

Intro to Berkeley Sockets in x86 ASM Under Linux

Many thanks to the folks who helped me get this stuff figured out. It was a long and frustrating process for me, so I'm writing this guide to hopefully spare some of you a similarly frustrating experience.

Whenever I try to talk with folk about programming in assembly language, reactions tend to range from reluctance to hostility. I understand the appeal of high level languages, and have no qualms with folk who choose to write software in them. I happen to prefer the directness of assembly language - it's the first language I learned, and I've never lost my love of it in spite of a number of years coding professionally in a variety of high level languages. I imagine most assembly language people have similar stories. This guide is for folks who for their own reasons want to bypass the established methods of implementing Berkeley Sockets in high level languages.

Additionally, this article is specific to GNU/Linux and makes use of the `socketcall()` kernel function 066h. Code snippets found in this document will NOT work under Windows, and this article assumes a passing familiarity with the Linux kernel and x86 assembly language. If you don't know what I'm talking about when I say 'Linux kernel', then this guide won't really make much sense to you. Educating you on the particulars of interacting with the Linux kernel is beyond the scope of this document: there are a TON of resources available on the internet for folk who want to learn about Linux, and I encourage you to avail yourself of them in accordance with your degree of interest in the topic matter.

The examples in this article are written in Intel syntax, and I've been assembling them with NASM 0.99.06-20071101.

DISCLAIMER: X86 assembly language is very powerful, and bypasses many of the controls and user protections found in higher level languages. It is not only possible, but *easy* to *really* screw up your system with assembly language if you don't know what you're doing. The reader is advised to keep this in mind, code carefully, and proceed with caution.

And now, down to business.

The Berkeley Sockets API is a synchronous I/O interface commonly used to facilitate communications between processes, be they threads within the same program, processes belonging to different programs running within the same core, or different programs executing on separate cores. The API accomplished this task by providing an abstraction layer for communication which is referred to as a 'socket'. Sockets are directly and indirectly manipulated through the use of associated descriptors provided by the operating system.

What we're going to deal with in this article (by way of building a simple server) is the client-server relationship, which can be extrapolated to communication between threads as well.

Let's look at a simple server program. The steps to getting a simple server running on Berkeley Sockets follow:

1. Create a socket.
2. Bind the socket to a port on the host machine.
3. Tell the socket to listen for incoming connections on the bound port.
4. Setup a polling loop to tell the server when someone's knocking at the door.
5. Arrange for the program to respond to the knock by accepting the incoming connection.

And that's about it. Once that's done, you've got the skeleton of a network server. From there you can shape your server to do whatever you want it to. It's relatively simple in concept, but the implementation involves making sure a lot of specific data gets plugged into the right spots at the right time.

The following listings are CODE FRAGMENTS, and do not terminate, which means if you assemble them and try to execute them on their own, they'll execute right into whatever's sitting in memory after they finish, instead of passing control back to the terminal. This usually results in a segmentation fault, but can potentially damage your system. These listings are not meant to be executed on their own and are here to demonstrate process.

Here's how you create the socket:

```
=====
;
;
; socket.create
;
; CODE FRAGMENT: This code can potentially BORK your system (although it'll more likely just segfault) if assembled and executed as is.
; This code outlines the steps involved in building a socket and is not intended to be executed by itself.

section .bss

%define sys.socket.call      0066h    ; kernel service number 066h corresponds to the Berkeley Sockets API
%define sys.socket.create    0001h    ; Socket service 001h corresponds to socket creation
%define sys.system.call      0080h    ; int 080h is how we access kernel services in Linux

%define sys.socket.protocol.family.inet  0002h    ; you may know this parameter as PF_INET
%define sys.socket.type.stream          0001h    ; and this one as SOCK_STREAM
%define sys.socket.address               0000h    ; and this one as INADDR_ANY

socket.data.protocol.family    resd    0001h    ; storage for protocol family, in this instance PF_INET
socket.data.flags              resd    0001h    ; storage for communication semantics, in this instance SOCK_STREAM
socket.data.address            resd    0001h    ; storage for address, in this instance INADDR_ANY

section .text

global socket.create

socket.create:    mov     eax, sys.socket.protocol.family.inet    ; eax - protocol family
                 mov     [socket.data.protocol.family], eax    ; store protocol family in arguments package for the socket creation
                 ; function
                 mov     eax, sys.socket.type.stream          ; eax - flags - socket type
                 mov     [socket.data.flags], eax              ; store flags in arguments package for the socket creation function
                 mov     eax, sys.socket.address               ; eax - address for socket
                 mov     [socket.data.address], eax            ; store address in arguments package for the socket creation function

                 mov     eax, sys.socket.call                  ; eax - kernel service 066h: Berkeley Sockets API
```

```

mov     ebx, sys.socket.create           ; ebx - socket service 001h: create socket
mov     ecx, socket.data.protocol.family ; ecx - pointer to arguments package for socket service 001h
int     sys.system.call                 ; create that socket!

