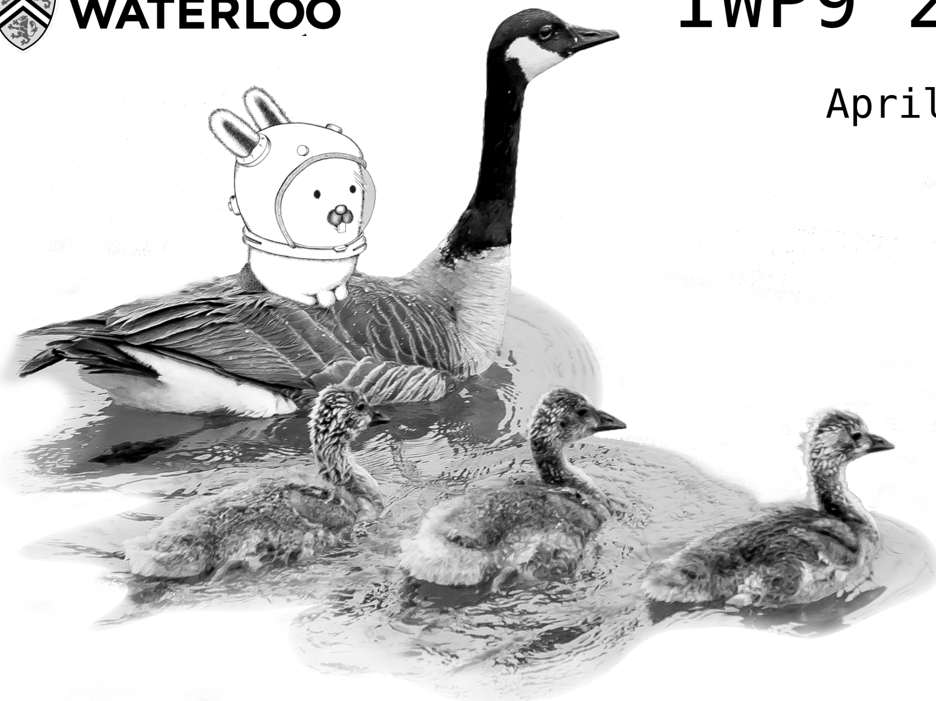
IWP9 2023

9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023

Proceedings

IWP9 2023

April 21-23



Sponsored by the Plan 9 Foundation

Volume Editor

Michael Engel
University of Bamberg
System Programming Group
An der Weberei 5, DE-96049 Bamberg, Germany

Copyright ©2023 for the individual papers by the papers' authors.
Copyright ©2023 for the volume as a collection by its editors.
This volume and its papers are published under the Creative Commons Attribution 4.0 International License (CC BY 4.0).

Preface

This book contains the proceedings for the Ninth International Workshop on Plan 9, IWP9. It was held from the 21st to the 23rd of April of 2023 at the University of Waterloo. We, the organizing committee, are happy to offer this workshop again after a long hiatus.

The workshop was organized by Ali José Mashtizadeh of the University of Waterloo, who deserves many thanks for providing support, lunch, coffee, meeting rooms, etc. We would also like to thank the Plan 9 Foundation and all members of the organizing and program committees for their work and support.

Proceedings updates and an online version of this book are available at

http://iwp9.org.

The workshop was generously sponsored by the Plan 9 Foundation.

Waterloo, April 2023

On behalf of the organizing committee
Michael Engel

Organization

Organizing Committee

Ori Bernstein	Plan 9 Foundation
Michael Engel	University of Bamberg
Ali José Mashtizadeh	University of Waterloo
Daniel Maslowski	oreboot
Ron Minnich	Plan 9 Foundation
Brian L. Stuart	Drexel University
Fariborz Tavakkolian	Plan 9 Foundation
Mitch Williams	Sandia National Labs

Local Organizer

Ali José Mashtizadeh	University of Waterloo

Web Chair

Skip Tavakkolian	Plan 9 Foundation

Program Committee Chair

Michael Engel	University of Bamberg

Program Committee

Geoff Collyer	
John Floren	Plan 9 Foundation
Charles Forsyth	Terzarima Systems
Ali Mashtizadeh	University of Waterloo
Daniel Maslowski	oreboot
Richard Miller	Miller Research
Brian L. Stuart	Drexel University

Sponsoring Institutions

Plan 9 Foundation

Contents

Dr Glendarme or: How I Learned to Stop Kerberos and Love Factotum
Edouard Klein and Guillaume Gette **1**

Namespaces as Security Domains
Jacob Moody **21**

Single Level Store Application Management With 9P
Emil Tsalapatis, Ryan Hancock, and Ali José Mashtizadeh **27**

GEFS, A Good Enough File System
Ori Bernstein **35**

MIPS Rides Again
Andrew D. Gibson **45**

Plan 9 on 64-bit RISC-V
Geoff Collyer **53**

Hell Freezes Over: Freezing Limbo modules to reduce Inferno's memory footprint
David Boddie **63**

An $O(1)$ Method for Storage Snapshots
Brian L. Stuart **67**

Ghostscriptbusters
Noam Preil and Sigrid **83**

Porting the Netsurf web browser to Plan 9
Jonas Amoson **91**

Plan 9 and Inferno Go to School
Brian L. Stuart **99**

NinePea – A Small 9P Library for Arduino and Plan 9
Eli Cohen **107**

Author Index **113**

Dr Glendarme or: How I Learned to Stop Kerberos and Love Factotum

Edouard Klein[1]
Guillaume Gette[2]

[1] Beaver Labs, Paris, France
[2] Ecole Polytechnique, Palaiseau, France

ABSTRACT

Authenticating users and securing resources in a distributed system is hard because identity, ownership and permissions must be kept in sync across all machines in a domain, despite being intrinsically local properties handled by the various hosts' kernels. Yet, this is a necessary first step in piecing together a local-looking virtual system out of multiple remote resources, which in turn allows one to remove all security and networking concerns from application source code, improving both security and ease of development. We augmented factotum with the Guillou-Quisquater protocol and created a userspace server, a PAM module and a NSS configuration option which, with the already existing plan9port project, effectively backport large parts of Plan 9's distributed security model back to Linux. Compared to existing solutions such as Kerberos, our setup is trivial to install and administrate and now, one can develop a set of programs as if they would be run locally, but have them run as processes communicating transparently across multiple computers sharing a consistent access-control policy.

1. Introduction

Securely sharing resources between users on a monolithic, powerful mainframe has been made possible by, among others, the UNIX operating system and its descendants. Fine-grained access control is handled by the kernel on the basis of a user's identity and groups and of a processes' and files' owners, groups and permissions.

Our IT infrastructure is made up of a few overpowered servers and of many powerful workstations and laptops (as is most small to midsize institutions' infrastructure nowadays). We wish our users could compose their computing environments out of bits and pieces of all the machines on our network, seamlessly and securely sharing resources as if working on one big virtual mainframe.

We first got by thanks to a progressively heavier use of SSH, which appealed to us with its rock-solid security, its ability to tunnel any other network protocol and its ease of deployment, configuration and use. We nevertheless quickly considered a switch to another model because of growing pains such as:

Key management: because in a truly distributed system, every machine is liable to host both client and server processes, every machine had to know every other host's key fingerprint and every user's public key.

User management: user accounts and groups had to be kept in sync on all devices.

`uid` **mismatch:** the same user would have a different `uid` on different workstations.

Meaninglessness of port numbers: our heavy use of ssh tunneling was made more difficult as we had to avoid collisions and could not tell at a glance which ports were free and which services were being tunneled where.

LDAP and Kerberos are obvious choices to alleviate at least the first three pains points we felt with SSH. We did not, however, make the leap because of the difficulty of setup and administration and because of Kerberos' inherent reliance on sharing secrets with the application code, which is much

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

more likely to leak those secret than code specifically developed for this purpose such as `ssh-agent` or `factotum`. (see subsection 5.6).

Instead, we need a model that would let us keep the two core attributes that dictated the way we had developed and deployed our application software so far:

Network Transparency: A process gets the impression that all resources it accesses are located on the same machine as itself. No application code need to care about connecting to a remote machine or receiving connections from anywhere but localhost.

Security Agnosticism: A server process runs, on the remote machine, owned by the user who will make all subsequent requests to it from the client machine. Therefore, a server process does not need to authenticate a user nor to check whether a request is legitimate. This job is delegated to ssh (the authentication part) and to the server host's kernel (the access control part).

We turned to the Plan 9 operating system (Pike et al., 1995), whose elegant security model (Cox et al., 2002) is exactly what we need: it relies on a secure piece of software called `factotum` to negotiate authentication between the server process and the client process. Those two processes only have to forward messages between the user's `factotum` and the server host's `factotum`, with no care for the meaning of those messages, thus ensuring that implementing new authentication protocols is simply a matter of upgrading `factotum`. This is *Security Agnosticism*.

Network transparency on Plan 9 comes from the ubiquitous use of the 9P protocol (Pike et al., 2019), which allow any remote resource to be mounted in the local namespace as a part of the file system.

As we can dream of neither porting Plan 9 to our diverse hardware nor porting our application software to Plan 9, we elected instead to backport parts of Plan 9's security model to Linux. This paper is a description of our successful, yet still early, work to this end:

vrs (subsection 2.3) We created a privileged daemon called `vrs` whose role is to start server processes owned by authenticated users. It ensures the server processes can be *Security agnostic* and runs amidst the already existing `Plan 9 from user space (plan9port)` tools (Cox et al., 2018) and `v9fs` (Van Hensbergen and Minnich, 2005) (subsection 2.2), which take care of all the *Network Transparency* of our setup.

Pluggable Authentication Modules (PAM) Module (subsection 3.1) We created a PAM module that handles the conversation between a client's `factotum` and the host's trusted `factotum`. The choice of which `factotum` to trust is therefore part of PAM's configuration, where one would expect to find it on Linux. This PAM module also allows any application to authenticate users with `factotum`, although we discourage application code from handling security at all.

Name Service Switch (NSS) (subsection 3.3) We modified the `factotum` from `Plan 9 from user space` to serve a `passwd` and `groups` file. Once mounted, these files can be pointed to in the NSS configuration, allowing all machines in the network to share the same user and group database.

Guillou-Quisquater (section 4) To avoid relying on shared secrets, we were able to add support for the Guillou-Quisquater zero-knowledge interactive proof (Guillou and Quisquater, 1988) with no particular effort thanks to the clean design of `factotum`.

2. Network-Transparent, Security-Agnostic servers and clients

2.1. Illustrative example

Figure 1 shows a typical example of what we strive to achieve. The network-wide user `alice` owns the machine `Atlantis`, but she wants to have some files on `bob`'s machine `Bermuda`.

`alice`'s process `foo` should have to care about neither the fact that the files it accesses are not actually on `Atlantis` (*network transparency*) nor whether `alice`, its owner, has any rights to those files (*security agnosticism*): the kernel should block any illegal action.

It is understood that if `bob` goes rogue, he can access `alice`'s files. `bob` is the owner of all of `Bermuda`'s resources, and if she wants to use those resources, `alice` must trust `bob`.

