Project Payload

CSE 481A Final Report

One small step for xv6, one giant leap for our brains

Colin Evans
colin21@cs.washington.edu

Megan McGrath
mkmg@cs.washington.edu

Sixto Josue Rios
jrios777@cs.washington.edu

Paul Yau
yaup@cs.washington.edu

Zhitao Zhang
zzt124@cs.washington.edu

1 Abstract

For our CSE 481A Cloud Capstone project, we implemented network support and container support for xv6 [4], a simple, Unix-like educational operating system. Added network support allows us to communicate with xv6 from outside of QEMU [12] and provides incentive for implementing containers. Container support provides isolation of resources. Our containers isolate processes, files, and network access from each other. We accomplish this by implementing namespaces, and isolating these resources based on namespace. Unlike Linux, where different namespaces can be mixed and matched to create isolation or allow sharing, our containers only completely isolate the namespace.

2 Introduction

For our capstone project, we wanted to choose a topic related to the cloud platforms common in today's world. With containers and virtual machines being popular services today, we chose to implement containers inside xv6 [4], an educational operating system based on Unix version 6. Since most container instances in the real world involve talking over the network, we also decided to add network support and isolation. From our past experience, we were familiar with Docker being used to host web servers in the cloud, and we wanted to create a similar capabilities in xv6.

The rest of this paper is organized as follows. **Section 3** describes related work. **Section 4** details some of the challenges of our project. **Section 5** gives an overview on xv6 and a description of some additional functionality we added to support the development of our planned features. **Section 6** describes our design, implementation, and results in adding network support. **Section 7** describes our design, implementation, and results for container support. **Section 8** concludes.

3 Related Work

In terms of networking support, there is plenty of related work that adds networking support to kernels. JOS [11] and hv6 [7] use IPC and a network serving process with multiple threads to integrate network support with lightweight IP (lwIP). xv6-networking-stack [6] integrates the E1000 driver and sends an ARP request but does not integrate lwIP. xv6plus [5], loosely based on xv6, integrates a different driver but similarly integrates lwIP though through the use of kernel threads. Our network support integration has some similarities but on top of that, we have network isolation for containers.

As for work related to our containers implementation, we based our design loosely on Linux namespaces. Linux provides a clone system call which creates a new namespace with the given resources isolated. There are many resources to choose from:

- **UTS** - Hostname namespace
- **mnt** - Mount points/file system namespace
- **PID** - Process ID namespace
- **User** - User namespace
- **Network** - Network namespace
- **IPC** - Interprocess Communication namespace

The namespaces are created by passing a flag OR-ed with each desired namespace's flag. We decided to implement a subset of these namespaces with an all or nothing interface since this better fit with our goal of supporting containers.

4 Challenges

Part of the challenge of implementing these features in xv6 is that xv6 itself is missing a lot of the tools and features that we are used to having in our standard Linux environment. For example, there are no users, no permissions, no procfs (a special process file system in UNIX-like systems), no sophisticated Interprocess Communication (IPC), no threads, no network support, among many other things. Thus, we either had to implement them or create workaround solutions.

For networking support, we had trouble integrating lightweight IP (lwIP) since xv6 was missing a lot of components. For one, we didn't have threads so we could not use the sequential API in the kernel nor did we have an easy way of sharing pages via IPC so we could not use the sequential API in a network serving process much like what's found in JOS. The solutions we found online for xv6 involved one or the other and usually involved more changes to how xv6 was structured than we desired.

Furthermore, having only previously used high level container programs such as Docker, we began this project lacking a understanding of how containers like Docker containers are implemented. It took us a while to comprehend that we would not be building a new version of Docker, but rather a version of Linux namespaces, a set of kernel features upon which container programs like Docker are built.

5 Xv6

xv6 [4] is a simple, Unix-like teaching operating system developed at MIT during the summer of 2006. Its a reimplementation of Unix Version 6 which was developed in the 1970s. We decided to build onto xv6 because we wanted to start with a fairly primitive operating system that was Unix-like. Specifically we used the xv6 source code provided as part of the lvisor repository [13] created for the CSE 481A capstone offered at the University of Washington. Since xv6 is a bare bones OS, we started our project by adding a few user programs and their corresponding system calls.