```

```

;=====
The interrupt will return one of two things in the eax register. If eax is positive, what you have there is the descriptor associated with your new socket. If eax is negative, that's an error code. The inverse of the error codes returned in eax can be found in /usr/include/asm-generic in the files 'errno.h' and 'errno-base.h'.

```

Assuming that the call returns without error, the next step is to bind the socket to a port. The following code assumes eax contains the descriptor returned during socket creation:

```

;=====
;
; socket.bind
;
; assumes eax contains a valid socket descriptor
;
; CODE FRAGMENT: This code can potentially BORK your system (although it'll more likely just segfault) if assembled and executed as is.
; This code outlines the steps involved in binding a socket to a port and is not intended to be executed by itself.

```

section .bss

```

#define sys.socket.call           0066h ; kernel service number 066h corresponds to the Berkeley Sockets API
#define sys.socket.bind          0002h ; Socket service 002h corresponds to port binding
#define sys.system.call          0080h ; int 080h is how we access kernel services in Linux

#define sys.socket.protocol.family.inet 0002h ; you may know this parameter as PF_INET
#define sys.socket.port.number         280ah ; port number to be bound to socket - stored in little endian byte order
#define sys.socket.address             0000h ; you may know this parameter as INADDR_ANY
#define sys.socket.padding            0000h ; padding for sockaddr structure

```

section .data

```

socket.data.sock.descriptor      dd      0000h ; storage for descriptor associated with socket
                                   ; to be bound
socket.data.pointer.sockaddr     dd      socket.data.sockaddr.protocol.family ; storage for pointer to sockaddr data
socket.data.length.sockaddr     dd      socket.data.sockaddr.length ; storage for length of sockaddr data
socket.data.sockaddr.protocol.family dw    sys.socket.protocol.family.inet ; sockaddr element for storing the protocol
                                   ; family, in this instance PF_INET
socket.data.sockaddr.port.number dw      sys.socket.port.number ; sockaddr element for storing the port number in
                                   ; little endian order
socket.data.sockaddr.address     dd      sys.socket.address ; sockaddr element for storing the address, in
                                   ; this instance INADDR_ANY
socket.data.sockaddr.padding     dq      sys.socket.padding ; padded element
socket.data.sockaddr.length     equ     $-socket.data.sockaddr.protocol.family ; length of sockaddr structure

```

section .text

global socket.bind

```

socket.bind:    mov     [socket.data.sock.descriptor], eax ; store descriptor associated with socket in arguments package for
                                                         ; the bind function
                                                         ; eax - kernel service 066h: Berkeley Sockets API
                                                         ; ebx - socket service 002h: bind socket to port
                                                         ; ecx, socket.data.sock.descriptor ; pointer to arguments package for socket service 002h
                                                         ; int     sys.system.call         ; bind that socket!

```

```

;=====
The bind function will return zero in eax on success, otherwise it will return as error code in the same format as the socket creation service. Note that the port number is stored in little endian byte order. This is an important point to remember, as if you tell a socket to listen on a port stored in big endian byte order, it's not going to be listening on the port you think it is, which can be frustrating.

```

Assuming all has gone well so far, the next step is to tell the socket to listen for incoming connections. So far, this is the easiest step as it only requires two parameters; the descriptor associated with the socket, and the number of pending connections to allow to pile up before it stops queueing them for acceptance. This code assumes that eax contains a descriptor associated with an active, bound socket:

```

;=====
;
; socket.listen
;
; assumes eax contains a valid descriptor associated with an active, bound socket
;
; CODE FRAGMENT: This code can potentially BORK your system (although it'll more likely just segfault) if assembled and executed as is.
; This code outlines the steps involved in telling a bound socket to listen for incoming connections and is not intended to be executed by itself.

