CSCI367
Fall 2015
Due October 30th by 11 a.m.
Second Programming Assignment
Bidirectional Data Flow Through Piggy

Piggy-2

Piggy1 only dealt with data flowing in one direction "left to right." Piggy1 created the capability to hook up a chain of piggies and send data from the head to the tail passing through the middle piggies from left to right. In piggy2 we are going to create the capability to transfer data in both directions at once. Data that comes in through the left side will be send out the right side. Data that comes in from the right side will be sent out the left side. In order to accommodate the bidirectional flow of data we need to be able to read from each side without risking blocking. If we try to read the left side and there is no data we would block. While we are blocked there may be massive amounts of data showing up from the right side that needs to get propagated to the left side. The solution to our problem is to use ***select*** to determine if there is data available to be read before actually trying to read it. In addition, in piggy2 we are going to add some interactive commands that are entered from the keyboard. The only way to control piggy1 was using command line parameters. In piggy2 we will have command line parameters to set the piggy into its initial configuration but we will also be able to control it by typing commands at the keyboard. His implies that in piggy2 all piggies read from the keyboard not just a "head" piggy as in piggy1.

The command line parameters piggy2 accepts are as follows.

All command line parameters used in piggy1 are still valid. The piggy1 parameters are included again here for the sake of completeness. Command line parameters can appear *in any order.* All command line parameters will assume we are dealing with the bash shell in Linux. For all commands if a port address is not specified you should use your default port address assigned to you for the course, 367xx, where xx is the number of the book you have been given.

Review of Piggy1 command line parameters

Specifying Address and port for the remote machine to connect to on the right side

```
-rraddr value
```

Read this as ***R***ight ***R***emote ***A***ddress. This is used to specify the address of the right side, i.e. the node we should *connect to*. Value is either and IP address in dotted decimal notation or a DNS name. Note that there is no default address for a right side. A `rraddr` must always be specified unless the `-noright` option is given.

```
-rrport value
```

Read this as ***R***ight ***R***emote ***P***ort. This is used to specify the port address to connect to one the machine on the right. As always, if a port is not given, you should use your default assigned 367xx port number.

```
-noleft
```

This indicates that there will be no incoming left side connection. This is useful to create the "*head*" of a chain of piggy processes from which we want to generate a stream of data. If `-noleft` is given then any `-lraddr` (left remote address) option is ignored. Similarly, if `-noleft` is given then any `-llport` (left local port) option is ignored.

`-noright`

This indicates that there will be no right side that piggy tries to connect to. This is useful to create the "*tail*" of a chain of piggy processes. If `-noright` is given then any `-rraddr` (right remote address) option is ignored. Similarly, if `-noright` is given then any `-rrport` (right remote port) command is ignored.

New Command Line Parameters in Piggy2

`-lraddr value`

Left ***R***emote ***A***ddress. This is used to specify what addresses are valid when *accept*ing a connection from the left side. Value is either and IP address in dotted decimal notation or a DNS name or the character *. Note that * must be quoted as in "*" in order to prevent the shell from thinking it should expand it to a list of the file names in the current directory. I know of now way to apply a "filter" to the addresses when making an accept call. This implies that in order to implement this function you will need to accept the connection, find out where it came from, and then decide if it should be dropped or not. Ultimately, it would be nice to be able to formulate the name or IP address that can connect to the left size as a fully featured regular expression. This is more elaborate than we are going to get in piggy2. The only "regular expression" we allow is the wildcard "*" to specify any address is OK.

Specifying a port address

`-lrport value`

Left remote port. This indicates what source port will be accepted from the left side remote machine. Value is a valid port address which in the case is any 16-bit unsigned integer value, i.e. 0..65535. It may also be the wild-card * indicating any port is acceptable. If this command is not given the default is to accept any valid source port address. Once again the * should be quoted to prevent the shell from attempting to perform file name expansion. Once again it would be nice to eventually be able to specify a full blown regular expression for the source port address used on the incoming connection.

`-llport value`

Left ***L***ocal ***P***ort. This option is used to indicate what port address should be used to listen for the left side connection. The default is to use your assigned port you have been given for the course, i.e. 367xx where xx is the number of the text you have been given.

Piggy2 Data Display Issues

Since we have data flowing in both directions we can't display it all at once in an uncluttered way without subdividing the screen or creating multiple windows. This capability will be added in later versions of piggy but to keep things simple in piggy2 we are going to use a parameter to indicate which direction (left to right stream or right to left stream) to show on the screen. This is accomplished with the `-dsplr` and `-dsprl` parameters.

