



issue:  
https://github.com/golang/go/issues/71424

bufio.Reader empty write issue

If the reader reads the input stream of data (example from stdin), it function as expected. But when the reader is wrapped in an bufio.Reader there is an extra empty write being made. In client-server interation using unix scoekts, this results into EOF and server considers this to be an EOF and closes before the actual writes are made.

**Step:1**

windows powershell:  
wsl (ubuntu)

How to install wsl( windows subsystem for Linux)  
wsl –list –online (lists available linux distributions)  
**wsl –install Ubuntu-20.04**

How to bring up ubuntu linux?  
wsl  
aarithi/aarithi

**Step 2**:  
How to install latest golang?  
<https://go.dev/doc/install> (instructions to install golang on ubuntu)

sudo su – (to switch to root user)  
password: aarithi   
  
wget <https://go.dev/dl/go1.23.5.linux-amd64.tar.gz>  
rm -rf /usr/local/go && tar -C /usr/local -xzf go1.23.5.linux-amd64.tar.gz  
export PATH=$PATH:/usr/local/go/bin

go version

This version has the bug and we should able to produce the reported issue.

**Step 3**:

Produce the reported issue   
  
cd /mnt/c/users/aarti/open-source-project  
copy the main.go which is used to report the issue @ <https://github.com/golang/go/issues/71424>

Compile the code:  
go build main.go  
  
Testing:  
success scenario:  
read input data stream using the reader:  
./main <<< “Test”  
./main | cat test-file

Failure scenario:  
read input data with bufio wrapped reader:

./main -bufio <<< “Test”  
server got EOF

How did you troubleshoot the issue?  
By going through the main.go server-client interaction, I found the statements where it is reading the input data stream with reader.  
  
var reader io.Reader = os.Stdin

if \*buffered {  
 reader = **bufio.NewReader**(os.Stdin)  
}  
\_, err = io.Copy(conn, reader)  
if err != nil {  
 log.Fatalln(err)  
}

After adding DEBUG statement and spend time in troubleshooting, I found that the issue with bufio package and the code which manon.go imports and bufio.go which has Reader struct and Reader interface. bufio routine implements the reader interface.

<https://github.com/golang/go/blob/master/src/bufio/bufio.go>

type Reader struct {  
 buf []byte  
 rd io.Reader // reader provided by the client  
 r, w int // buf read and write positions  
 err error  
 lastByte int // last byte read for UnreadByte; -1 means invalid  
 lastRuneSize int // size of last rune read for UnreadRune; -1 means invalid  
}

buf

r

w

// NewReader returns a new [Reader] whose buffer has the default size.  
func NewReader(rd io.Reader) \*Reader {  
 return NewReaderSize(rd, defaultBufSize)  
}  
  
// writeBuf writes the [Reader]'s buffer to the writer.  
func (b \*Reader) writeBuf(w io.Writer) (int64, error) {  
 n, err := w.Write(b.buf[b.r:b.w])  
 if n < 0 {  
 panic(errNegativeWrite)  
 }  
 b.r += n

return int64(n), err

}

The issue was identified in go/src/bufio/bufio.go. I made the following changes to address the problem:

1. **Added the** isEmpty() **Function**: This function returns true if the buffer in the Reader struct is empty, and false otherwise.

// Size returns the size of the underlying buffer in bytes.  
func (b \*Reader) Size() int { return len(b.buf) }

**//bufio issue 71424 - Aarithi Rajendren  
// Checks if the buffer is empty.**

**func (b \*Reader) isEmpty() bool {  
 tmpBuff := b.buf[b.r:b.w]  
 return len(tmpBuff) == 0  
}**

// Reset discards any buffered data, resets all state, and switches  
// the buffered reader to read from r.  
// Calling Reset on the zero value of [Reader] initializes the internal buffer  
// to the default size.  
// Calling b.Reset(b) (that is, resetting a [Reader] to itself) does nothing.