```

section .bss

```

#define sys.socket.call           0066h ; kernel service number 066h corresponds to the Berkeley Sockets API
#define sys.socket.listen        0004h ; Socket service 004h corresponds to telling a socket to listen for incoming
                                   ; connections
#define sys.system.call          0080h ; int 080h is how we access kernel services in Linux

#define sys.socket.queue.length  0005h ; for the sake of this article we'll let 005h connections pile up

```

```

socket.data.sock.descriptor  resd    0001h    ; storage for descriptor associated with bound, active socket
socket.data.queue.length    resd    0001h    ; storage for connection queue length

```

```
section .text
```

```
global socket.listen
```

```

socket.listen:    mov     [socket.data.sock.descriptor], eax    ; store descriptor associated with socket in arguments package for
                 ; the listen function
                 mov     eax, sys.socket.queue.length      ; eax - queue length for socket
                 mov     [socket.data.queue.length], eax  ; store queue length in arguments package for the listen function
                 mov     eax, sys.socket.call              ; eax - kernel service 066h: Berkeley Sockets API
                 mov     ebx, sys.socket.listen           ; ebx - socket service 004h: tell socket to listen
                 mov     ecx, socket.data.sock.descriptor ; pointer to arguments package for socket service 004h
                 int     sys.system.call                  ; tell that socket to listen!

```

```

;=====
Again, the listen function will either return an error or zero.

```

Now you've got a functioning socket listening for incoming connections. However, it's not going to answer the door on it's own - we've got to set up a loop to periodically check and see if the socket has queued up incoming connections, and if so, answer them. For this purpose we're going to go outside of the Berkeley Sockets API and use the `sys_poll` kernel function to check the connection, and accept if it returns a data event on the listening socket:

```

;=====
;
; socket.poll.and.accept
;
; assumes eax contains a valid descriptor associated with an active, bound, and listening socket
;
; CODE FRAGMENT: This code can potentially BORK your system (although it'll more likely just segfault) if assembled and executed as is.
; This code outlines the steps involved in polling a bound and listening socket for incoming connections and accepting them, and is not
; intended to be executed by itself.

```

```
section .bss
```

```

#define sys.socket.call      0066h    ; kernel service number 066h corresponds to the Berkeley Sockets API
#define sys.poll.call       00a8h    ; kernel service number 0a8h corresponds to descriptor polling
#define sys.poll.in         0001h    ; indicator for data on the socket
#define sys.poll.timeout    0010h    ; timeout for polling function
#define sys.poll.number.of.structures 0001h ; number of event structures to poll for

```

```

#define sys.socket.accept   0005h    ; Socket service 005h corresponds to accepting incoming connections on a
                                   ; listening socket
#define sys.system.call     0080h    ; int 80h is how we access kernel services in Linux

```

```

#define sys.socket.protocol.family.inet 0002h ; you may know this parameter as PF_INET
#define sys.socket.port.number          280ah ; port number bound to socket - stored in little endian byte order
#define sys.socket.address              0000h ; you may know this parameter as INADDR_ANY
#define sys.socket.padding              0000h ; padding for sockaddr structure

```

```

socket.data.connected.sock.descriptor  resd    0001h    ; storage for descriptor associated with accepted socket

```

```
section .data
```

```

socket.data.event.sock.descriptor  dd     0000h    ; storage for descriptor associated with bound, active, and listening
                                   ; socket
socket.data.event.requested        dd     sys.poll.in ; storage for requested event(s)
socket.data.event.returned         dd     0000h    ; storage for returned event(s)
socket.data.accept.sock.descriptor dd     0000h    ; storage for descriptor associated with bound, active, and listening
                                   ; socket
socket.data.accept.sockaddr.pointer dd     socket.data.sockaddr.protocol.family ; storage for pointer to sockaddr structure
socket.data.accept.buffer.pointer   dd     socket.data.connected.sock.descriptor ; storage for pointer to storage for descriptor
                                   ; returned by accept function

socket.data.sockaddr.protocol.family dw     sys.socket.protocol.family.inet ; sockaddr element for storing the protocol family, in
                                   ; this instance PF_INET
socket.data.sockaddr.port.number    dw     sys.socket.port.number ; sockaddr element for storing the port number in little endian
                                   ; order
socket.data.sockaddr.address        dd     sys.socket.address ; sockaddr element for storing the address, in this instance
                                   ; INADDR_ANY
socket.data.sockaddr.padding        dq     sys.socket.padding ; padded element
socket.data.sockaddr.length         equ     $-socket.data.sockaddr.protocol.family ; length of sockaddr structure