ps Since we planned to add process isolation as part of container support, we started by implementing the user level program `ps`. This program prints out all the processes along with their process IDs, the ID of their parent, and the name of the program they're running. To implement this user program, no additional system calls were added. Instead we decided to model our implementation off of Linux's `/proc` directory. Basically Linux makes information about processes available through the file system. If you run `ls /proc` on a Linux machine, you'll see a directory for each process, named with the process' ID. Inside each process folder are more folders containing information about each process. We made our implementation a bit simpler. Rather than having a separate directory for each process, we simply populated the `/proc` directory with a file for each process (this design minimizes traps into the kernel). The process file contains the data of a custom struct we designed, called `struct procf`. `struct procf` contains information such as the process ID, the ID of the parent process, and the name of the program being run. This content is generated dynamically using the information stored in the global process table. Just like a normal file, a file in `/proc` is first opened, creating a file descriptor and an entry in the global file table used to store data such as offset. This offset is used to determine how much of the `/proc` directory or process file has been read. We added an extra field to the struct stored in the global file table to store a pointer to a process struct to keep track of our place when iterating through the global process table. With this `/proc` interface implemented, the user level program `ps` simply has to open files and read them using normal file system calls. Much of the logic in ps is similar to the logic in the program `ls`.

pwd For the purpose of testing file system isolation, we built a `pwd` program to display the full file path of the current process. It consists of a user program and a system call into the kernel. When the syscall is invoked, it will recursively traverse up the hierarchy of inodes from process's current working directory to the process' root inode. Since the inode struct doesn't store any names, it has to get the inode of the parent and retrieve the name from the parent inode struct. The user program makes a call to the corresponding syscall. There are other designs to implement a `pwd` program, we chose this particular design to reduce the number of times we trapped into the kernel.

6 Networking

Our networking design and implementation is loosely based on the E1000 driver we wrote for JOS and multiple online networking solutions we found online that integrated lwIP or the E1000 driver, such as xv6plus [5], xv6-networking-stack [6] and hv6 [7]. Unfortunately, none of those solutions were easily portable since they all depended on either threads or more sophisticated IPC than what pipes can easily provide. Additionally, except for xv6-networking-stack [6], we were unable to successfully compile and run any of these solutions on the school lab machines. We eventually gave up on porting anything over and instead decided to make sense of all these solutions and the lwIP source code in order to try to understand how we might accomplish a simpler solution. Unfortunately our solution was all but simple, but designing and implementing pieces one at a time helped facilitate the process.

6.1 E1000 Driver

The E1000 driver is responsible for setting up the network interface card properly, which involves configuring the driver and initializing queues for sending and receiving of data. The design and implementation of this is as straight forward as following Intel's E1000 driver manual [9]. We first implemented it in JOS' lab 6 to confirm our understanding. When we tried to port it over to xv6 [4] though we learned that xv6 was missing Memory-mapped I/O (MMIO) and PCI support. This is where deciphering online solutions helped out, especially when looking at xv6-networking-stack [6] and xv6plus [5].

Memory-mapped I/O (MMIO) Our solution in JOS relied on memory mapped I/O but we had no idea how to do this in xv6 initially. After reading through the driver code in xv6plus and xv6-networking-stack we came up with a working solution for memory mapping I/O.

PCI Support Since xv6-networking-stack contained minimal changes to xv6 from our xv6's existing files, we were able to add PCI support using the same techniques they had after specifying our attach function, which gets run during PCI initialization in the boot process.

Finally, to ensure our driver worked before integrating lwIP, we found an `arptest` program in xv6-networking-stack that didn't rely on lwIP. When we replaced xv6-networking-stack's [6] driver with ours (theirs doesn't receive packets from the driver), after some small modifications we were able to send and receive ARP requests and replies. This user program was also added to our xv6 implementation for demo purposes and regression testing.

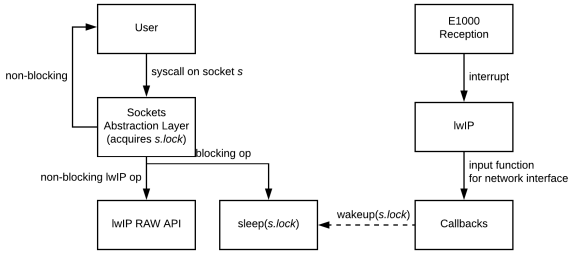

Figure 1: User program to network card flow. This is after creating a socket and assumes the callback has context on the socket.

6.2 lwIP Integration

Once we had a working driver we needed to integrate lwIP. LwIP has three APIs as detailed in the lwIP Wiki [8]:

- low-level "core" / "callback" or "raw" API.
- higher-level "sequential" netconn API.
- BSD-style socket API that wraps around the netconn API.