`-dsplr` display the data flowing from left to right on the screen. This is the default if neither of the display options is specified.
`-dsprl` display the data flowing from right to left on the screen

It is an error to give both parameters. In this case the default of displaying the data flowing from left to right will be used.

Interactions of the display parameter and the `-noleft` and `-noright` parameters

If the `-noleft` parameter is given then obviously the only data we can display is flowing in from the right. If `-noleft` is given then display direction parameters are ignored and the data flowing in from the right is displayed.

Similarly, if the `-noright` parameter is given then obviously the only data we can display is flowing in from the left. If `-noright` is given then display direction parameters are ignored and the data flowing in from the left is displayed. In other words, the `-dsplr` and `-dsplr` commands only make sense when it is a "middle" piggy and has data flowing through it in both directions. They are used to select which stream of data to display.

Cycling Data Back Toward Where it Came From

Eventually in later piggies we will add a capability to perform some processing on the data as it move through from one side to the other. When we have that capability it will be important in a "tail" piggy or "head" to be able to process the data stream and then send it back in the other direction. We introduce the "loop" command in piggy2 to provide this capabilty.

`-loopr`

This parameter causes piggy2 to to take any data that would be written out to the ***R***ight side and inject it into the data stream leaving the let side. If the piggy is a tail, i.e. has no right side connection, loopr the data that arrives from the left is "looper back," i.e. written out the left side.

`-loopl`

This parameter causes piggy2 to take the data that would be written out to the ***L***eft side and inject it into the data stream leaving the right side. If the piggy is a "head" (-noleft) then -loopl causes it to bounce the incoming data stream back out to the right.

Example invocations and their interpretation

```
example 1

piggy2 -lraddr 140.160.140.5 -rraddr 140.160.140.5 -dsprl
```

This command would cause piggy to accept a connection on the left side only if it originates from a computer with the IP address 140.160.140.5. Additionally the program would try to make a connection to the node with IP address 140.160.140.5. The incoming connection would accept any source port. The incoming, left side, connection would listen on the default 367xx port. The outgoing connection would try to connect to the default 367xx port address on the destination computer with IP address 140.160.140.5. The program would show the data moving from the right side to the left side. If the `-dsprl` parameter had been omitted the default would have been to show the data moving from left to right.

```
example 2
piggy1 -lraddr bucky.cs.wwu.edu -rraddr 140.160.140.70
```

This command would cause piggy to accept a connection form the node with DNS name bucky.cs.wwu.edu and connect to a node with an IP address of 140.160.140.70. The incoming connection would listen at the default 367xx port address accepting any source port from the connecting computer. The outgoing connection would use whatever local port the OS issued and attempt a connection to the default 367xx port address on the destination computer 140.160.140.70. The data flowing from left to right would be displayed on the terminal as this is the default in the absence of a display directive.

```
example 3

piggy1 -lraddr “*”  -rraddr bucky.cs.wwu.edu -dsprl
piggy1 -rraddr bucky -dsprl
```

Both of these commands would accept a connection on the “left side” from any IP address with any source port. Note that `-lraddr “*”` is never actually required to be specified as this is the default action. Note , however, that if the `-noleft` option is given then any `-lraddr` option is ignored so strictly speaking * is the default in the absence or a `-noleft` option. Piggy would make a connection to a node with the DNS name bucky. Assuming that the default DNS domain for the computer executing the command is cs.wwu.edu each command would attempt to connect to bucky.cs.wwu.edu. The incoming connection would accept any source port. The outgoing connection would use the default port address. In each case the data displayed would be the data flowing from right to left.

```
Example 4
piggy1  -dsplr
piggy1 -noleft -noright -dsprl
```

Both of these commands would produce an error message indicating that you must have at least one of either a left or a right side address. In piggy2 you can have a left side connection, a right side connection, or both but you can't have neither.

```
Example 5
```

```
piggy1 -llport 26799 -lrport 36799 -raddr bucky
```

This command would accept a left side connection from any IP address as long as the source port address from the connecting machine was 36799. It would use port address 26799 as its local port address for the left side connection. It would try to make a right side connection to the node named `bucky` on the default DNS domain using the default port at the remote right side machine.