```

```
section .text
```

```
global socket.poll.and.accept:
```

```

socket.poll.and.accept:    mov     [socket.data.event.sock.descriptor], eax    ; store descriptor associated with listening
                                   ; socket in arguments package for the poll
                                   ; function
                 mov     [socket.data.accept.sock.descriptor], eax    ; store descriptor associated with listening
                                   ; socket in arguments package for the accept
                                   ; function

```

```

socket.poll.and.accept.loop: mov     eax, sys.poll.call           ; eax - kernel service 0a8h: poll descriptor
                             mov     ebx, socket.data.event.sock.descriptor ; ebx - pointer to arguments package for kernel
                             ; service 0a8h
                             mov     ecx, sys.poll.number.of.structures ; ecx - number of event structures in arguments
                             ; package
                             mov     edx, sys.poll.timeout           ; edx - timeout for poll function
                             int     sys.system.call                 ; poll that socket!

                             cmp     eax, sys.poll.in                ; is there a connection waiting?
                             jnz     socket.poll.and.accept.loop      ; nope - go back and poll again

                             mov     eax, sys.socket.call            ; eax - kernel service 066h: Berkeley Sockets API
                             mov     ebx, sys.socket.accept          ; ebx - socket service 005h: accept incoming
                             ; connection
                             mov     ecx, socket.data.accept.sock.descriptor ; ecx - pointer to arguments package for socket
                             ; service 005h
                             int     sys.system.call                 ; accept that connection!

```

=====
The accept function returns one of two things in eax; either an error message which will show up as the inverse of the errors in errno.h and errno-base.h, or a brand new descriptor which corresponds to the connected socket. This is the descriptor that should be used when sending and receiving data on that connection. The listening socket, by the way, is *still* listening for incoming connections on the same port, using the old descriptor.

As should be pretty clear now, operating the Berkeley Sockets API in x86 assembly language is largely a matter of juggling data, and lots of it. Let's tie it all together now - here is a very simple 'Hello World' server, which, unlike the previous listings, is complete code, and CAN be assembled and executed in it's entirety. This listing not only ties everything in the previous listings together, but also introduces the 'send' kernel function in addition to a couple of bread and butter kernel services, which should be pretty easy to pick out in the code and understand:

```

;=====  

;
; simple 'hello world' server
;
; NOT A CODE FRAGMENT: This code can *still* potentially BORK your system, but is highly unlikely to do so without external influences.
; Assemble and execute at your own risk.
;
; Written for NASM 0.99.06-20071101. Assembling, linking and execution instructions follow, and assume that NASM and ld are installed on
; your system:
;
;     user@localhost:/localdir$ nasm -f elf -o filename.o filename.asm
;     user@localhost:/localdir$ ld -s -o filename filename.o
;     user@localhost:/localdir$ ./filename
;
; Execution can be halted via the use of the <CTRL+C> command from the executing terminal.

```

```

section .bss

%define sys.socket.call           0066h ; kernel service number 066h corresponds to the Berkeley Sockets API
%define sys.system.call          0080h ; int 080h is how we access kernel services in Linux

%define sys.poll.call            00a8h ; kernel service number 0a8h corresponds to descriptor polling
%define sys.poll.in              0001h ; indicator for data on the socket
%define sys.poll.timeout         0010h ; timeout for polling function
%define sys.poll.number.of.structures 0001h ; number of event structures to poll for

%define sys.write.call           0004h ; kernel service number 004h writes data to a descriptor (not a socket)
%define sys.write.standard.output 0001h ; descriptor for standard output
%define sys.close.call           0006h ; kernel service 006h closes out an active descriptor
%define sys.exit.call            0001h ; kernel service 001h terminates the active process and passes control back to
; the terminal

%define sys.socket.create        0001h ; Socket service 001h corresponds to socket creation
%define sys.socket.bind          0002h ; Socket service 002h corresponds to port binding
%define sys.socket.listen        0004h ; Socket service 004h corresponds to telling a socket to listen for incoming
; connections
%define sys.socket.accept        0005h ; Socket service 005h corresponds to accepting incoming connections on a
; listening socket
%define sys.socket.send          0009h ; Socket service 009h corresponds to sending data on a connected socket

%define sys.socket.protocol.family.inet 0002h ; you may know this parameter as PF_INET
%define sys.socket.type.stream  0001h ; and this one as SOCK_STREAM
%define sys.socket.address       0000h ; and this one as INADDR_ANY
%define sys.socket.port.number   280ah ; port number to be bound to socket - stored in little endian byte order
%define sys.socket.address       0000h ; you may know this parameter as INADDR_ANY
%define sys.socket.padding       0000h ; padding for sockaddr structure
%define sys.socket.queue.length  0005h ; for the sake of this article we'll let 005h connections pile up

socket.data.connected.sock.descriptor resd 0001h ; storage for descriptor associated with accepted socket
socket.data.protocol.family         resd 0001h ; storage for protocol family, in this instance PF_INET
socket.data.flags                    resd 0001h ; storage for communication semantics, in this instance SOCK_STREAM
socket.data.address                  resd 0001h ; storage for address, in this instance INADDR_ANY

socket.data.listening.sock.descriptor resd 0001h ; storage for descriptor associated with bound, active socket
socket.data.queue.length             resd 0001h ; storage for connection queue length