The sequential APIs rely on threads, so since we didn't have them nor IPC, we chose to use the low-level "raw" API. This API relies on a per-socket Protocol Control Block (PCB). Since we can keep track of this in the kernel per file descriptor that corresponds to a socket, our design is basically to call the relevant `tcp_*` or `udp_*` functions, track that in a struct with a spin lock and use that struct as a callback arg. Then, on blocking calls the kernel puts the process to sleep on the socket's lock, to be later awoken by a callback when there's activity on the PCB. Callbacks are usually triggered by an E1000 interrupt. The flow for this design is shown in Figure 1.

To integrate this API we had to implement a couple things. First was an `arch/cc.h` header file which defines certain types lwIP needs and an `lwipopts.h` header file for lwIP options. Additionally since lwIP needs a network interface, for communicating with the driver, we added that as well in a file called `netif.c`. `netif.c` defines functions for configuring the interface, reading from driver when there's input, and writing to the

driver when there's output. Once we had these files defined, to setup the network interface we added some initialization code that provides lwIP with our network interface and brings the link up. Then, for interrupts, to allow the E1000 driver to notify lwIP that a packet has been received, we borrowed the strategies in xv6-networking-stack [6] to set them up. On a hardware interrupt, if we see it's from the network card xv6 notifies lwIP that there is input and lwIP handles calling the appropriate callback.

6.3 Socket Abstraction

The lwIP "raw" API provides you with functions that are somewhat similar to BSD sockets. However, instead of blocking you pass a callback function that gets notified when there is a result for the function. We decided to build out an abstraction that provides an API as close as possible to the BSD sockets API for user programs to use and also to easily port over external user programs that rely on them. Disregarding any network isolation specific state for containers, from a high level our socket abstraction keeps track of:

- **socket type** - UDP or TCP.
- **pcb** - lwIP specific UDP/TCP PCBs.
- **lock** - spin lock for atomic operations and condition variables.
- **accept state** - incoming connections received asynchronously.
- **receive state** - packets received asynchronously and a read offset into them.
- **closed state** - whether the socket is closed, as it could've happened asynchronously.

So our design basically relies on asynchronous events, triggered by callbacks. When a socket is created its callback argument is set to this socket abstraction such that all callbacks have context on the related socket. To make the calls blocking, a system call sleeps on the socket (as if it were a condition variable) if it needs to wait for something. When executed successfully, callbacks wake up processes sleeping on the socket. To ensure socket operations are non-interfering, a socket's spin lock is acquired when operating on it from a system call or from a callback, which disables interrupts until the operation completes.

Integrating With Xv6 File Descriptors Our sockets are built with a file abstraction in mind. We added to the existing `struct file` in xv6 an `FD_SOCK` type and a field with the socket abstraction above. User programs work with file descriptors just like in Linux.

Supporting Loopback Connections Our implementation also allows for loopback connections on 127.0.0.1 (localhost). LwIP supports this, though to actually get it to work we added a periodic call to `netif_poll` in the scheduler so that loopback operations flowed through.

Sockets API Our sockets implementation supports the following "Big Socket Daddy" (BSD) Sockets API calls:

- `accept()`
- `bind()`
- `connect()`
- `getpeername()`
- `getsockname()`
- `listen()`
- `recv()`
- `recvfrom()`
- `send()`
- `sendto()`
- `socket()`

User Programs and Results Along with the API we included a lot of network-specific types, struct definitions and functions such as `struct sockaddr` and `inet_aton` for maximum compatibility with user programs that depended on them. These definitions and functions were independent of the kernel so they're easily able to be used by both user and kernel code.

To test our implementation we added a couple of user programs borrowed from JOS and online sources:

- `arptest` - Sends an ARP request.
- `curl` - Curls a web page via HTTP and displays the content to the console. We modified the code we borrowed to not do DNS and instead take in an IP address.
- `httpd` - Runs an HTTP server on port 80. Borrowed from JOS, we made some modifications to have it generate HTML for a directory and know more mime types.
- `tcpecho` - Runs a TCP server on port 7 that just sends to client what it received.
- `udpecho` - Same as `tcpecho` but with UDP.

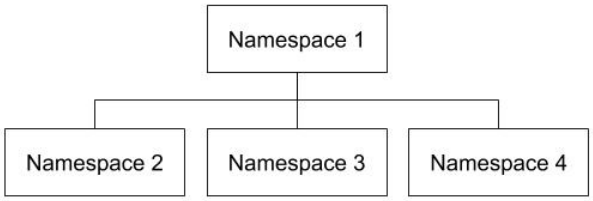

Figure 2: The hierarchal structure of namespaces. The root namespace, namespace 1, is the parent of all other namespaces. Namespaces 2 and greater are all siblings, and are unable to access the resources of their siblings.