Keyboard Input

In piggy1 we intentionally designed things so that we could ignore any issues related to blocking and the need to deal with multiple streams of input flowing into the program and through the program. The only time we read the keyboard was when `-noleft` was given. In piggy2 when -`noleft` is given we display what is typed on the keyboard and send it out the right side connection. When `-noright` is given we display what arrives from the left side connection. The `-dsplr` and `-dsprl` parameters are ignored when `-noleft` or -`noright` is given as there is only one stream to display. In piggy2 we always read from the keyboard.

Keyboard Commands

In piggy1 the characters typed at the keyboard when a `-noleft` command line parameter had been given were sent to the outgoing right side connection. In piggy2 we always read the keyboard regardless whether we have a left side only, a right side only, or both. The default action is to take data the is typed into the keyboard and send in to the current "output" direction. The -noleft command now implies that the default "output" is to the right. The -noright command now implies that the default "output" is to the left. If neither a -noleft or a -noright is given the default "output" is to the right.

In addition to reading keyboard input for the purpose of injecting data into the data stream in the current "output" direction in piggy2 the keyboard is used to issue commands to the program interactively. In piggy1 we had no "clean" way of terminating the programs. We simply used CTRL-C to "ungracefully" kill the process. Commands are given to piggy from the keyboard in a way that mimics the behavior of the "insert text versus issue a command" conventions of the vi text editor. In vi to enter insert mode and type text into the document you are editing you enter the "i" command. Then anything you type is inserted into the document until you hit the escape key which returns you to command mode. Piggy2 will handle keyboard input using the same scheme. As the piggies evolve we are going to keep adding interactive commands to be able to control the piggy's behavior in various ways. When we wish to inject data, we use the "i" command. Now, anything we type goes to the default output direction until we hit the escape key to return to command mode.

Interactive Piggy2 commands

You will notice that these command mirror the command line parameters.

`i`	enter insert mode
`esc`	exit insert mode and return to command mode. Esc entered while in command mode is ignored.
`:outputl`	set the output direction to left
`:outputr`	set the output direction to right

`:output`	show what direction the output is set to
`:dsplr`	display the left to right data stream on the screen
`:dsprl`	display the right to left data stream on the screen
`:dsp`	show what direction the display is set to. Write display= either lr or rl
`:dropright`	drop the right side connection
`:dropleft`	drop the left side connection
`:right`	show information about the right side connection*
`:left`	show information about the left side connection*
`:lraddr`	show the currently connected left side address
`:rraddr`	show the currently connected right side addresses
`:loopr`	causes piggy2 to take the data that would be written to the right side and inject it into the data stream flowing to the left side
`:loopl`	causes piggy2 to take the data that would be written to the left side and inject it into the data stream flowing to the right side
`:q`	terminate the program

* The information is printed as local IP:local port:remote IP:remote port

Errors

In piggy1 when we encountered an error we just terminated the program. From now on we write out an error message but do not terminate the program as we have interactive commands to control the program now. Any error in a command line parameter should be caught and an appropriate error message written to the screen. Any command given to piggy2 in command mode should be caught and an appropriate error message given. If you try to make a connection and fail print an error message.

Using select to deal with multiple sources of input and avoiding blocking

In all cases piggy2 is started with either two or three sources of input. *One input source is always the keyboard.* The others are a left or right side connection or both. We cannot read the keyboard to read a command from the user and block while there is a possibly large stream of input coming in from the left or right side. We need a way that we can tell when there is input available to be read so that when we read we always get something and never block waiting for input to arrive. This is what the `select` function will do for us. There are many ways to use select. I will describe the simple way we wish to utilize it and then discuss some of its other features. We wish to be able to see if there is input available from any of the sources we need to read. In our case that is the keyboard and the left and right sides (we will assume we have both sides connected for this example). When you call select, you give it a set of file descriptors, in our case the three described above. When we return from select it indicates which of the file descriptors we gave it have input available. We can then safely read from those file descriptors knowing that we will receive data and not be blocked waiting for input to arrive.

The select function actually takes three sets of file descriptors: a read set; a write set; and an error set. In piggy2 we need only concern ourselves with the read set. The other two can be "nulled out." In addition the last parameter to select allows us to supply a time value. If there are no descriptors that are "ready" select will return after waiting for the timeout period. This gives us the last piece of the puzzle we need to be able to ensure that we never block indefinitely waiting on I/O.

