

COMP8051 – Operating System Engineering - Assignment3/Project

Completion Date: 5th May 2023

Value: 20 marks

On completion please zip up your files and upload to Canvas.

Question 1: concurrent web server in Linux

Given the code in server.c (on Canvas as server.txt) write a simple concurrent web server. The server should be able to process HTTP GET requests.

Compile and build server.c with:

```
gcc -o server server.c -lpthread
```

You can connect to the server using telnet or a web browser

e.g. run the following command from the command prompt.

```
telnet localhost 3000
```

or

enter localhost:3000 in address bar of the browser

Hints

1. To get the length of a file do:

```
#include <stdio.h>
filesize(){
    FILE *fp = fopen("test.txt","r");
    fseek(fp, 0L, SEEK_END); // go to the end of the file
    __off_t file_size = ftell(fp); // ftell returns the current position in the file relative to the start
                                // of the file
    fseek(fp, 0L, SEEK_SET); // go back to the start of the file
}
```

2. Get the current working directory with:

```
char cur_dir[200];
getcwd(cur_dir, 200);
```

3. Minimal HTTP response

You need a carriage return followed by line feed on each line of the response in c this is `\r\n`.

The last line is a blank line with `\r\n`. See the code on line 57 of server.c

```
printf(buf, "%sContent-Length: %lu\r\n\r\n", buf, content_length);
```

see also here

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Response_message

Example response

```
HTTP/1.1 200 OK
Content-Length: 13
Content-Type: text/plain; charset=utf-8
```

Hello World!

(5 marks)

Question 2: Asynchronous I/O

Give a brief overview of libuv the underlying runtime for node.js and how it interacts with the Linux kernel using epoll. See here

<https://eli.thegreenplace.net/2017/concurrent-servers-part-3-event-driven/>

Node.js uses Libuv for implementing asynchronous I/O on Linux. The Linux kernel provides different types of interfaces for managing async I/O such as the select and epoll functions. Programming language runtimes for web servers and microservices on Linux are typically build on these interfaces. See this video for a discussion on epoll and the libuv/nodejs async I/O model

<https://www.youtube.com/watch?v=P9csgxBgaZ8>. See also here:

<https://eli.thegreenplace.net/2017/concurrent-servers-part-4-libuv/>

(5 marks)

Question 3 - Socket layer xv6

Describe how the user is able to send and receive packets to/from the E1000 device with simple system calls such as **read** and **write**.

See appendix 1 for more detail.

(5 marks)

Question 4 - eBPF and XDP

Give an overview of eBPF and XDP and explain how one would build a simple load balancer using eBPF and XDP?

Simple load balancer to review for this question:

https://youtu.be/L3_AOFSNKK8

<https://github.com/lizrice/lb-from-scratch.git>

Links and references eBPF and XDP

paper in ebpf_XDP.zip

https://youtu.be/Hed2DOrk_kk

<https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/>

<https://www.youtube.com/watch?v=YUI78vC4qSQ>

Høiland-Jørgensen, Toke, et al. "The express data path: Fast programmable packet processing in the operating system kernel." *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 2018.

Vieira, Marcos AM, et al. "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications." *ACM Computing Surveys (CSUR)* 53.1 (2020): 1-36.

(5 marks)

Appendix 1

Note: If you get qemu-system-i386 -redir: invalid option error

Try one of the commands below in the directory where you run make qemu (you will need everything on one line). See also these commands on the assignment in Canvas – they may not copy correctly from here.

----- Ubuntu 20.04 -----

```
qemu-system-i386 -serial mon:stdio -drive
file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -netdev
user,id=net0,hostfwd=tcp::20001-:7,hostfwd=tcp::20002-:80,hostfwd=udp::200
00-:7 -object filter-dump,id=net0,netdev=net0,file=qemu.pcap -net
nic,id=net0,netdev=net0,model=e1000
```

----- Ubuntu 22.04 ---

```
qemu-system-i386 -serial mon:stdio -drive
file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -netdev
user,id=net0,hostfwd=tcp::20001-:7,hostfwd=tcp::20002-:80,hostfwd=udp::200
00-:7 -object filter-dump,id=net0,netdev=net0,file=qemu.pcap -net
nic,id=net2,netdev=net0,model=e1000
```

Background - question 4

Download the network-sockets-xv6-e1000-lab.zip from Canvas.

The code is taken from here <https://pdos.csail.mit.edu/6.828/2019/labs/net.html>

Note the code at this link <https://pdos.csail.mit.edu/6.828/2019/labs/net.html> is for the 2019 version of xv6 i.e. riscv. See canvas for a port of the code to xv6 for the x86 i.e. network-sockets-xv6-e1000-lab.zip.

You may find it helpful to review "Traps and device drivers", " File descriptor layer" from the xv6 book, and [the lecture notes on networking](#).

We are using a virtual network device called the E1000 to handle network communication. To xv6, the E1000 looks like a real piece of hardware connected to a real Ethernet local area network (LAN). But in reality, the E1000 that the driver talks to is an emulation provided by qemu, connected to a LAN that is also emulated by qemu. On this LAN, xv6 (the "guest") has an IP address of 10.0.2.15. The only other (emulated) computer on the LAN has IP address 10.0.2.2. qemu arranges that when xv6 uses the E1000 to send a packet to 10.0.2.2, it's really delivered to the appropriate application on the (real) computer on which you're running qemu (the "host").

We will be using QEMU's user mode network stack since it requires no administrative privileges to run. QEMU's documentation has more about user-net [here](#). We've updated the Makefile to enable QEMU's user-mode network stack and the virtual E1000 network card.

QEMU's network stack will record all incoming and outgoing packets to `packets.pcap`. To get a hex/ASCII dump of captured packets use `tcpdump` like this:

```
tcpdump -XXnr packets.pcap
```

or use `wireshark`

```
wireshark packets.pcap
```

<http://localhost:20001/>

Overview of the socket layer in xv6

Network sockets are a standard abstraction for OS networking that bear similarity to files. Sockets are accessed through ordinary file descriptors (just like files, pipes, and devices). Reading from a socket file descriptor receives a packet while writing to it sends a packet. If no packets are currently available to be received, the reader must block and wait for the next packet to arrive (i.e. allow rescheduling to another process). The code in `xv6-e1000-sockets.zip` is a stripped down version of sockets that supports the UDP network protocol.

Each network socket only receives packets for a particular combination of local and remote IP addresses and port numbers, and `xv6` is required to support multiple sockets. A socket can be created and bound to the requested addresses and ports via the `connect` system call, which returns a file descriptor. The implementation of this system call is in `kernel/sysfile.c`. The code for `sockalloc()` and related functions is in `kernel/sysnet.c`.

Take note of the provided data structures; one `struct sock` object is created for each socket. `sockets` is a singly linked list of all active sockets. It is useful for finding which socket to deliver newly received packets to. In addition, each socket object maintains a queue of `mbufs` waiting to be received. Received packets will stay in these queues until the `read()` system call dequeues them.

Running the test program.

```
qemu-system-i386 -serial mon:stdio -drive  
file=fs.img,index=1,media=disk,format=raw -drive  
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -netdev  
user,id=net0,hostfwd=tcp::20001-:7,hostfwd=tcp::20002-:80,hostfwd=udp::200
```

```
00-:7 -object filter-dump,id=net0,netdev=net0,file=qemu.pcap -net  
nic,id=net2,netdev=net0,model=e1000
```

(in one terminal on your laptop)

```
$ python2 server.py 26099
```

listening on localhost port 26099

(then on xv6 in another terminal on the same machine run

make qemu

and run nettests at the xv6 shell prompt

```
$ nettests
```