Our example deals with sharing actual files located on a hard drive spinning inside `Bermuda`. The 9P protocol (Pike et al., 1992, 2019) can intuitively be seen as mapping filesystem operations (`open()`,`write()`, etc.) to network messages whose names begin with `T` for the request and `R` for the response (`Topen`/`Ropen`, `Twrite`/`Rwrite`, etc.).

Via 9P, Plan 9's design allow most, if not all, resources to be presented as filesystems and thus mounted locally from a remote host: keyboard, mouse, and screen, network connections, and even debugger access to a remote process. As Pike et al. (1992) put it:

> files in Plan 9 are similar to objects, except that files are already provided with naming, access, and protection methods that must be created afresh for objects.

However, even today, twenty one years later, no widely used operating system can expose its host's resources in such a unified way. In our real life use-cases, most of the software our users need expose their functionality as HTTP endpoints speaking JSON or XML.

Nevertheless, we contend that despite its seemingly marginal adoption in the world of linux user-facing applications, 9P is still a valuable target :

- securely sharing actual files is still problematic (subsection 5.4),
- virtual file systems are slowly making their way into the collective mindset thanks to FUSE (Szeredi, 2019) (subsection 5.5),
- we are working on tunneling HTTP over 9P (subsection 5.3),
- 9P enjoys very good client support on Linux thanks to `v9fs` (Van Hensbergen and Minnich, 2005) and its use in mainstream applications such as QEMU (Jujjuri et al., 2010).

This example will let us explain the role of the existing plan9port tools in the *network transparency* of our setup (subsection 2.2) and explain how our new tool `vrs` is the missing piece in charge of the *security agnosticism* (subsection 2.3). We invite the reader to follow these explanations along with Figure 1 to Figure 6, which use the following conventions:

a host is a rounded corner box with its name in the upper part,

a process is a circle inside a host

a named UNIX socket is a diamond shape tied to a host,

a TCP port is an ellipsis tied to a host,

the kernel is the box on top of the host, while

the filesystem is the box below.

The color illustrates who owns what. `alice` is blue while `bob` is red.

A dashed arrow denotes a virtual relationship: one that is made of smaller components.

A solid arrow denotes a direct relationship, with no intermediary party (e.g. it denotes a system call like `listen`).

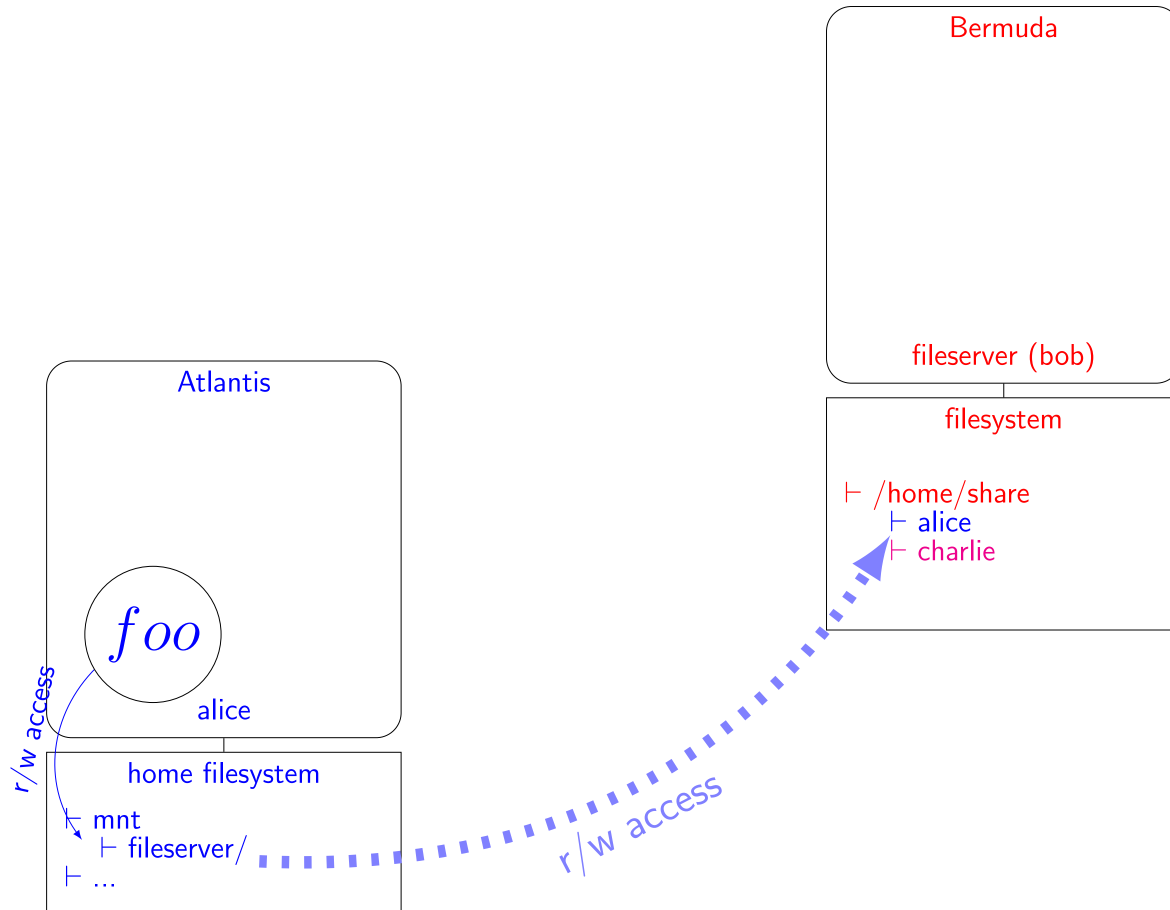


Figure 1: On `alice`'s machine `Atlantis`, a process `foo` owned by `alice` accesses `alice`'s files on `bob`'s machine `Bermuda` as if those files were on `Atlantis`. `Bermuda`'s kernel offers the same protection as `Atlantis`'s kernel would.

2.2. Network Transparency with plan9port

2.2.1. u9fs owned by alice

We need to translate what foo does to Atlantis' filesystem into similar actions on Bermuda's filesystem. Therefore there must exist a process on Bermuda tasked with receiving 9P messages and translating them into system calls impacting Bermuda's file system.

The existing u9fs fileserver does exactly that. It was designed to be *network-transparent*, as it reads and writes 9P on its standard input and output. In subsection 2.3 we will explain how the u9fs process is also *security agnostic*, because it is owned by alice.

This means that Atlantis' and Bermuda's kernels must agree on who alice is, including her numeric uid[1], groups, and their numeric gid. In lieu of LDAP, we make both kernels refer to our modified factotum running on a central authentication server (see subsection 3.3). These components are drawn on Figure 2.

2.2.2. A factotum dialog

Before u9fs can be started on Bermuda under her identity, alice must authenticate on Bermuda. Following the Plan 9 security model (Cox et al., 2002), this is just a matter of having alice's factotum talk with the factotum Bermuda sees as authoritative.

factotum's communication and authentication protocols do not require any intermediary to interpret the messages. Any application intermediary only forwards messages between a factotum client and a factotum server to perform the authentication. Thus, alice just needs to run a client instance of factotum to authenticate on Bermuda, see Figure 3

2.2.3. v9fs and srv

To translate foo's system calls into 9p messages, we use the kernel layer v9fs (Van Hensbergen and Minnich, 2005), via plan9port's mount command.

v9fs intercepts system calls made by foo to access the file system, translates them in 9P, then writes the 9P messages to a socket. Which socket to connect to is decided when invoking the mount command. Here Figure 4 shows a named UNIX socket on Atlantis.

The 9P response messages read by v9fs from the socket determine the system calls outcomes, with foo being unaware of all this machinery.

Listening on the socket is the plan9port program srv. It first sends the Tauth message to the server, then relays messages back and forth between the server and alice's factotum on Atlantis, until alice's factotum tells it to stop.

In a second phase -if the authentication succeeded- srv creates the socket and simply forwards 9P messages back and forth between the server and whoever (here, v9fs) connects to the UNIX named socket, as shown in Figure 4.

2.2.4. 9pserve

The TCP socket on Bermuda that Atlantis' srv connects to is created by the plan9port utility 9pserve. Its man page explains:

> [...] the Plan 9 kernel multiplexes the potentially many processes accessing the server into a single 9P conversation. The user-level server need not concern itself with how many processes are accessing it or with cleaning up after a process when it exits unexpectedly. On Unix, 9pserve takes the place of the Plan 9 kernel, multiplexing clients onto a single server conversation and cleaning up after clients when they hang up unexpectedly.

This 'single server conversation' can not be directly forwarded to alice's u9fs on Bermuda. Other clients, such as one owned by charlie, for example, may be connected to the same port, too. charlie's messages need to be forwarded to charlie's instance of u9fs on Bermuda. Furthermore, 9pserve is unable to authenticate a connection. It expects the server to handle Tauth messages. See Figure 5

[1] Plan 9 uses actual strings instead of numbers

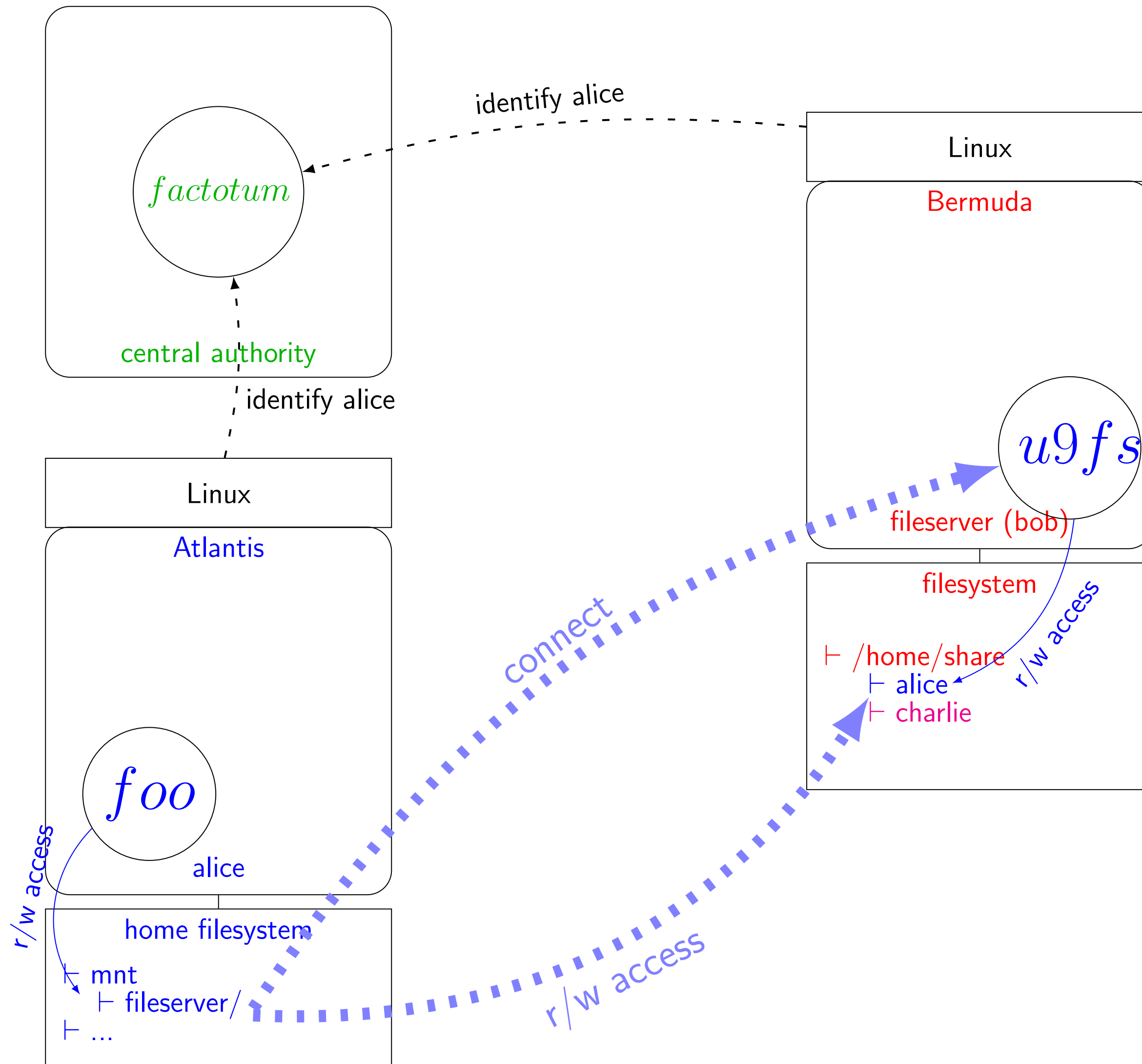


Figure 2: The `u9fs` process actually making changes in `Bermuda`'s file system on behalf of `foo` is owned by `alice`. This means that both kernel refer to the same central authority to agree on who `alice` is.

