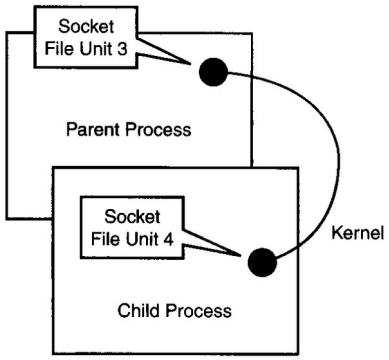
> NOTE
> The shutdown function does not release the sockets file unit number, even when SHUT_RDWR is used. The socket's file descriptor remains valid and in use until the function close is called to release it.

Note also that shutdown can be called more than once for the same file unit, provided that the socket is still connected.

Writing a Client/Server Example

You have now looked at enough of the socket API set to start having some fun with it. Now you will examine, compile, and test a simple client and server process that communicates with a pair of sockets.

To keep the programming code to a bare minimum, one program will start and then fork into a client process and a server process. The child process will assume the role of the client program, whereas the original parent process will perform the role of the server. The figure illustrates the relationship of the parent and child processes and the sockets that will be used.



A Client/Server example using fork and socketpair.

The parent process is the original starting process. It will immediately ask for a pair of sockets by calling socketpair and then fork itself into two processes by calling fork.

The server will accept one request, act on that request, and then exit. The client likewise in this example will issue one request, report the server response, and then exit.

The request will take the form of the third argument to the strftime function. This is a format string, which will be used to format a date and time string. The server will obtain the current date and time at the time that the request is received. The server will use the client's request string to format it into a final string, which is returned to the client. The strftime function's synopsis is as follows:

```
#include <time.h>

size_t strftime(char *buf, size_t max, const char *format, const struct tm *tm);
```

The arguments buf and max indicate the output buffer and its maximum length respectively. The argument format is an input string that allows you to format a date and time string. Finally, argument tm is used to supply all the date and time components necessary to create the output date and time string.

Listing3 shows the source listing for the demonstration client/server program.

```
/* Listing3
 *
 * Client/Server Example Using socketpair(2)
 * and fork(2) :
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>

#ifndef SHUT_WR
#define SHUT_RD     0
#define SHUT_WR     1
#define SHUT_RDWR   2
#endif

/*
 * Main program :
 */
int main(int argc,char **argv)
{
  int z;          /* Status return code */
  int s[2];       /* Pair of sockets */
  char *msgp;     /* A message pointer */
  int mlen;       /* Message length */
  char buf[80];   /* Work buffer */
  pid_t chpid;    /* Child PID */

  /*
   * Create a pair of local sockets :
   */
  z = socketpair(AF_LOCAL,SOCK_STREAM,0,s);

  if ( z == -1 )
    {
      fprintf(stderr,"%s: socketpair(2)\n",
              strerror(errno));
      exit(1);
    }

  /*
   * Now fork() into two processes :
   */
  if ( (chpid = fork()) == (pid_t)-1 )
    {

      /*
       * Failed to fork into two processes :
       */
      fprintf(stderr,"%s: fork(2)\n",
```

```
            strerror(errno));
    exit(1);

  }
else if ( chpid == 0 )
  {

    /*
     * This is the child process (client) :
     */
    char rxbuf[80]; /* Receive buffer */

    printf("Parent PID is %ld\n",(long)getppid());

    close(s[0]);    /* Server uses s[1] */
    s[0] = -1;      /* Forget this unit */

    /*
     * Form the message and its length :
     */
    msgp = "%A %d-%b-%Y %l:%M %p";
    mlen = strlen(msgp);

    printf("Child sending request '%s'\n",msgp);
    fflush(stdout);

    /*
     * Write a request to the server :
     */
    z = write(s[1],msgp,mlen);
    if ( z < 0 )
      {
        fprintf(stderr,"%s: write(2)\n",
                strerror(errno));
        exit(1);
      }

    /*
     * Now indicate that we will not be writing
     * anything further to our socket, by shutting
     * down the write side of the socket :
     */
    if ( shutdown(s[1],SHUT_WR) == -1 )
      {
        fprintf(stderr,"%s: shutdown(2)\n",
                strerror(errno));
        exit(1);
      }

    /*
     * Receive the reply from the server :
     */
    z = read(s[1],rxbuf,sizeof rxbuf);
    if ( z < 0 )
      {
        fprintf(stderr,"%s: read(2)\n",
                strerror(errno));
        exit(1);
      }

    /*
     * Put a null byte at the end of what we
     * received from the server:
     */
    rxbuf[z] = 0;

    /*
     * Report the results :
```

```
     */
    printf("Server returned '%s'\n",rxbuf);
    fflush(stdout);

    close(s[1]);    /* Close our end now */

  }
 else
  {

    /*
     * This is the parent process (server) :
     */
    int status; /* Child termination status */
    char txbuf[80];         /* Reply buffer */
    time_t td;       /* Current date & time */

    printf("Child PID is %ld\n",(long)chpid);
    fflush(stdout);

    close(s[1]);    /* Client uses s[1] */
    s[1] = -1;      /* Forget this descriptor */

    /*
     * Wait for a request from the client :
     */
    z = read(s[0],buf,sizeof buf);

    if ( z < 0 )
     {
       fprintf(stderr,"%s: read(2)\n",
               strerror(errno));
       exit(1);
     }

    /*
     * Put a null byte at the end of the
     * message we received from the client:
     */
    buf[z] = 0;

    /*
     * Now perform the server function on the
     * received message :
     */
    time(&td);          /* Get current time */

             strftime(txbuf,sizeof txbuf,  /* Buffer */
             buf,                /* Input format */
             localtime(&td));      /* Input time */

    /*
    * Send back the response to client :
     */
    z = write(s[0],txbuf,strlen(txbuf));

    if ( z < 0 )
     {
       fprintf(stderr,"%s: write(2)\n",
               strerror(errno));
       exit(1);
     }

    /*
     * Close our end of the socket :
     */
    close(s[0]);
```

```
    /*
     * Wait for the child process to exit..
     * See text.
     */
    waitpid(chpid,&status,0);
   }

  return 0;
}
```