To see a sample of a server using select to multiplex I/O refer to the tcpcliserv/tcpserverselect01.c file in the examples you downloaded from the Stevens UNIX Network Programming book.

In C the select function looks like this.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);
```

The manual pages from section 2 of the Linux Programmers Manual for select have been included at the bottom of this document for your convenience.

Screen Output

Piggy2 is designed so that we do not need to deal with "splitting the screen" or opening multiple windows from a GUI. The underlying objective of piggy2 is to get you dealing with multiple sources of input using `select`. In later versions we will add split screen display output.

NAME

select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing

SYNOPSIS

```
/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);

#include <sys/select.h>

int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
pselect(): _POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE >= 600
```

DESCRIPTION

select() and pselect() allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., read(2)) without blocking.

The operation of select() and pselect() is identical, other than these three differences:

(i) select() uses a timeout that is a struct timeval (with seconds and microseconds), while pselect() uses a struct timespec (with seconds and nanoseconds).

(ii) select() may update the timeout argument to indicate how much time was left. pselect() does not change this argument.

(iii) select() has no sigmask argument, and behaves as pselect() called with NULL sigmask.

Three independent sets of file descriptors are watched. Those listed in readfds will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of-file), those in writefds will be watched to see if a write will not block, and those in exceptfds will be watched for exceptions. On exit, the sets are modified in place to indicate which file descriptors actually changed status. Each of the three file descriptor sets may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Four macros are provided to manipulate the sets. FD_ZERO() clears a set. FD_SET() and FD_CLR() respectively add and remove a given file descriptor from a set. FD_ISSET() tests to see if a file descriptor is part of the set; this is useful after select() returns.

nfds is the highest-numbered file descriptor in any of the three sets, plus 1.

The timeout argument specifies the interval that select() should block waiting for a file descriptor to become ready. This interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. If both fields of the timeval structure are zero, then select() returns immediately. (This is useful for polling.) If timeout is NULL (no timeout), select() can block indefinitely.

sigmask is a pointer to a signal mask (see sigprocmask(2)); if it is not NULL, then pselect() first replaces the current signal mask by the one pointed to by sigmask, then does the "select" function, and then restores the original signal mask.

Other than the difference in the precision of the timeout argument, the following pselect() call:

```
    ready = pselect(nfds, &readfds, &writefds, &exceptfds,
                    timeout, &sigmask);
```

is equivalent to atomically executing the following calls:

```
    sigset_t origmask;

    pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
    ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
    pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The reason that pselect() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of select() could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, pselect() allows one to first block signals, handle the signals that have come in, then call pselect() with the desired sigmask, avoiding the race.)

The timeout

The time structures involved are defined in <sys/time.h> and look like

```
    struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* microseconds */
    };
```

and

```
    struct timespec {
        long    tv_sec;         /* seconds */
        long    tv_nsec;        /* nanoseconds */
    };
```

(However, see below on the POSIX.1-2001 versions.)

Some code calls select() with all three sets empty, nfds zero, and a non-NULL timeout as a fairly portable way to sleep with subsecond precision.

On Linux, select() modifies timeout to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1-2001 permits either behavior.) This causes problems both when Linux code which reads timeout is ported to other operating systems, and when code is ported to Linux that reuses a struct timeval for multiple select()s in a loop without reinitializing it. Consider timeout to be undefined after select() returns.

RETURN VALUE

On success, select() and pselect() return the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in readfds, writefds, exceptfds) which may be zero if the timeout expires before anything interesting happens. On error, -1 is returned, and errno is set appropriately; the sets and timeout become undefined, so do not rely on their contents after an error.

ERRORS

EBADF An invalid file descriptor was given in one of the sets. (Perhaps a file descriptor that was already closed, or one on which an error has occurred.)

EINTR A signal was caught; see signal(7).

EINVAL nfds is negative or the value contained within timeout is invalid.

ENOMEM unable to allocate memory for internal tables.

VERSIONS

pselect() was added to Linux in kernel 2.6.16. Prior to this, pselect() was emulated in glibc (but see BUGS).

CONFORMING TO

select() conforms to POSIX.1-2001 and 4.4BSD (select() first appeared in 4.2BSD). Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants). However, note that the System V variant typically sets the timeout variable before exit, but the BSD variant does not.

pselect() is defined in POSIX.1g, and in POSIX.1-2001.

NOTES

An fd_set is a fixed size buffer. Executing FD_CLR() or FD_SET() with a value of fd that is negative or is equal to or larger than FD_SETSIZE will result in undefined behavior. Moreover, POSIX requires fd to be a valid file descriptor.