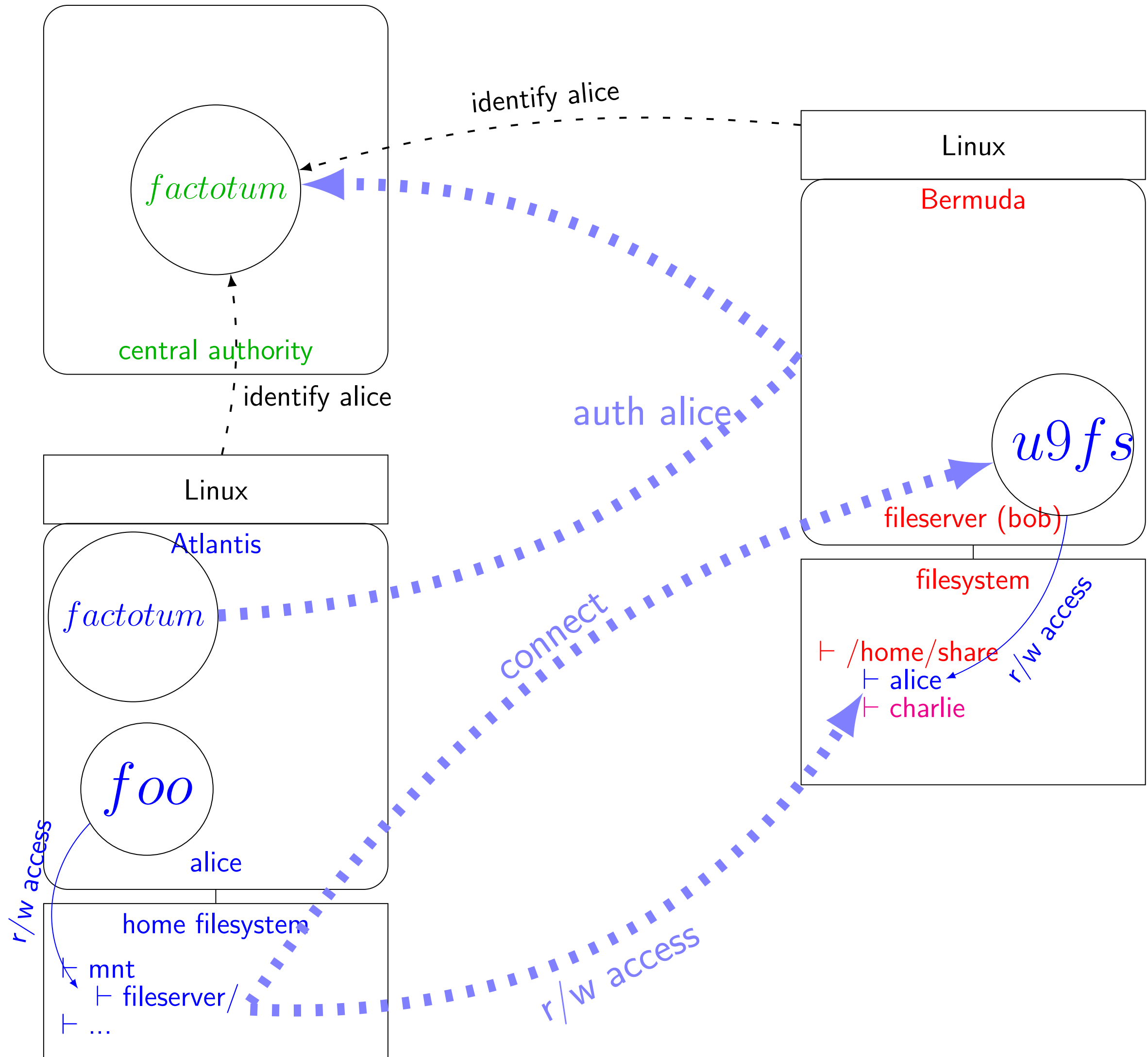


Figure 3: Authenticating `alice` on `Bermuda` is
a conversation between two `factotum` processes

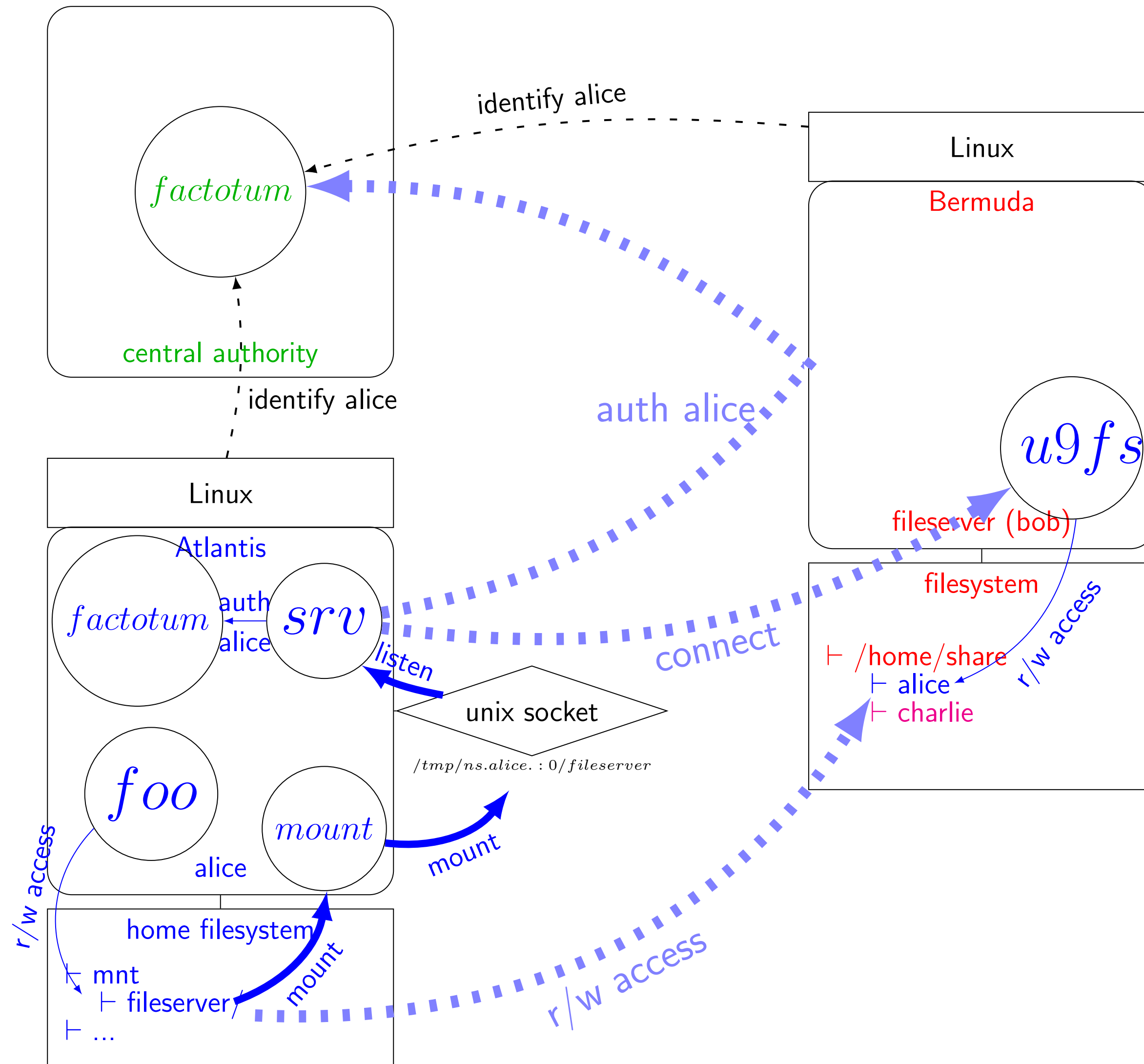


Figure 4: `srv` authenticates `alice` by relaying information between her `factotum` and the server, and then relays 9P messages between the server and the UNIX socket. `v9fs` talks 9P to the socket to offer a virtual directory on `Atlantis`' file system