All of the programs above worked as expected from all use cases we could think of. We were even able to download web content (`curl sompage > somefile`)and then serve it up all from within xv6 using `httpd`. A minor note on this functionality though, we're not sure what it is, but either lwIP or QEMU or our code is slow so fetching web content from xv6 is not as fast as we're normally used to.

7 Containers

The second part of our project involved adding container support to xv6. A container inside xv6 has a new view of the system resources separate from that of the root system. It sees a different view of the active processes, the file system, and the network. We added container support by implementing namespaces in xv6.

7.1 Namespace

Namespaces in xv6 achieve isolation by showing processes only the resources their specific namespace has access to and preventing access to resources in other namespaces. Unlike Linux namespaces, where each of the six different resources can be selectively isolated with their respective namespaces, our xv6 implementation only allows one type of namespace where processes, the file system, and networking are isolated.

To manage the data associated with namespaces, we had to make two main changes. First we added an array of namespace structs (the global namespace table) to keep track of the active namespaces and associated data, and then we added a field to the process struct to keep track of which namespace each process belongs to. The namespace ID stored in the process' namespace field determines which view of the system the process sees. The first namespace, which we call the *root namespace*, always has an ID of 1. Every other namespace is considered a child of the root namespace but equal in hierarchy to each other (called siblings), as seen in Figure 2.

Our implementation of namespaces are organized in a similar fashion as process IDs. The global namespace table tracks the number of processes currently in each namespace. When a new namespace is created, the kernel loops through the global table in search of a namespace to which no process belongs, and assigns that ID to the new namespace. Like process IDs, namespace IDs start at 1. When other processes are then created in that namespace, the count in the table for that namespace increments. Similarly, when a process belonging to a specific namespace is cleaned up, the count decrements. When the count returns to 0, resources allocated for that namespace are cleaned up.

Namespaces are created with a new system call we added called `spawn()`. The `spawn()` system call is modeled after the `fork()` system call semantics, and essentially does a `fork()` with the child existing inside a new namespace. Like `fork()`, it also copies over the program state and continues executing from the same place, differentiating the parent from the child using the return value of the system call. Programs can then follow the `fork-and-exec` model to launch different applications. We also modified `fork()` to let the child inherit the namespace from the parent.

Lastly, we created a new user program `spawn` that will create a new namespace and launch the `nsinit` program, which loops indefinitely, cleaning up children processes that have finished. `spawn` can take additional arguments to launch an additional program in this new namespace. Since the `nsinit` program will run indefinitely, the namespace will not be cleaned up even after the additional program launched has finished executing. However, `nsinit` running in a child namespace can be killed from root namespace, allowing the child namespace to be cleaned up.

7.2 PID Isolation

Each process inside a child namespace has its own unique process ID with respect to its namespace. That is, each child namespace resets the view of the process IDs. The initial process of a namespace always has a process ID of 1 when viewed from within its namespace (e.g. `ps` is run in a shell process belonging to the namespace). However, since we followed Linux conventions and allow a parent namespace (in this case the root namespace) to view all processes of its child namespace, each process of a child namespace must have a different process ID when viewed from the root namespace. Otherwise, we would

see many processes with a process ID of 1 when running `ps` from the root namespace. Since all namespaces other than the root namespace are children of the root namespace and equal to each other, we have a hierarchy limited to two levels, therefore requiring at most two process IDs per process (see Figure 2). To accommodate an additional process ID for each process, we added a field to the process struct called `nid` to store the namespace specific process ID. This is in addition the the `pid` field which stores the globally unique process ID. For processes inside the root namespace, the `pid` is equal to the `nid`. We have modified user interaction with the system so that all user actions use the process ID visible from the current view(e.g. `kill()` will take the process ID from the current namespace's view). This prevents namespaces from being able to kill processes created in sibling namespaces. We also add to modify `ps` to use `nid`s instead of `pid`s.

The program with `nid` 1 for every namespace is the `init` process. In the root namespace, this will be the actual `init` program launched when xv6 first boots up. For a child namespace, this will be the `nsinit` program launched by `spawn` when the namespace is first created. When a process inside a namespace has finished executing, it assigns all of its children to the `init` process of its own namespace, instead of the global `init` process. In addition to taking over responsibility of all orphaned processes, the `init` program for each namespace is also responsible for killing all remaining processes of the namespace should it exit while there are still other processes remaining in the namespace. This is done by killing all processes still alive inside the namespace, and assigning them to the global `init`, which will finish the clean up process with `wait()`.