func (b \*Reader) Reset(r io.Reader) {  
 // If a Reader r is passed to NewReader, NewReader will return r.  
 // Different layers of code may do that, and then later pass r  
 // to Reset. Avoid infinite recursion in that case.

if b == r {  
 return  
 }

if b.buf == nil {  
 b.buf = make([]byte, defaultBufSize)  
 }  
 b.reset(b.buf, r)  
}

1. **Modified the** writeBuf **Function**: In the same file (bufio.go), I added a conditional check to see if the buffer b is empty using the newly added isEmpty method. If the buffer is empty, the function returns a tuple (0, nil). Otherwise, it writes a portion of the buffer from the Reader struct to an io.Writer and then updates the reader's position.

// writeBuf writes the [Reader]'s buffer to the writer.  
func (b \*Reader) writeBuf(w io.Writer) (int64, error) {

**//bufio issue 71424 - Aarithi Rajendren  
    if b.isEmpty() {  
        return 0, nil  
    }**

    n, err := w.Write(b.buf[b.r:b.w])  
    if n < 0 {  
        panic(errNegativeWrite)  
    }

    b.r += n  
    return int64(n), err  
}

These changes resolve the bufio.Reader empty write issue during io.Copy(). I tested the modifications with the main.c file, and it worked as expected.

**Step 4:**

Code contribution to golang is documented at <https://go.dev/doc/contribute#before_contributing>

sudo su – (switch to root user)  
password: aarithi  
  
cd /mnt/c/user/aarti/opens-source-project  
git clone <https://go.googlesource.com/go>  
cd go  
git checkout -b dev.bufio.issue.71424  
cd go/src  
  
make the changes outlines it step 3 bufio/bufio.go  
  
export PATH=$PATH:/usr/local/go/bin (before compiling the dev go code, set the default go version to Go 1.22.6 or above)

$ ./all.bash # compile and test  
export PATH=/mnt/c/Downloads/open-source-project/go/bin:$PATH (dev version of go)  
  
**step 5:**

Testing the code with above changes:  
success scenario:  
read input data stream using the reader:  
./main <<< “Test”  
./main | cat test-file

Earlier failure scenario is now successful:  
read input data with bufio wrapped reader:

./main -bufio <<< “Test”  
server got EOF

Step 6:  
contribute the code:

git codereview mail

This Go code demonstrates a simple server-client interaction using Unix domain sockets. The program imports necessary packages including bufio, flag, fmt, io, log, net, and os, which provide functionalities such as reading from standard input, command line flag parsing, and network communication.

It uses a socket file located at "/tmp/testsock123" and can operate in two modes, either using buffered input with bufio or without. The main function initializes a command-line flag -bufio to determine whether buffered I/O is used. If the socket file already exists, it is removed to ensure the server can bind to the address.

Flag Handling: The program uses a flag to determine whether to use buffered input (-bufio). This flag is stored in the buffered variable.

Socket Creation: The program removes any existing socket file at /tmp/testsock123. It then creates a Unix domain socket of type "unixpacket" (reliable, message-oriented). If you switch to type "unix", it changes to a stream-oriented socket.

Server Routine:

The program creates a Unix domain socket using net.ListenUnix with unixpacket as the protocol type. A goroutine is launched to handle server operations. Inside the goroutine, the server waits for a connection using `Accept()`. Once a connection is established, it reads data from the connection in a loop, printing the number of bytes read and the message content to the standard output. If it encounters an io.EOF (end of file), it prints "server got EOF" and ends the goroutine via a channel.

Client Connection:

A connection is established to the server using net.DialUnix. The program reads input from `os.Stdin` ((standard input)) and sends it through the socket to the server. Depending on the command-line flag, it can use either buffered or unbuffered input for reading. The io.Copy function is responsible for transferring data from the input stream to the connection.

After copying input data to the socket, the connection is closed and the program waits for the server goroutine to signal completion. This program demonstrates creating a Unix domain socket listener, accepting connections, reading data, and handling client-server communication within Go.