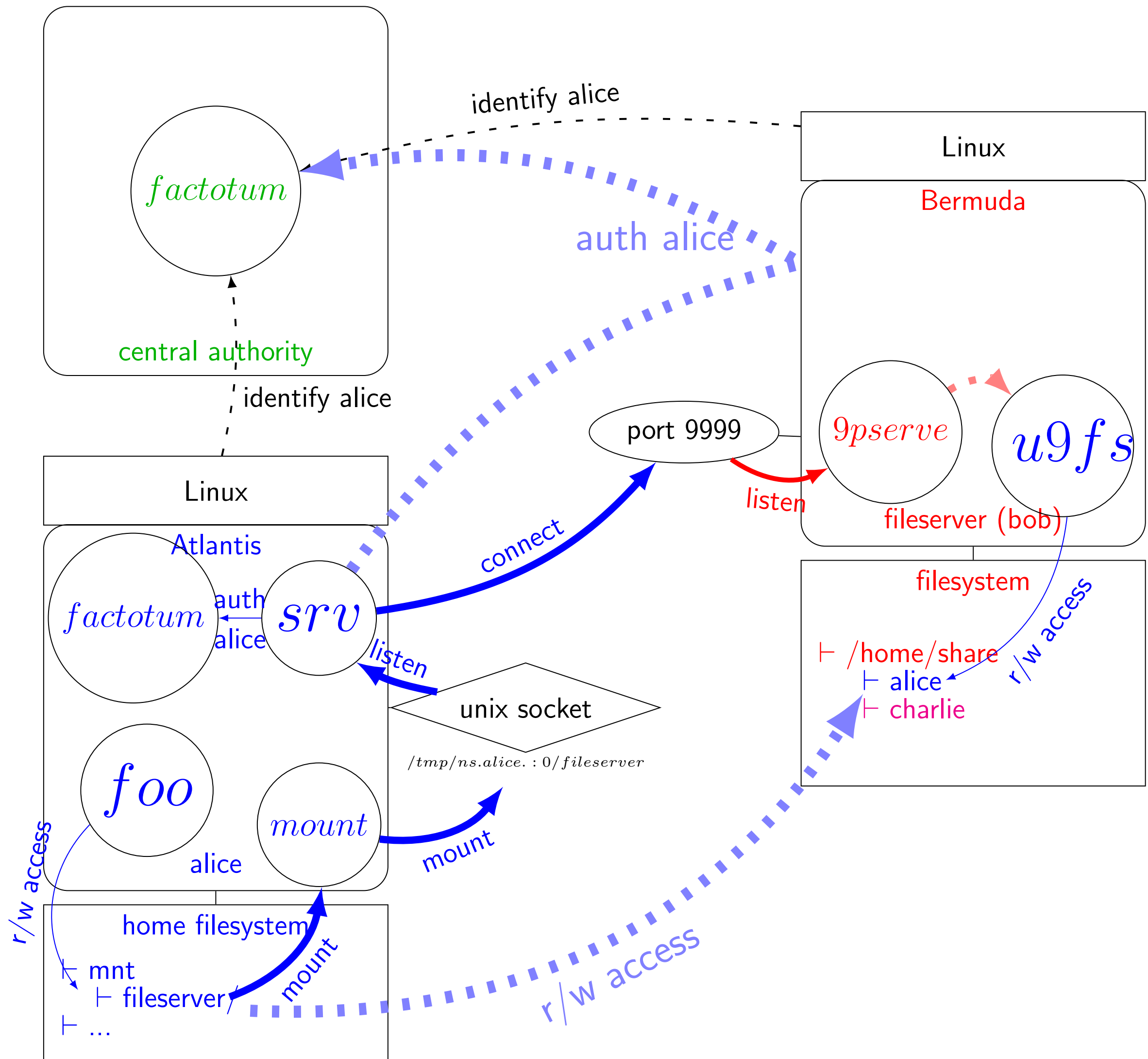


Figure 5: On Bermuda, `9pserve` multiplexes multiple clients (only alice is shown here) connections. Each client access their own server instance.

2.3. Security Agnosticism with vrs

2.3.1. What is vrs ?

On `Atlantis`, `srv` first authenticates as `alice` by relaying messages between her `factotum` and the server, and then simply relays 9P messages between the client process `foo` and the server.

One of our contributions is creating the `vrs` utility that mirrors this behavior on `Bermuda`: `vrs` first authenticates `alice` by relaying messages between `Bermuda`'s trusted `factotum` and the client, and then simply relays 9P messages between the client and the server process `u9fs`. See (Figure 6).

As Figure 6 shows, while the tools from `plan9port` were enough to ensure the *network transparency* of our setup, `vrs` is needed to ensure its *security agnosticism*. Whereas there are as many `srv` as there are clients, there is only one `vrs`. `vrs'` job is to authenticate each user and start a dedicated instance of `u9fs` owned by the corresponding user, then demultiplex the 9P flow coming from `9pserve` and forward the messages to the correct server instance.

2.3.2. vrs implementation

In Figure 7 and Figure 8, we show how three clients, Alice, Charlie and Dave, can send 9P messages to Bermuda and have their messages processed by each their own Network-Transparent, Security-Agnostic `u9fs` instance. We invite the reader to follow this explanation along with the circled letter spots (e.g. Ⓐ) on both figures.

Figure 7 shows three incoming messages:

- Alice's message is a Tauth, meaning she is not logged in yet. She is starting the authentication process.
- Charlie's message is a Tattach, meaning he has been successfully authenticated and is actually mounting the filesystem on his machine.
- Finally, Dave's message is a Twrite: After authenticating and attaching, Dave is actually using the filesystem.

9pserve These messages are partially rewritten by `9pserve`, Ⓐ, so that there is no collision between the values chosen by the various clients for their:

tag: When it receives an R-message, the tag lets the user know which T-message this R-message is a response to.

fid: A number that identifies a file. The server-side book keeping number is the qid.

afid The fid given in a Tauth becomes, after a successful authentication, a short-lived capability. It is called an afid. The server maintains a corresponding aqid.

`9pserve` maintains a mapping Ⓑ between incoming and multiplexed tags and (a)fids, in order to edit and demultiplex the returning R-messages to their correct clients Ⓝ.

Our new program `vrs` receives this stream of rewritten messages on its standard input Ⓒ, being oblivious to their respective emitters' location (*Network-transparency*).

Its job is to make sure these messages are legitimate, and demultiplex them by routing them to the correct subprocess.

Twrite The simplest example is Dave's Twrite message. `vrs` checks that the fid is mapped Ⓓ to an active and valid afid Ⓔ, which means that this Twrite comes after a successful Tauth and Tattach (and obviously one Topen, and probably some Twalk). It is therefore routed unmodified to Dave's subprocess Ⓕ, as it contains no cryptographic or authentication information.

Similarly, the Rwrite response from Dave's server Ⓘ is forwarded unmodified Ⓜ to Dave, save for `9pserve`'s demultiplexing Ⓝ, according to its mapping Ⓑ.

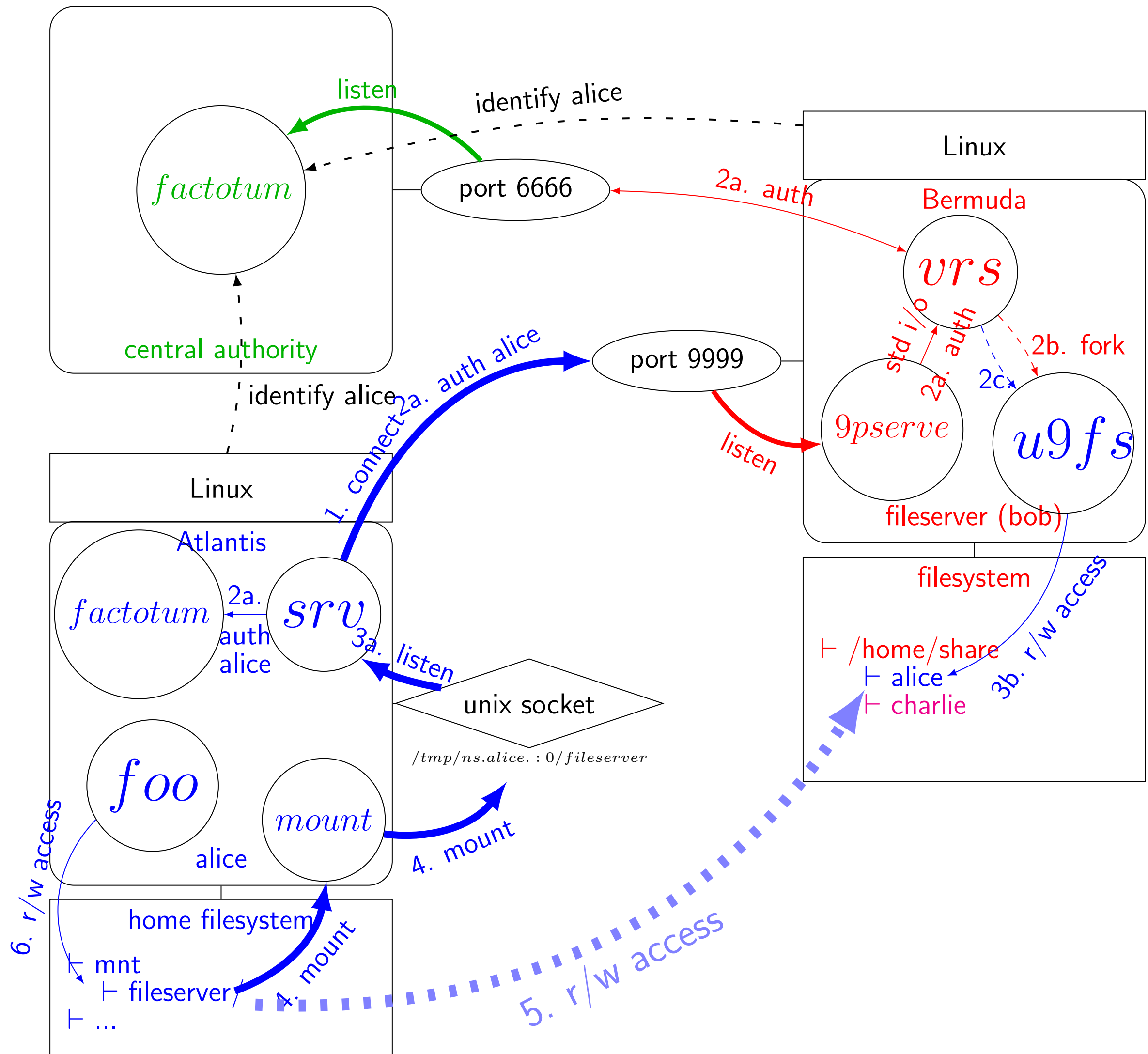


Figure 6: Our tool `vrs` is needed to ensure the security agnosticism of the server process (here, `alice`'s `u9fs` in `Bermuda`)

Tattach Upon receiving Charlie's Tattach message, `vrs` checks the given afid's PAM context Ⓔ (see subsection 3.1), and, if valid,

- forks Ⓖ,
- switches to uid 0, having temporarily dropped root privileges during normal operation,
- switches to `charlie`'s `uid` and `gid` as obtained through NSS (see subsection 3.3), permanently dropping root privileges in doing so,
- execs the 9P server, the main `vrs` process still keeping tabs on its standard input and output.

Once Charlie's server is up and running, `vrs`' fid mapping Ⓓ is updated to map `charlie`'s attach fid (here, 3) to its active and valid afid Ⓔ (here, 1) and to descriptors of `charlie`'s server's standard input and output.

The Tattach message is then stripped of its afid, which is replaced by the special value NOFID, which tells Ⓖ the newly started server process for `charlie` that no authentication is required. Thanks to `vrs`, Charlie's server is *security agnostic*, but being owned by `charlie`, it is protected by `Bermuda`'s kernel from outside interference and prevented from doing anything illegal.

It is of utmost importance to control the user-chosen afid's lifecycle. As Cox et al. (2002) put it:

> since [an afid] act like a capability, it can be treated like one: handed to another process to give it special permissions; kept around for later use when authentication is again required; or closed to make sure no other process can use it.

Its confidentiality during transport is ensured by TLS (see subsection 2.4), and the end of its lifecycle is dealt with by `vrs` intercepting any Tclunk message for an afid (as the server would not understand them anyway, being *security agnostic*) and unmapping the corresponding afid from Ⓔ. Any subsequent request referring to it either directly (Tattach) or indirectly (any other T-message with a fid mapped to this afid in Ⓓ) would fail.

Charlie's server's Rattach Ⓙ is transferred Ⓜ to Charlie, modified only for the demultiplexing Ⓝ, Ⓑ of its tag.