Concerning the types involved, the classical situation is that the two fields of a timeval structure are typed as long (as shown above), and the structure is defined in <sys/time.h>. The POSIX.1-2001 situation is

```
    struct timeval {
        time_t         tv_sec;     /* seconds */
        suseconds_t    tv_usec;    /* microseconds */
    };
```

where the structure is defined in <sys/select.h> and the data types time_t and suseconds_t are defined in <sys/types.h>.

Concerning prototypes, the classical situation is that one should include <time.h> for select(). The POSIX.1-2001 situation is that one should include <sys/select.h> for select() and pselect().

Libc4 and libc5 do not have a <sys/select.h> header; under glibc 2.0 and later this header exists. Under glibc 2.0 it unconditionally gives the wrong prototype for pselect(). Under glibc 2.1 to 2.2.1 it gives pselect() when _GNU_SOURCE is defined. Since glibc 2.2.2 the requirements are as shown in the SYNOPSIS.

Multithreaded applications

If a file descriptor being monitored by select() is closed in another thread, the result is unspecified. On some UNIX systems, select() unblocks and returns, with an indication that the file descriptor is ready (a subsequent I/O operation will likely fail with an error, unless another the file descriptor reopened between the time select() returned and the I/O operations was performed). On Linux (and some other systems), closing the file descriptor in another thread has no effect on select(). In summary, any application that relies on a particular behavior in this scenario must be considered buggy.

Linux notes

The pselect() interface described in this page is implemented by glibc. The underlying Linux system call is named pselect6(). This system call has somewhat different behavior from the glibc wrapper function.

The Linux pselect6() system call modifies its timeout argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call. Thus, the glibc pselect() function does not modify its timeout argument; this is the behavior required by POSIX.1-2001.

The final argument of the pselect6() system call is not a sigset_t * pointer, but is instead a structure of the form:

```
    struct {
        const sigset_t *ss;     /* Pointer to signal set */
        size_t          ss_len; /* Size (in bytes) of object pointed
                                   to by 'ss' */
    };
```

This allows the system call to obtain both a pointer to the signal set and its size, while allowing for the fact that most architectures support a maximum of 6 arguments to a system call.

BUGS

Glibc 2.0 provided a version of pselect() that did not take a sigmask argument.

Starting with version 2.1, glibc provided an emulation of pselect() that was implemented using sigprocmask(2) and select(). This implementation remained vulnerable to the very race condition that pselect() was designed to prevent. Modern versions of glibc use the (race-free) pselect() system call on kernels where it is provided.

On systems that lack pselect(), reliable (and more portable) signal trapping can be achieved using the self-pipe trick. In this technique, a signal handler writes a byte to a pipe whose other end is monitored by select() in the main program. (To avoid possibly blocking when writing to a pipe that may be full or reading from a pipe that may be empty, nonblocking I/O is used when reading from and writing to the pipe.)

Under Linux, select() may report a socket file descriptor as "ready for reading", while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use O_NONBLOCK on sockets that should not block.

On Linux, select() also modifies timeout if the call is interrupted by a signal handler (i.e., the EINTR error return). This is not permitted by POSIX.1-2001. The Linux pselect() system call has the same behavior, but the glibc wrapper hides this behavior by internally copying the timeout to a local variable and passing that variable to the system call.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int
main(void)
{
```

```
           fd_set rfds;
           struct timeval tv;
           int retval;

           /* Watch stdin (fd 0) to see when it has input. */
           FD_ZERO(&rfds);
           FD_SET(0, &rfds);

           /* Wait up to five seconds. */
           tv.tv_sec = 5;
           tv.tv_usec = 0;

           retval = select(1, &rfds, NULL, NULL, &tv);
           /* Don't rely on the value of tv now! */

           if (retval == -1)
               perror("select()");
           else if (retval)
               printf("Data is available now.\n");
               /* FD_ISSET(0, &rfds) will be true. */
           else
               printf("No data within five seconds.\n");

           exit(EXIT_SUCCESS);
       }
```

SEE ALSO

accept(2), connect(2), poll(2), read(2), recv(2), send(2), sigprocmask(2), write(2), epoll(7), time(7)

For a tutorial with discussion and examples, see select_tut(2).

COLOPHON

This page is part of release 3.54 of the Linux man-pages project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.