```

section .data

```

socket.data.bind.sock.descriptor dd 0000h ; storage for descriptor associated with socket to be bound
socket.data.bind.pointer.sockaddr dd socket.data.sockaddr.protocol.family ; storage for pointer to sockaddr data
socket.data.bind.length.sockaddr dd socket.data.sockaddr.length ; storage for length of sockaddr data

socket.data.sockaddr.protocol.family dw sys.socket.protocol.family.inet ; sockaddr element for storing the protocol family, in
; this instance PF_INET
socket.data.sockaddr.port.number dw sys.socket.port.number ; sockaddr element for storing the port number in little endian
; order
socket.data.sockaddr.address dd sys.socket.address ; sockaddr element for storing the address, in this instance
; INADDR_ANY
socket.data.sockaddr.padding dq sys.socket.padding ; padded element
socket.data.sockaddr.length equ $-socket.data.sockaddr.protocol.family ; length of sockaddr structure

socket.data.event.sock.descriptor dd 0000h ; storage for descriptor associated with bound, active, and listening socket
socket.data.event.requested dd sys.poll.in ; storage for requested event(s)
socket.data.event.returned dd 0000h ; storage for returned event(s)

socket.data.accept.sock.descriptor dd 0000h ; storage for descriptor associated with bound, active, and listening socket
socket.data.accept.sockaddr.pointer dd socket.data.sockaddr.protocol.family ; storage for pointer to sockaddr structure
socket.data.accept.buffer.pointer dd socket.data.connected.sock.descriptor ; storage for pointer to storage for descriptor
; returned by accept function

socket.data.send.sock.descriptor dd 0000h ; storage for descriptor associated with socket upon which to send data
socket.data.send.buffer.pointer dd socket.data.hello.world.message ; storage for pointer to buffer containing data to be
; sent
socket.data.send.buffer.length dd socket.data.hello.world.message.length ; storage for length of buffer containing data to
; be sent
socket.data.send.flags dd 0000h ; storage for communication semantics

socket.data.listening.message db 'There were no problems. ' ; message indicating success
db 'The socket is now listening, and can '
db 'be connected to on port 2600.', 10
socket.data.listening.message.length equ $-socket.data.listening.message ; length of message indicating success

socket.data.connection.notifier db 'Incoming connection established.', 10 ; message indicating a connection
socket.data.connection.notifier.length equ $-socket.data.connection.notifier ; length of message indicating a connection

socket.data.hello.world.message db 10, 'Hello, World!', 10, 10 ; message to be sent to incoming connections
socket.data.hello.world.message.length equ $-socket.data.hello.world.message ; length of message to be sent to incoming
; connections

socket.data.create.error.message db 'There was a problem creating the ' ; message to be sent to user on error creating
db 'socket.', 10 ; the socket
socket.data.create.error.message.length equ $-socket.data.create.error.message ; length of message to be sent to user on error
; creating the socket

socket.data.bind.error.message db 'There was a problem binding the ' ; message to be sent to user on error binding the
db 'socket to the specified port.', 10 ; socket
socket.data.bind.error.message.length equ $-socket.data.bind.error.message ; length of message to be sent to user on error
; binding the socket

socket.data.listen.error.message db 'There was a problem getting the ' ; message to be sent to user on error getting the
db 'socket to listen on the specified ' ; socket to listen
db 'port.', 10
socket.data.listen.error.message.length equ $-socket.data.listen.error.message ; length of message to be sent to user on error
; getting the socket to listen

socket.data.accept.error.message db 'There was a problem accepting an ' ; message to be sent to user on error accepting
db 'incoming connection.', 10 ; an incoming connection

socket.data.accept.error.message.length equ $-socket.data.accept.error.message ; length of message to be sent to user on error
; accepting an incoming connection