Tauth Alice's Tauth message makes `vrs` create a new PAM context in its mapping Ⓛ. The AQID returned in the Rauth Ⓚ message will let Alice's `factotum` run through its conversation with the `factotum` trusted by `Bermuda`: any subsequent Tread or Twrite messages sent by Alice on fid 0 will be multiplexed to fid 4 by `9pserve` and forwarded by `vrs` through our PAM module (see subsection 3.1) to `Bermuda`'s trusted `factotum`. `vrs` stays blissfully ignorant of the content of those read and write operations, which allowed us to edit `factotum` to add a new authentication protocol (see section 4) without modifying anything else.

Until the PAM context switched to SUCCESS, all Tattach on fid 4 will receive an Rerror from `vrs`.

Until a successful Tattach happens, no server is even started for `alice`, thus reducing the attack surface.

2.4. TLS

Because afid are capabilities, 9P messages should not be transmitted in the clear. Yet, so far we refrained from talking about transport security. The reason is that this would have made Figure 1 to Figure 6 quite hard to read.

We use `stunnel` (Trojnara, 2019) to authenticate the server to the client and protect the confidentiality of the 9P messages against eavesdroppers.

On `Bermuda`, it is `stunnel` who is actually listening on port 9999. It decrypts the incoming messages and forward them to `9pserve`, who is listening on a UNIX named socket. That way no external attacker can connect to it, and the name of the socket is more descriptive than a port number.

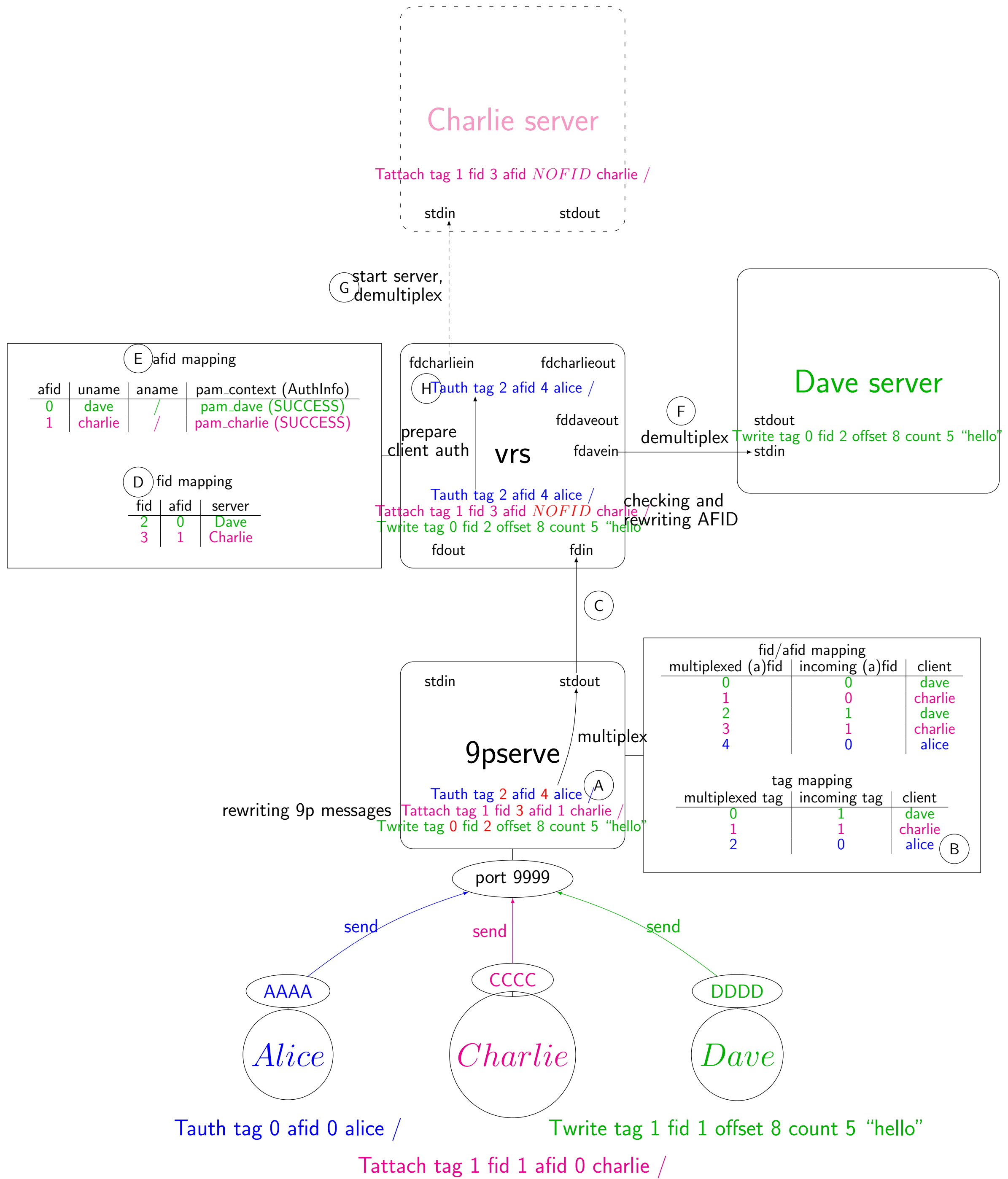


Figure 7: `9pserve` multiplexes 9P messages from multiple clients to a single conversation with `vrs`. `vrs` handles authentication and demultiplexes the client's messages to the corresponding *network transparent*, *security agnostic* server.

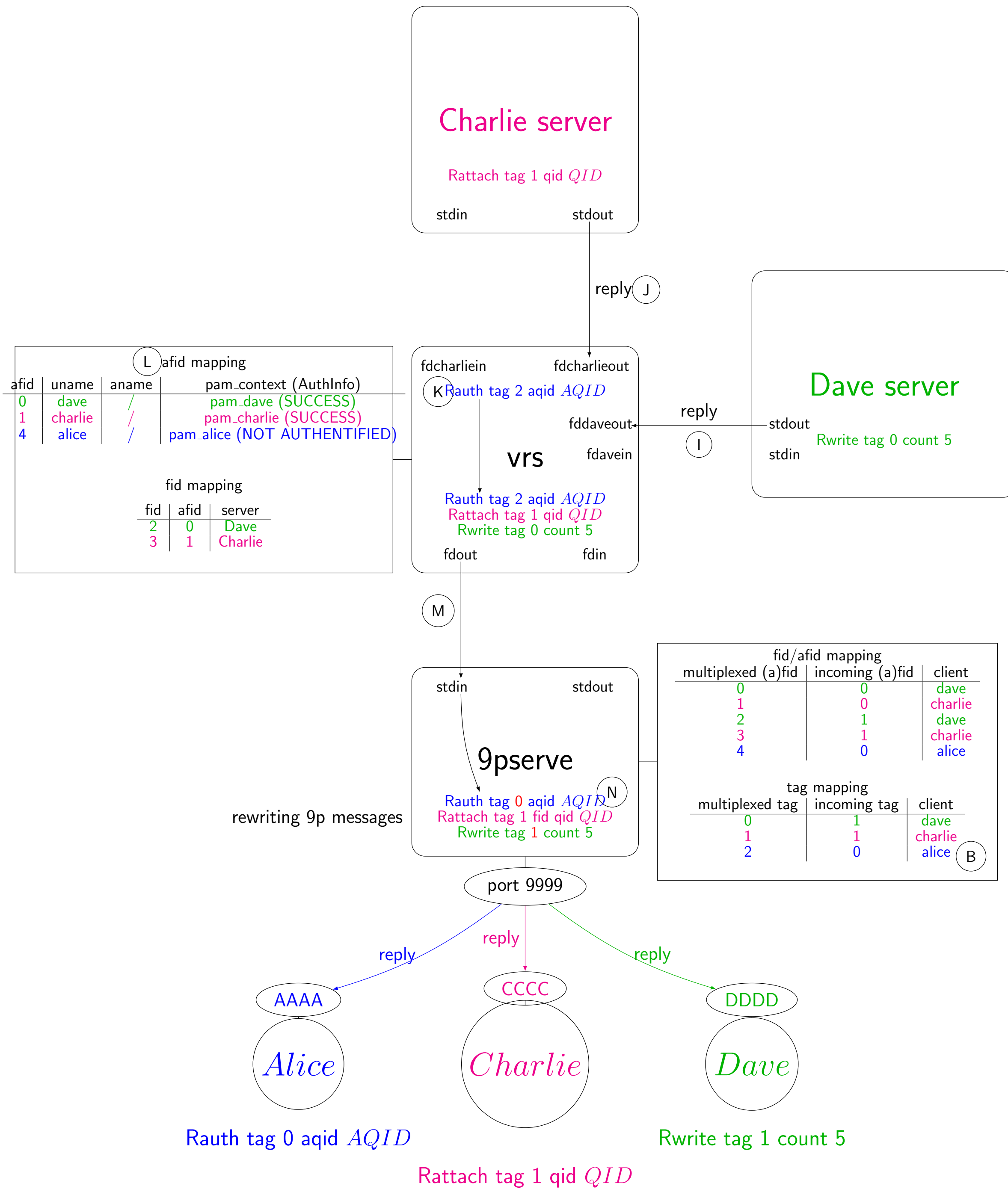


Figure 8: R-messages are forwarded back to the clients. Authentication related messages are coming from our PAM module, other messages come from the client owned server.

On `Atlantis`, `srv` is not connecting to port 9999 of `Bermuda`, but to a named UNIX socket, served by another instance on `stunnel`, who is actually connecting to `Bermuda`.

We installed our own root certificate on all machines, so we can renew a server's certificate by simply signing it and installing it on the server machine, without having to tell all the clients about this change (as we had to when a server's host key changed when we used SSH).

3. System administration

Configuring a Linux system to trust a central `factotum` is as easy as establishing a secure connection to an authoritative server of one's choosing (we use `stunnel` for that, see subsection 2.4) and mounting `factotum`'s virtual filesytem, *e.g.* in `/etc/factotum/`. We use the service managers of our systems to perform this task at startup.

3.1. PAM and factotum

`vrs` does not directly forward messages to the server's trusted `factotum`. Instead it calls our PAM module.

This makes implementing `vrs` a bit simpler, and is the expected way of adding an authentication mechanism on Linux. The main advantage is that it defers the choice of the authoritative `factotum` to the system wide configuration of PAM, where an administrator would expect to find it, and where it is protected from malicious interference.

This module will communicate with the authoritative `factotum` by reading and writing in the `rpc` file of the directory where `factotum` has been mounted.

The existence of this PAM module also let other application use factotum as an auth mechanism. We strongly discourage application code from dealing with security, but if there is no other way, a little bit of PAM boilerplate is probably the best way to go.

3.2. Factotum

Managing users is a simple matter of maintaining a configuration file of all users and their keys in a secure location (Plan 9 uses the `secstore`), and dumping it in the central authority's `factotum`'s `ctl` file at startup.

Instead of storing secrets in the central authority, we elected to only store the users' public keys (see section 4), and have them store their private keys in a secure location, giving them to their own `factotum` instance when needed.

The `ctl` file understands and exposes records as lists of key-value pairs, *e.g.*:
`key proto=p9sk1 dom=anonymous.lan user=alice...`
Because `vrs` uses the `p9any` protocol, the client and the authoritative factotum will negotiate until a matching key is found between the client and the server.