Obtain listing3c. and run it, (the source code is found with this lab)
Record the child and parent PID #s and the time sent.

Child PID#	Parent PID#	Request String	Time

The format of the time request depends on the request string, the details are as follows:

DESCRIPTION
The strftime() function formats the broken-down time tm according to the format specification format and places the result in the character array s of size max.

Ordinary characters placed in the format string are copied to s without conversion. Conversion specifiers are introduced by a % character, and are replaced in s as follows:

%a The abbreviated weekday name according to the current locale.
%A The full weekday name according to the current locale.
%b The abbreviated month name according to the current locale.
%B The full month name according to the current locale.
%c The preferred date and time representation for the current locale.
%C The century number (year/100) as a 2-digit integer.
%d The day of the month as a decimal number (range 01 to 31).
%D Equivalent to %m/%d/%y. (Yecch - for Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in international context this format is ambiguous and should not be used.)
%e Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space.
%E Modifier: use alternative format, see below.
%F Equivalent to %Y-%m-%d (the ISO 8601 date format).
%G The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %V). This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.
%g Like %G, but without century, i.e., with a 2-digit year (00-99).
%h Equivalent to %b.
%H The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I The hour as a decimal number using a 12-hour clock (range 01 to 12).
%j The day of the year as a decimal number (range 001 to 366).
%k The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.)
%l The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.)
%m The month as a decimal number (range 01 to 12).
%M The minute as a decimal number (range 00 to 59).
%n A newline character.
%O Modifier: use alternative format, see below.

%p Either ‘AM’ or “PM’ according to the given time value, or the corresponding strings for the current locale. Noon is treated as ‘pm’ and midnight as ‘am’.
%P Like %p but in lowercase: ‘am’ or ‘pm’ or a corresponding string for the current locale.
%r The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to ‘%I:%M:%S %p’.
%R The time in 24-hour notation (%H:%M). For a version includ- ing the seconds, see %T below.
%s The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.
%S The second as a decimal number (range 00 to 61).
%t A tab character.
%T The time in 24-hour notation (%H:%M:%S).
%u The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w.
%U The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also %V and %W.
%V The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also %U and %W.
%w The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.
%W The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
%x The preferred date representation for the current locale without the time.
%X The preferred time representation for the current locale without the date.
%y The year as a decimal number without a century (range 00 to 99).
%Y The year as a decimal number including the century.
%z The time-zone as hour offset from GMT. Required to emit RFC822-conformant dates (using "%a, %d %b %Y %H:%M:%S %z").
%Z The time zone or name or abbreviation.
%+ The date and time in date(1) format.
%% A literal % character.

Using the above format specifiers edit the program and rerun it with an additional five different ways to format the date and time. The line to be modified in the program is:

msgp = "%A %d-%b-%Y %l:%M %p";

Child PID#	Parent PID#	Request String	Time

Record the information gathered on the next page and just turn in that page for this lab.

Deliverables:

For the report for this lab, complete and turn in the following page.

DUE DATE 11:00am 8 November 2022

CS211
Lab7

Name: _____

Complete the tables below from you data above

s[0]	s[1]	Program Name	I-node #	PID

Child PID#	Parent PID#	Request String	Time

Child PID#	Parent PID#	Request String	Time