```

section .text

global _start

```

_start: mov     eax, sys.socket.protocol.family.inet ; eax - protocol family
mov     [socket.data.protocol.family], eax ; store protocol family in arguments package for the socket creation function
mov     eax, sys.socket.type.stream ; eax - flags - socket type
mov     [socket.data.flags], eax ; store flags in arguments package for the socket creation function
mov     eax, sys.socket.address ; eax - address for socket
mov     [socket.data.address], eax ; store address in arguments package for the socket creation function

mov     eax, sys.socket.call ; eax - kernel service 066h: Berkeley Sockets API
mov     ebx, sys.socket.create ; ebx - socket service 001h: create socket
mov     ecx, socket.data.protocol.family ; ecx - pointer to arguments package for socket service 001h
int     sys.system.call ; create that socket!

or     eax, eax ; does eax contain an error code?
jns    socket.bind ; no - continue on to socket binding

jmp    socket.create.error ; yes - exit with an error message

```

socket.bind:

```

mov     [socket.data.bind.sock.descriptor], eax ; store descriptor associated with socket in arguments package for the bind
; function
mov     [socket.data.listening.sock.descriptor], eax ; store descriptor associated with socket in arguments package for
; the listen function
mov     [socket.data.event.sock.descriptor], eax ; store descriptor associated with socket in arguments package for
; the poll function
mov     [socket.data.accept.sock.descriptor], eax ; store descriptor associated with socket in arguments package for
; the accept function

mov     eax, sys.socket.call ; eax - kernel service 066h: Berkeley Sockets API
mov     ebx, sys.socket.bind ; ebx - socket service 002h: bind socket to port
mov     ecx, socket.data.bind.sock.descriptor ; pointer to arguments package for socket service 002h
int     sys.system.call ; bind that socket!

or     eax, eax ; does eax contain an error code?
jns    socket.listen ; no - continue on to telling the socket to listen

jmp     socket.bind.error ; yes - exit with an error message

socket.listen:

mov     eax, sys.socket.queue.length ; eax - queue length for socket
mov     [socket.data.queue.length], eax ; store queue length in arguments package for the listen function

mov     eax, sys.socket.call ; eax - kernel service 066h: Berkeley Sockets API
mov     ebx, sys.socket.listen ; ebx - socket service 004h: tell socket to listen
mov     ecx, socket.data.listening.sock.descriptor ; pointer to arguments package for socket service 004h
int     sys.system.call ; tell that socket to listen!

or     eax, eax ; does eax contain an error message?
jns    socket.display.success ; no - continue on to polling the socket

jmp     socket.listen.error ; yes - exit with an error message

socket.display.success:

mov     ecx, socket.data.listening.message ; ecx - pointer to success message to be displayed to user
mov     edx, socket.data.listening.message.length ; edx - length of success message to be displayed to user
call    socket.write.user ; display the success message to the user

socket.poll:

mov     eax, sys.poll.call ; eax - kernel service 0a8h: poll descriptor
mov     ebx, socket.data.event.sock.descriptor ; ebx - pointer to arguments package for kernel service 0a8h
mov     ecx, sys.poll.number.of.structures ; ecx - number of event structures in arguments package
mov     edx, sys.poll.timeout ; edx - timeout for poll function
int     sys.system.call ; poll that socket!

cmp     eax, sys.poll.in ; is there a connection waiting?
jnz    socket.poll ; nope - go back and poll again

mov     eax, sys.socket.call ; eax - kernel service 066h: Berkeley Sockets API
mov     ebx, sys.socket.accept ; ebx - socket service 005h: accept incoming connection
mov     ecx, socket.data.accept.sock.descriptor ; ecx - pointer to arguments package for socket service 005h
int     sys.system.call ; accept that connection!

or     eax, eax ; does eax contain an error message?
jns    socket.send.message ; no - continue on to sending the 'hello world' message

jmp     socket.accept.error ; yes - exit with an error message

socket.send.message:

mov     ecx, socket.data.connection.notifier ; ecx - pointer to message indicating an established connection
mov     edx, socket.data.connection.notifier.length ; edx - length of message indicating an established connection
call    socket.write.user ; display the message to the user

mov     [socket.data.send.sock.descriptor], eax ; store the descriptor returned when we accepted the connection

mov     eax, sys.socket.call ; eax - kernel service 066h: Berkeley Sockets API
mov     ebx, sys.socket.send ; ebx - socket service 009h: send data on a connected socket
mov     ecx, socket.data.send.sock.descriptor ; ecx - pointer to arguments package for kernel service 009h
int     sys.system.call ; send that data!

mov     eax, sys.close.call ; eax - kernel service 006h: close out an active descriptor
mov     ebx, [socket.data.send.sock.descriptor] ; ebx - descriptor for connected socket
int     sys.system.call ; close that socket!

jmp     socket.poll ; done sending message, go back to watching the listener for more connections

socket.create.error:

mov     ecx, socket.data.create.error.message ; ecx - pointer to error message to be displayed to user
mov     edx, socket.data.create.error.message.length ; edx - length of error message to be displayed to user
call    socket.write.user ; display the error message to the user
jmp     socket.exit ; terminate the process and return control to the terminal