Local administrators having access to the `ctl` file remotely is no big deal, as they only gain read access. Write access is only for the owner of the `factotum` process on the central authority server.

3.3. NSS and factotum

The last problem to solve is having all machines synchronize their users and groups database, which would normally be done by pointing to LDAP in the NSS configuration.

Instead, we modified `factotum` to make it serve two virtual files: `passwd` and `groups` with the same format as their namesakes in `/etc`.

These virtual files expose the same information as `factotum`'s `ctl` file, in a different format.

We then just use the `altfiles` module to point NSS to where the authoritative `factotum` is mounted.

That way, user management (name, uids, groups, keys, etc.) is done all in one place: by writing the `ctl` file of the authoritative `factotum` (see subsection 3.2).

4. Adding the Guillou-Quisquater ZKIP to factotum

A lot of Plan 9's file servers use the `p9sk1` protocol, which relies on shared secrets held by a trusted third party, the authentication server. The authentication server uses a file server's secret to craft tickets relayed by the client's `factotum` to the server to prove the user's identity.

As a matter of policy, we don't like relying on sharing secrets. We elected instead to keep using one RSA key pair per user like we did when using SSH, and have the role of a central authority be endorsed directly by a `factotum` process remotely mounted by all server machines which chose to trust it (see section 3).

`factotum` already knows about RSA keys. We just added a new protocol that can use them, the Guillou-Quisquater zero knowledge proof (Guillou and Quisquater, 1988). By adding an RSA private key to its `factotum` with the `service=gq` attribute, a user can authenticate using this protocol to any service trusting a central authority `factotum` to which the corresponding public key has been added.

Once both `factotums` agrees on a key with the `service=gq` attribute, the client `factotum` sends a *commitment* computed from a random number and the public key to the server. The server answers with a random number of its own, which constitutes a *challenge*, and the client then sends a *response* computed from the initial random number, the private key and the *challenge*. The server then checks that this *response* is the same as another number computed from the *commitment*, the *challenge* and the *response*. If so, the client has proved its own knowledge of the private key (or is very, very lucky, but the process may be repeated) without disclosing information about it.

Thanks to the clean design of Plan 9's security model, adding this new protocol was just a matter of modifying `factotum`. The authentication transactions are sent back and forth unaltered between both `factotum` by `srv`, `9pserve`, and `vrs` (see Figure 6), which therefore need not be altered.

5. Related and future Works

5.1. Setuid

On Linux, a process asking a PAM module before switching its owner is just being courteous. There is nothing preventing a privileged process from switching its `uid` at will, which is why giving this power to application code (more likely to have bugs and security flaws) is a bad idea.

Privileged processes can now use Linux' capabilities system (same name, but not exactly the same thing as the capabilities we referred to when discussing 9P) to prevent themselves from doing certain classes of privileged actions. OpenBSD's `pledge()` has a similar aim and in our humble opinion a cleaner interface. In both cases, if a process can call `setuid()`, it can `setuid` to anyone.

Plan 9 is like France on the morning of August 5, 1789 in that privilege does not even exist anymore. Switching the file server's owner to `alice` is done by the host owner's `factotum` after she has earned its trust. `factotum` writes a capability to a file called `caphash` and gives it to the server process, which then writes it to a file called `capuse` making the kernel changes the server process' `uid` to `alice`. There is nothing else this capability is good for.

On Plan 9, some servers are not completely security agnostic because they must still use `auth`'s functions to switch their identity using this fine grained capability system, but

- the boilerplate is minimal (less than 10 lines, error management included),
- the server gains absolutely no other right than changing its owner to *e.g.* `alice` within the allotted time frame, whereas on Linux it could switch to anyone at anytime and do even more unless Linux' capability system is used to restrict its privilege somewhat (an uncommon case so far in most of the software we use),
- wrappers such as `rexexec` will switch identity before `execl`-ling to the truly security agnostic server.

In practice, some servers expecting UNIX clients or using deprecated forms of authentication such as `telnetd` or `ftpd` are not security agnostic at all. Others (e.g. `ctdfs`) are completely security agnostic and their actions will be limited by the kernel, given their owner's `uid`.

Plan 9's elegant capability-based mechanism has been ported to Linux (Ganti, 2008). Were this to be mainlined, we would use it on our machines and immediately thereafter modify `vrs` to use this

system and therefore run unprivileged. We then would hunt all remaining privileged software and, if possible, convert them to use the `cap*` files as well.

5.2. Namespaces

On Plan 9, processes build their own composite view of the network-wide resources they need. This is done by mounting the needed services in one's so called *namespace*. This *namespace* is a per-process property. The `ns` utility can display the current *namespace* of a process as a script that can be run to recreate this *namespace* elsewhere. The commands are descriptive enough for the output of `ns` to also be used as documentation of the process' setup. Indeed it is almost always the name of a remote service that is used instead of the port number this service is listening on.

Our heavy use of SSH tunneling and sshfs led to port numbers or mount points conflicts. Port numbers made setup scripts hard to read and required careful synchronization between clients and servers.

Our switch to `vrs` made things a little bit better, as services can be named, making reading setup scripts easier. Also, `plan9port` provides the `namespace` utility, which can reasonably emulate the toe stepping avoiding functionality of Plan 9's per process namespace, but can not provide the secure isolation Plan 9 processes enjoy from one another.

Linux lacks this kind of process isolation. All processes share the same file tree, one has to be privileged to open a low numbered port, etc. This is being remediated by the slow but growing penetration of features like linux namespaces, cgroups and containerization. Albeit more awkward to use than the native namespaces in Plan 9, we think these features are exciting steps in the right direction.

Sadly, we know from successfully running most of the software our users want on Linux lxzones in SmartOS, which do not support complex namespace operations, that very few pieces of software make use of these functionalities, despite them having been introduced more than one decade ago.

Nevertheless, in the last couple of years, we achieved an almost Plan 9-like level of process isolation through the use of GNU Guix Courtès (2013), either as a package manager on top of our user's Linux distribution of choice, or as the whole system. In the process of creating a reproducible package management system, GNU Guix' creators have made first-class citizens of those seldom-used namespace, cgroups, and containerization features of Linux. By prefixing a command with `guix shell --container`, or simply using `call-with-container` in a service description file, one can enjoy on Linux the same level of process isolation as on Plan 9.

5.3. 9P vs. HTTP

Most of the software our users want expose their functionality through HTTP endpoints. Claiming RESTful properties (but often failing in subtle ways), when they implement authentication they do so by authenticating every transaction, awkwardly reimplementing the access control logic already available to them through the kernel.

We intended to work on a HTTP over 9P bridge, naturally mapping the HTTP endpoint to a file of the same name, with read calls translated as GET requests, write calls as PUT, POST or PATCH, etc.

Using this bridge, `alice` can authenticate, attach, walk to a endpoint of interest and open it with specific rights (read only, for example). The corresponding `fid` acts as a *de facto* capability and can be given away to another process for it to complete a very specific and narrow task, with no access whatsoever to anything else than what is strictly needed. `alice` can close the file and thus repudiate the capability as soon as she wants. This kind of capabilities is more fine grained and easy to use than any RESTful software access control we have seen in the wild.

On the backend side, if the software can have multiple instances of itself run concurrently on the same data files, we use `vrs` as described in this paper. The backend server's kernel deals with access control using the usual UNIX file permissions and the software is run *security agnosticly*.

This case is rare, however, and we more often than not end up having to run a single instance (typically database software like Postgresql or ElasticSearch do not support running multiple instances on the same data). Because our users are well behaved we run one instance per project, which gives them the ability to shoot one another in the foot. The alternative is to either somehow enforce access control at the bridge level, which increases the complexity of the bridge code and necessitates a way to tell the bridge about the permissions, or map with each software's own access control mechanism, which takes us back to square one.

Despite not getting any work done on this particular topic in the last couple of years, we still think this is a worthwhile avenue of research.

5.4. Sharing actual files

We tried NFS, AFS, SSHFS and Samba to share files. In terms of security and ease of use, the most serious contender to the solution exposed here was SSHFS, which we finally rejected because of `uid` management. NFS and AFS needs Kerberos (see subsection 5.6) to be secured, there is no AFS server for OpenBSD (which runs on one of our biggest fileserver) and Samba is very awkward to use without an Active Directory which we don't have.

All these solutions are battle tested and solve the file sharing problem day in day out everywhere across the world.

Comparatively, `srv` + `u9fs` may suffer from performance issues (although we have not run into that yet) and bugs, but are easier to install and expose a truly consistent access control policy to all users on the network. We are working on a BSD port which is currently at the experimental stage, not deployed on all our infrastructure.

Because sharing filesystems has become so easy, most of our workflows now rest on spools, sentinel files, named pipes, etc. Running those workflows locally or in a completely distributed way requires absolutely no change in the code, just a little bit of setup (mounting the remote resources) beforehand.

5.5. FUSE and 9P servers

The idea of exposing resources as virtual file systems is gaining traction on Linux thanks to FUSE. Applications such as sshfs or GlusterFS deal with data storage as "normal" files would, other such as encfs are overlays over data stored on the disk, and finally some other like perkeep (formerly camlistore), or Plan 9's plumber and acme expose an API, as modern software usually does nowadays with HTTP endpoints.

Plan 9 and Inferno are full of virtual filesystems to access things as diverse as *e.g.* emails, web pages or Lego Mindstorms. With Inferno especially, those can be started by `vrs` without having to port them to UNIX. One just starts, for each client, an embedded Inferno system running the service.

5.6. Kerberos, LDAP and SSH

We argue that our solution is easier to install, configure and administrate than the industry standards, Kerberos and LDAP. Furthermore, Kerberos is inherently and irremediably flawed in that it requires application code to handle security by sharing secrets with the Ticket Generating Service. This application code is likely not to be always up to date, or to have some flaw that would leak memory, thus granting an attacker access to a powerful secret.

By contrast, our solution handles security in a small number of well defined places that are suitably hardened (*e.g.* `factotum` will not let itself be debugged, nor will it swap its RAM (Cox et al., 2002)), while the servers and clients are built as completely *security agnostic*.

Furthermore, (a)fid are short lived and narrow capabilities, that will grant an attacker a lot less power than a Kerberos (Ticket-Generating) Ticket should he get ahold of them.

Finally, thanks to our implementation of the Guillou Quisquater protocol, our security may rest on asymmetric cryptography (our personal preference) instead of sharing secrets, once again limiting the attack surface.

6. Conclusion

Striving to bring the simplicity and elegance of Plan 9's security model to UNIX despite the latter's inherent limitations, we devised a distributed authentication setup where the only privileged process is our `vrs` daemon. This let us push all security and network code out of both client and server software, and places the burden of access control back on the kernel's shoulders, where it belongs.

Now, being entirely unprivileged, a buggy or compromised application process is extremely limited in the damage it can do. Furthermore, application code is now easier to write and configure, because it need care about neither networking nor security. This in turn reduce the number of bugs an attacker can exploit.

Through our use of the 9P protocol, application workflows run unmodified whether they are run locally or in a composite environment made up of resources from all over our network. Our modification of `factotum` as a central authority for the user and groups database ensures that the access control policy is consistent all over the network and that administration is dead simple.