7.3 File System Isolation

In order to support a container and secure the content in a namespace, we create a new file system when a namespace is spawned. Therefore, the content in a namespace cannot be modified by other namespaces within the same host.

To isolate the file system, a new root inode will be allocated when creating a namespace and assigned to the `proc_rootino` field of the init process. If a process is forked within a namespace, it will inherit the root inode from its parent process. Since file lookup in xv6 is recursive until it reaches the root, by setting a new inode as the `proc_rootino`, we have effectively isolated the files of a namespace from those of another namespace. Since the `proc_rootino` is defined when the namespace is initialized, and children inherit the `proc_rootino` from their parent, processes inside a namespace can never access files outside of its namespace. This change results in the / in a file path no longer pointing to `ROOTINO`, but the inode defined as the root for that namespace.

Creating a brand new root, however, removes the executables from the view of the new namespace, effectively turning it into a file system with no programs, which is pretty useless. We could add symbolic links to the executables to the new namespace. However, this would allow children namespaces to modify the executables by writing to it and corrupt the executables for the whole system, breaking isolation. To solve this, we copied the executables to the new namespace.

While copying executables achieve complete system isolation, we found the process to took a really long time in xv6. In order to improve performance, we decided to move all the executables files into a globally visible bin directory. Every exec call will start search in that bin directory. Otherwise, exec will search in current process directory. Other file system methods such as open, read and write do no access the bin. We modified the file system image to allocate another inode as a bin directory while the fs image is being initialized, and copied executables into that directory instead of root directory. Moreover, we also implement the functionality to copy over files from the root namespace to a child namespace. If a child namespace needs its own copy of a certain file, the parent can call "`nspcopy <filename> <child namespace id>`" to copy the specified file into the child's root directory. This creates a deep copy with no shared data in order to achieve complete isolation.

7.4 Network Isolation

One of the most interesting properties about containers in the real world is that you can run applications in different containers that bind to the same port number due to network namespace isolation. In reality those applications are bound to different port numbers on the host machines, often times specified by a configuration file. Another property of network isolation is that containers are often configured such that they have their own IP addresses within the host machine and can only talk to each other over the network using those IP addresses and the container's port number if the listening container is listening on it. For our implementation we kept things simple by defining container IP addresses and port forwarding based on namespace ID. Namely, for some namespace ID n the container IP address is 192.168.1.n. Underneath, since lwIP is unaware of the container addresses, xv6 actually translates container IP addresses and ports passed

in through system calls to ones lwIP can used based on the process' namespace and the task. We are able to get the IP address/port for a connection because lwIP provides Protocol Control Blocks for TCP and UDP that contain the local and remote IP address/port. For our purposes we'll refer to these as the host address and host port. In this way, the kernel acts much like a Network Address Translation box. In terms of network isolation, there are three interesting cases that are handled: `bind()`, `accept()/recvfrom()`, `connect()/sendto()`.

When binding, an application passes in a socket and the address and port it'd like to have the socket listen on. Our first step involves inspecting the address passed in, validating it and doing any necessary translation. If the address is not 0.0.0.0 nor 127.0.0.1 nor 192.168.1.*n* (where *n* is the namespace ID of the calling process) then the call fails. Otherwise, if the address is 192.168.1.*n*, this is a container address, so we change it to the host address before binding. If the address is 127.0.0.1 we take note that the socket is bound to a loopback address. Then, right before binding we modify the port number based on the namespace and keep track of the container port and host port for a socket.

Then, when a TCP connection or UDP packet comes in via `accept()/recvfrom()`, if it's coming from 0.0.0.0, we first check its port number to see if it's a valid host port (unfortunately, QEMU does not distinguish between connections from within xv6 and outside xv6 on the real host machine) and if so, what container it's coming from. If we know that the destination socket is a loopback socket then we drop the connection/packet if it's not coming from the same container. Otherwise, we let the packet through, and before returning the address and port to the user, if the connection came from a container, we set the address and port output parameters to the container address and port it came from.

For connecting or sending UDP packets, the process is very similar, but it's reversed. If the target address is a container address and port, we check to see if it's a valid container address and port, and if not reject the connection/sending. Because xv6 needs to know all container port to host port mappings for everything to work, one trick we did in `connect()/sendto()` is bind to port 0 and store that as the container port and host port before connecting/sending.