```

socket.bind.error:

```
mov    ecx, socket.data.bind.error.message    ; ecx - pointer to error message to be displayed to user
mov    edx, socket.data.bind.error.message.length ; edx - length of error message to be displayed to user
call   socket.write.user                    ; display the error message to the user

mov    eax, sys.close.call                  ; eax - kernel service 006h: close out an active descriptor
mov    ebx, [socket.data.bind.sock.descriptor] ; ebx - descriptor for socket producing errors on bind
int    sys.system.call                      ; close that socket!

jmp    socket.exit                          ; terminate the process and return control to the terminal
```

socket.listen.error:

```
mov    ecx, socket.data.listen.error.message ; ecx - pointer to error message to be displayed to user
mov    edx, socket.data.listen.error.message.length ; edx - length of error message to be displayed to user
call   socket.write.user                    ; display the error message to the user

mov    eax, sys.close.call                  ; eax - kernel service 006h: close out an active descriptor
mov    ebx, [socket.data.listening.sock.descriptor] ; ebx - descriptor for socket producing errors when told to listen
int    sys.system.call                      ; close that socket!

jmp    socket.exit                          ; terminate the process and return control to the terminal
```

socket.accept.error:

```
mov    ecx, socket.data.accept.error.message ; ecx - pointer to error message to be displayed to user
mov    edx, socket.data.accept.error.message.length ; edx - length of error message to be displayed to user
call   socket.write.user                    ; display the error message to the user

jmp    socket.poll                          ; get back to work looking for incoming connections
```

socket.write.user:

```
push   eax                                  ; preserve caller's eax
push   ebx                                  ; preserve caller's ebx

mov    eax, sys.write.call                  ; eax - kernel service 004h: write data to a descriptor (not a socket)
mov    ebx, sys.write.standard.output      ; ebx - descriptor corresponding to STDOUT
int    sys.system.call                      ; display that message!

pop    ebx                                  ; restore caller's ebx
pop    eax                                  ; restore caller's eax

ret                                         ; return to caller
```

socket.exit:

```
mov    eax, sys.exit.call                  ; eax - kernel service 001h: terminate current process and pass control back to
                                           ; the terminal
int    sys.system.call                      ; terminate.
```

=====

Now the fun and excitement truly begins. After you've assembled and linked the code, execute the resulting program (at your own risk) - if nothing gets borked, you'll get a message indicating that the socket is listening and can be connected to via port 2600. It CAN! Try it with telnet or your favorite mud client, or any other way you want to do it. You'll connect up, the server will send you "Hello World!" and then disconnect the socket. A notification will also appear on the terminal running the server, then the server will go back to listening for incoming connections.

There are certainly more elegant ways to do this, shaving cycles, cutting down on binary size... I know. The purpose of this document is to demonstrate the basics of Berkeley Sockets in x86 assembly language.

That should be about enough to get the gears turning. Please feel free to modify the code, turn it into whatever you want, have fun, and play rough.