This scheme has been deployed on our institution's network for weeks. Some applications still require SSH tunneling, but work is well underway on a HTTP over 9P wrapper, which will bridge the gap and let us make almost all resources we care about available as 9P servers.

Source code The code and the source of this paper can be found at `https://gitlab.com/edouardklein/vrs`

Thanks and Acknowledgements We would like to thank *Messieurs les Colonels* Duvinage et Nollet, for letting us enough leeway to work on our research despite pressing operational needs.

We owe a great deal to our friends and colleagues at the C3N, especially Jérôme Hénon, for their help in setting up the machines, and their willingness to try out strange software from the nineties (to be fair, a suprisingly high share of our hardware was from the nineties as well ;-).

References

Courtès, L. (2013). Functional package management with guix. `https://arxiv.org/abs/1305.4584`.

Cox, R. et al. (2003-2018). Plan9 from user space.

Cox, R., Gross, E., Pike, R., Presotto, D., and Quinlan, S. (2002). Security in plan 9. In Association, T. U., editor, *Proceedings of the 11th USENIX Security Symposium*, Available online: `https://9p.io/sys/doc/auth.html` (2018/04/04). Bell Labs and MIT LCS and Avaya Labs.

Ganti, A. (2008). Plan 9 authenitcation in linux. *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, 42:27–33. Available online: `http://gsoc.cat-v.org/people/ashwin/p9authLinux.pdf` (2018/04/04).

Guillou, L. C. and Quisquater, J.-J. (1988). A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In Guenther, C., editor, *Advances in Crytology*, pages 123–128. Available online: `https://cseweb.ucsd.edu/~mihir/papers/gq.pdf` (2018/07/07).

Jujjuri, V., Van Hensbergen, E., Liguori, A., and Pulavarty, B. (2010). Virtfs–a virtualization aware file system pass-through. In *Ottawa Linux Symposium (OLS)*, pages 109–120.

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from bell labs. *Computing systems*, 8(2):221–254.

Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. (1992). The use of name spaces in plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pages 1–5. ACM.

Pike, R., Thompson, K., Presotto, D., Winterbottom, P., and al. (1987-2019). *Introduction to the Plan 9 File Protocol, 9P*. Alcatel-Lucent, Available online: `http://9p.io/magic/man2html/5/0intro` (2019/01/04).

Szeredi, M. e. a. (2010-2019). Fuse: Filesystem in userspace. `http://fuse.sourceforge.net`.

Trojnara, M. (2019). Stunnel. `https://www.stunnel.org/`.

Van Hensbergen, E. and Minnich, R. (2005). Grave robbers from outer space. using 9p2000 under linux. In The USENIX Association, editor, *USENIX Annual Technical Conference*, Available online: `https://www.usenix.org/legacy/events/usenix05/tech/freenix/full_papers/hensbergen/hensbergen.pdf` (2018/07/30).

Namespaces as Security Domains

Jacob Moody, moody@posixcafe.org

ABSTRACT

We aim to explore the use of Plan 9 namespaces as ways of building isolated processes. We present here code for increasing the ability and granularity for which a process may isolate itself from others on the system.

Introduction

In a Plan 9 system the kernel exposes hardware and system interfaces through a myriad of filesystem trees. These trees, or sharp devices, replace the functionality of many would be system calls through use of standard file system operations. A standard Plan 9 environment is comprised of a composition of these individual devices together, the collection of such being the processes namespace.

With these principles it is quite easy for a process to build a slim namespace using only what it may need for operation. This could be done in service to reduce the "blast radius" of awry or malicious code to some effect. But to be fully effective a process must also be able to remove the ability to bootstrap these capabilities back. We will explore different ways of building isolated namespaces, their pitfalls, ways to address those issues, along with new solutions.

Outside World

There have been many solutions for sandboxing within the UNIX™ world. There are more classical approaches such as `zones`, [Price04] and `jails`, that all provide an abstraction of building some number of smaller full unix boxes out of a single physical host. However these interfaces are presented more as a systems management tool, the mechanisms for which an administrator creates and manages these resources is unergonomic to use on a per-process basis. Instead it seems more the fashion now to isolate specific pieces of the system, and expect it possible that each process on the system may choose to manage its environment. The most successful execution of this idea in the wild is the OpenBSD project's `unveil` and `pledge` [Beck18] system calls, allowing a processes to cut off specific parts of the filesystem or system call interfaces. Linux namespaces [Biederman06] implement this idea by allowing a process to fork off private versions of specific global resources. In both these cases the sandboxing of a process is through gradual steps, removing potentially dangerous tools one by one.

Existing Work

Let us first define the resources we are restricting access to. The aforementioned gradual solutions provide ways in which a process can remove itself from specific kernel interfaces. In plan9 the kernel exposes almost all of its functionality through individual filesystems. These devices are accessed globally by prefixing a path with a sharp('#'), and have conventional places they are bound within the namespace.

A processes namespace in plan9 is typically constructed using a namespace file. These files are a collection of namespace operations formatted as one would expect to see

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

them in a shell script. They typically begin by binding in some number of sharp devices in to their expected location.

```
bind #d /fd
bind -c #e /env
bind #p /proc
bind -c #s /srv
```

Then using the globals provided, in particular /srv, to bring in the rest of the root filesystem. A process can at any point choose to construct itself a new namespace, but it must do so when changing users. This is done in part to ensure that each filesystem that the program would like to use has their chance to authenticate and be notified. Because this information is only exchanged on attach, the new user must construct a namespace from scratch.

Many programs, like network services, wish to drop their current user and become the special user none user on startup, and in doing so must rebuild their namespace. The conventional default namespace files used is /lib/namespace, but most programs allow the user to specify an alternative with a flag. It is here that we already can approximate a chroot style environment by changing the root filesystem used in a namespace file.

```
bind #s /srv
mount /srv/myboot /root
bind -a /root /
```

By having another filesystem exposed in /srv/myboot and modifying the provided namespace file, we've allowed this process to work within an entirely separate root filesystem.

RFNOMNT

The issue in using these namespaces as security barriers is that there is nothing preventing a process from bootstrapping a resource back. While our example code places a different root filesystem in the namespace, nothing is preventing that process or its children from potentially rebootstrapping the real root filesystem back. For this issue there is a special rfork flag RFNOMNT the prevents a process from accessing any almost any sharp device of consequence. This is done by preventing a process from walking to a device by its location within '#'. This allows existing binds of resources to continue working within the namespace but restricts a process from binding in new resources from the kernel.

While effective we found this to be too large a hammer in practice. Doing as its name implies RFNOMNT also prevents a process from performing any mounts or binds. This in practice creates a single point in time in which a process gives up all of its control, instead of the idealized gradual process. This makes it quite hard to make use of in practice, only a singly program in a chain may be the one to invoke RFNOMNT or must hope that no other program further in the chain may want to make use of its namespace. The interface itself feels very clunky, there is a nice gradual addition of these kernel devices to the namespace why must the removal be all at once?

Chdev

We propose a new write interface through /dev/drivers that functionally replaces RFNOMNT. /dev/drivers now accepts writes in the form of

```
chdev op devmask
```

Devmask is a string of sharp device characters. Op specifies how devmask is interpreted. Op is one of

&	Permit access to just the devices specified in devmask.
&~	Permit access to all but the devices specified in devmask.
~	Remove access to all devices. Devmask is ignored.

This allows a process to selectively remove access to sections of sharp devices with quite a bit of control. In order to mimic all of RFNOMNT's features, removing access to `devmnt`, which is not normally accessible directly, disables the processes ability to perform mount and bind operations.

For the implementation, we extended the existing RFNOMNT flag attached to the process namespace group in to a bit vector. Each bit representing a index into `devtab`. The following function illustrates how this vector is set.

```
void
devmask(Pgrp *pgrp, int invert, char *devs)
{
	int i, t, w;
	char *p;
	Rune r;
	u64int mask[nelem(pgrp->notallowed)];

	if(invert)
		memset(mask, 0xFF, sizeof mask);
	else
		memset(mask, 0, sizeof mask);

	w = sizeof mask[0] * 8;
	for(p = devs; *p != 0;){
		p += chartorune(&r, p);
		t = devno(r, 1);
		if(t == -1)
			continue;
		if(invert)
			mask[t/w] &= ~(1<<t%w);
		else
			mask[t/w] |= 1<<t%w;
	}

	wlock(&pgrp->ns);
	for(i=0; i < nelem(pgrp->notallowed); i++)
		pgrp->notallowed[i] |= mask[i];
	wunlock(&pgrp->ns);
}
```

Devmask is called from the write handler for /dev/drivers. This bitmask is then consulted any time a name is resolved that begins with '#'. This is done from within the `namec()` function using the following function to check if a particular device `r` is permitted.

```
int
devallowed(Pgrp *pgrp, int r)
{
	int t, w, b;

	t = devno(r, 1);
	if(t == -1)
		return 0;

	w = sizeof(u64int) * 8;
	rlock(&pgrp->ns);
	b = !(pgrp->notallowed[t/w] & 1<<t%w);
	runlock(&pgrp->ns);
	return b;
}
```

We found that once removal is made a core verb of these sharp devices it becomes easy to start to view access to them as capabilities. This is aided by system functionally already neatly organized in to the various devices themselves. For example, one could say a process is capable of accessing the broader internet if it has access to the `devip` device. This access can either be direct via it's path under '#' or through a location in the namespace where this device had already been bound. With these changes, the entire capability list of a process is on display through just its /proc/$pid/ns file. This `ns` file would indicate if a particular device is bound and now also includes the list of devices a process has access to.

In practice, this results in a pattern of binding in a sharp device, making use of them and removing them when no longer needed. A namespace file for a web server could now look like

```
bind #s /srv
# /srv/www created by srvfs www /lib/www
mount /srv/www /lib/
unmount /srv
chdev -r s # chdev &~ s
```

In this example we have created a new root for the process by using exportfs to expose a little piece of the boot namespace. We unmount `devsrv` and remove access to it with `chdev` ensuring there is no way for our process to talk to the real `/srv/boot`. This provides a nice succinct lifetime of access to `devsrv` and makes the removal of these sharp devices as easy as it is to use them in the first place.

Like RFNOMNT, `chdev` does not restrict access to sharp devices that had already been mounted. This allows a process to use a subsection or only one piece of sharp devices as well. One example of this may be to restrict a process to just a single network stack

```
bind '#I1' /net
chdev -r I
```

/srv/clone

With this `chdev` mechanism, the ability for a device to provide isolation of its own became more powerful. Partially illustrated in the previous `devip` example. `Devsrv`, the sharp device providing named pipes, was an ideal target for adding isolation. Devsrv provides a bulletin board of all posted 9p services for a given host. We wanted to provide a mechanism for a process, or family tree of process to share a private `devsrv` between themselves.

The design for this was borrowed from devip, one in which a process opens a `clone` file to read its newly allocated slot number. This new 'board' appears as a sibling

directory to the clone it was spawned from. This new board is itself a fully functioning devsrv with its own clone file, making nesting to full trees of srvs quite easy, and completely transparent. The following illustrates how one could replace their global /srv with a freshly allocated one.

```
</srv/clone {
        s='/srv/'^`{read}
        bind -c $s /srv
        exec p
}
```

Also like devip, once the last reference to the file descriptor returned by opening clone is closed the board is closed and posters to that board receive an EOF. It is important to bake this kind of ownership in to the design, as self referential users of /srv are quite common in current code.