For purposes of testing, our `httpd` program binds on port 80 and can take in an address to bind to (0.0.0.0 is the default) and our `curl` program can take in an address, port (default 80), path (default `/`), and hostname (for real web servers, garbage header field in 'httpd') to send a GET request to. Since we provided a sockets API similar to Linux, they're based on programs from JOS and online solutions for curling web pages. We did modify `httpd` though to generate a web page for directories.

Since xv6 is aware of all host port to container port mappings, changes to the `ps` program (explained later) involved simply exposing those mappings to user space and outputting that. To facilitate data structures, we disallowed multiple simultaneous listening sockets per process since we don't have threads anyways. For controlling what gets returned to user space, we decided to expose the underlying host address/port if the `ps` program was run in namespace 1 and in any case always return the container port.

Results after modifying `httpd`, `curl`, `ps` show that containers are indeed isolated since one cannot be reached by another unless it's receiving packets from outside. Additionally, each container can independently bind to the same port number and use that communicate with others as if they were indeed machines with isolated network stacks. Finally, within namespace 1 all port mappings can be seen when doing `ps` but other ones think they're really listening on the container port they binded to.

7.5 Switching Between Namespaces

After implementing namespaces, we decided we needed a way to switch between namespaces so we could interact with multiple namespaces. Since xv6 only supports one terminal, we couldn't simply create each namespace in a new terminal. Instead, we implemented a user level program and corresponding system call called `attach`. The usage is as follows: say you're in namespace 1 and want to switch to an existing namespace 2. You would run "`attach 2`". This would cause the current shell to be placed into a waiting state and a new shell in namespace 2 would be created and immediately run. The reason the existing shell has to be placed in an unrunnable state is the new shell will use the same standard in and standard out as the existing shell. Having two shell programs reading from the same standard in stream leads to unpredictable behavior. When the user wants to return to namespace 1, they simply type "`attach 1`" and the shell that was created for the user to use in namespace 2 exits and the shell in namespace 1 is changed from waiting to runnable, and is immediately run by the scheduler. In order to guarantee the desired shell would be scheduled next, we modified the scheduler to use the process stored in a new variable, `nextProc`, if there is one, otherwise it resumes its normal scheduling algorithm. In order to ensure our containers are secure and don't leak information such as the existence of other containers, users cannot

switch between sibling namespace (e.g. from 2 to 3). Instead the user must first attach to the root namespace and then attach to the desired namespace.

8 Conclusion

In this project we added network support and container support to the xv6 operating system. Processes running inside a container have a separate view of the process IDs, file system, and network resources that is unique to itself. Working with xv6, which is pretty minimal, we had to implement many additional functionalities to support these two features, solving different challenges along the way. In the end, we were able to run multiple containers, each running a web server with its own view of processes, its own file system and the illusion of being bound to the same port. Throughout this project, we learned so much. At the beginning, we had used modern day containers such as Docker, but had no idea how they were implemented. Most of us weren't familiar with Linux namespaces before. Now we understand the Linux namespace API and behavior. We read up on how Linux implemented the various namespace types and then designed our own implementation that worked well with xv6. During the process we also learned more about operating systems; we're now mini xv6 experts. On top of all this we also got more familiar with the E1000 driver manual and lwIP source code than we ever wanted to. With this project completed, we can finally start paying attention to the movie plots again.

References

[1] Containers and Images
$https://docs.openshift.org/latest/architecture/core\_concepts/containers\_and\_images.html.$

[2] Linux Namespace Overview
$https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces$

[3] Linux Kernel
$https://github.com/torvalds/linux$

[4] xv6, a simple, Unix-like teaching operating system
$https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf$

[5] xv6plus
$https://github.com/HenryHu/xv6plus$

[6] xv6-networking-stack
$https://github.com/vibhorvatsa/xv6-networking-stack$

[7] hv6
$https://github.com/locore/hv6$

[8] light weight IP wiki
$http://lwip.wikia.com/wiki/LwIP\_Wiki$

[9] E1000 driver manual
$https://courses.cs.washington.edu/courses/cse451/16au/readings/e1000.pdf$

[10] light weight IP, lwIP
$www.nongnu.org/lwip/$

[11] CSE 451 JOS
$https://courses.cs.washington.edu/courses/cse451/16au/index.html$

[12] QEMU
$https://www.qemu.org/$

[13] lvisor repository
$https://gitlab.cs.washington.edu/cse481a/lvisor-18wi$