This along with chdev can be used to create a sandbox for /srv quite easily, the process allocates itself a new /srv then removes access to the global root srv. This allows potentially untrusted process to still make use of the interface without needing to worry about their access to the global state. The practice of having new boards appear as subdirectories allows the entire state to easily be seen by inspecting the root of devsrv itself.

Restricting Within a Mount

As shown earlier with the use of srvfs, an intermediate file server can be used to only service a small subsection of a larger namespace. In that example we used this to expose only /lib/www from the host to processes running a web server. This can be limited as the invocation of exportfs can become more complicated if the user wishes to use multiple pieces from completely separate places within the file tree. To address this a utility program protofs was written to easily create convincing mimics of the filesystem it was run from. protofs accepts a proto file, a text file containing a description of file tree, and uses it to provide dummy files mimicking the structure. These dummies can then be used by a process as targets for bind mounts of its current namespace, providing the illusion of trimming all but select pieces. This new root can not be simply bound over the real one, that still allows an unmount to escape back to the real system but rexporting the namespace still works. To illustrate a more involved setup then before.

```
# We want to provide our web server
# with /bin, /lib/www and /lib/git
; cat >>/tmp/proto <<.
bin     d775
lib     d775
        www     d775
        git     d775
; protofs -m /mnt/proto /tmp/prot
; bind /bin /mnt/proto/bin
; bind /lib/www /mnt/proto/lib/www
; bind /lib/git /mnt/proto/lib/git
# A private srv could be used, omitted for brevity
; srvfs webbox /mnt/proto
# Namespace file for using our new mini-root
; cat >>/tmp/ns <<.
mount #s/webbox /root
bind -b /root /
chdev -r s
; auth/newns -n /tmp/ns ls /
bin
lib
;
```

Remarks

Some colleagues have expressed concern in how restricted namespaces are more effective then the more classical "Just fix security holes" approach. We find this approach carries the assumption that the security domain of a program is static. Just as value was found in the ability to pass programs namespace files, we found value in the ability to specify different security domains for different invocations of the same binary.

Future Work

While we think these bring us closer to namespaces as security boundaries, there is still plenty of work and understanding to be done. One particular item of interest is attempting some kind of isolation of `devproc`, possibly in a similar fashion to the `/srv/clone` implementation, but attempts have yet to be made. The exact nature of `namespace` files and how they relate to sandboxing as a whole has yet to be fully worked out. There is clear potential, but it is likely additional abilities may be required. It is somewhat difficult to synthesize a namespace entirely from nothing, which is something we found ourselves reaching for when building alternative roots to run processes within. There is potential for some merger of `proto` and `namespace` files to provide a template of the current namespace to graft on to the next one.

Both `chdev` and `/srv/clone` are merged into 9front and their implementations are freely available as part of the base system.

[Beck18] Bob Beck, ‘‘Pledge, and Unveil, in OpenBSD’’, *BSDCan Slides* Ottawa, July, 2018.

[Price04] Daniel Price, Andrew Tucker, ‘‘Solaris Zones: Operating System Support for Consolidating Commercial Workloads’’, *Proceedings of the 18th Large Installation System Administration Conference* pp. 241–254, Atlanta, November, 2004.

[Biederman06] Eric W. Biederman ‘‘Multiple Instances of the Global Linux Namespaces’’, *Proceedings of the 2006 Linux Symposium Volume One* pp. 102–112, Ottawa, Ontario July, 2006.

Single Level Store Application Management With 9P

Emil Tsalapatis, University of Waterloo
emil.tsalapatis@uwaterloo.ca

Ryan Hancock, University of Waterloo
krhancoc@uwaterloo.ca

Ali José Mashtizadeh, University of Waterloo
ali.mashtizadeh@uwaterloo.ca

ABSTRACT

Single Level Stores (SLSes) are OSes where applications can be written as if the system never crashes. SLSes use high-frequency checkpointing to create an executable image out of all application state. The system uses the image after a crash to restore the applications, which resume oblivious to the interruption. Current SLSes like Aurora use high-frequency checkpointing to keep the image up to date with the application.

Aurora does not propagate updates from images to running applications. Such a mechanism would allow users to scale, migrate, and modify applications by operating on their images. Representing images as file directories provides an intuitive interface and allows using standard system utilities to manage application state. The mechanism would resemble a `/proc` file system that translates writes and file creation/deletion into system-level operations on application state.

We design a 9P server for Aurora to propagate the state of an image back to its application. The server stops the applications when its image is opened as writable, and kills it if any writes occur. The server then restores the application from the updated image once all files are closed.

We combine the applications-as-data abstraction of the SLS with the 9P protocol to implement complex operations as sequences of file operations. These include persisting applications to have them survive across reboots, live migrating them across machines, and transparently autoscaling them. We also create complex OS abstractions using just file operations on the application images. We describe such an example, application fork.

Introduction

Single Level Stores (SLSes) are OSes that abstract the difference between volatile memory and persistent storage from applications. They instead provide persistent byte addressable memory that programs use both for volatile and persistent data. SLSes implement this abstraction using high-frequency checkpointing to continuously persist application state. In case of a crash the system restores all applications, which resume

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

oblivious to the interruption. SLSes like Aurora [Tsal21] achieve checkpointing frequencies upwards of 100 checkpoints per second, keeping image data closely synced with application state.

The correspondence between images and applications opens the possibility of expressing complex system-level operations as operations on images. For example, copying an image is equivalent to cloning an entire application. The new instance is identical to the original down to individual threads, processes, and open files. The two instances do not interfere with each other because Aurora applies copy on write (COW) to all OS resources, down to file system roots.

Aurora does not directly expose image data to userspace, and instead uses an `ioctl` based API that provides a limited number of operations on the data. The API has proven brittle during development and arbitrarily prevents users from directly interacting with images. Adding new functionality requires creating logic for a new call across userspace and the kernel, and uses ad hoc flags to control configuration. For example, Aurora uses a flag for the restore call to switch between eagerly and lazily paging in an image when restoring an application.

A file interface makes it easier to develop new functionality for Aurora and is more intuitive to users. This interface lets us move operation logic from the kernel into separate userspace applications, making development easier. For example, the restore operation now does not require a flag to decide between eager and lazy; for eager restores, a userspace utility just reads the image into memory before restoring. Users can also use the file API to do complex tasks like move an image between machines using existing system utilities.

This paper describes Aurora 9P, a 9P server that allows for the manipulation of running applications through a file interface. Aurora9P is built on top on a Unix (FreeBSD) system and represents running applications as sets of files. Aurora9P presents a self-contained directory for each application, and one file in that directory for each OS resource that the application owns. By OS resource here we mean such state as threads, address spaces, memory regions, file handles, vnodes, and so on. Manipulating the image leads to live modifications in the application.

The server presents applications much like the Unix `/proc` file system does, but adds semantics for writing and by extension copying/deleting the files it presents. The server has more powerful semantics than `/proc` [Kill81] because it attaches semantics to write operations for files, which enables the duplication and modification of live applications. Writing to a file changes the application resource it represents. Aurora9P's semantics are expressive because its files do not just represent a resource, the files *are* the resource. Modifications to the file propagate to the running application, and creating or deleting files results in the creation or destruction of a resource. The original Plan9 paper [Pike95] mentions the difficulty of attaching semantics to file operations of a `/proc` -like server. Aurora9P uses the equivalence between checkpoint images and running applications in SLSes to implement these file operations using the semantics of the underlying OS.

Users can now treat running applications as data and perform complex system administration tasks using standard system utilities. For example, the server moves applications between machines by moving the application directory from a local to a remote Aurora9P server. Autoscaling is possible by copying an application directory to create a new running instance of the application. Developers can interact with applications at the system level by modifying the image. For example, creating a new file in the image

creates and attaches a new resource, e.g. mapping or open file descriptor. Even creating a new application from scratch is possible using a shell script which creates files for process and thread information, mappings, and files for input and output.

In this paper we first explain the motivation behind single level stores and their current capabilities. We describe Aurora's API and analyze its drawbacks, then introduce Aurora9P and analyze its semantics. We then derive APIs for more complex operations including autoscaling and migration, and describe a multiprocess application fork primitive. We close out with a discussion on how file interfaces combine with system introspection to simplify the system call interface.

The Aurora SLS

The Aurora SLS simplifies application development by presenting a persistent memory abstraction to userspace. The OS guarantees that the application can be restored after a crash without losing data and without requiring developer effort. By contrast, conventional OSes require applications to store persistent data into files and use `fsync` calls to send them to the disk. File-based persistence is difficult to implement correctly, and must be done by application developers. Persisting complex data structures like database tables in the wrong order can cause corruption.

Aurora implements a persistent memory abstraction using high-frequency checkpointing on the entire application. Checkpointing happens every few milliseconds, 10 by default, and updates the persistent application image to reflect changes in the application state. The OS stops the application, gathers its OS state, and lets the application continue while it flushes out the image. Aurora uses differential checkpointing, and only sends out to the disk data that was updated since the last checkpoint. It also applies copy on write (COW) to application data to parallelize application execution with flushing. Checkpointing in total only causes a small performance penalty because it stops the application for mere microseconds.

Aurora applies external synchrony based on external consistency [Nigh08] to prevent other hosts from observing unflushed application state. The system buffers messages sent to other hosts until the checkpoint in which the messages were sent persists. This technique prevents inconsistencies caused by external machines observing application state that can be lost during a crash. For example, a database in Aurora that receives an INSERT command must confirm it has persisted the value to the client. Without external synchrony the database may crash after sending the acknowledgement, losing the value. Aurora prevents this by flushing the acknowledgement message only after the checkpoint is safely on the disk.

The restore process rebuilds the application as it was at checkpoint time, including kernel state like file descriptors. No state is lost, as Aurora restores even thread state and pending events. The application does not require any error handling for the crash, and can immediately continue executing. The only application state that is not restored is connected sockets, which are replaced with invalid closed sockets. The application cleans up these invalid sockets using the same error handling it would use for a connection closed by the remote machine.

Aurora demonstrates that SLS capabilities are general in their use and can be retrofitted to existing OSes with reasonable effort. SLS capabilities are introspection mechanisms at their core, and are not inherently tied to persistence. These mechanisms can be added to an OS without much effort: Aurora uses FreeBSD as its base and adds about 10KSLOC for the checkpoint and restore operations. The amount of code necessary for

checkpoint/restore scales with the complexity of the base OS, as it must handle all OS mechanisms. Regarding performance, the cost of checkpointing is feasible even for under-powered machines because it scales with application size.

Managing Aurora with 9P

We have tried a variety of interfaces for Aurora's persistent applications, but all have proven difficult to work with. Until recently the system used a virtual device that provided different `ioctl` commands for application management. Using a device instead of hardcoding new system calls avoided further bloating the already large system call API. This version made it difficult to adjust the SLS interface because it required significant boilerplate for each new call. Commands used argument flags to pass information between the user and the SLS, but these flags quickly became complex and composed in unexpected ways. Adjusting the semantics of the commands was complicated, because we needed to change multiple components across userspace and the kernel.

We design a new interface for managing application images based on the 9P protocol. The protocol models Aurora operations as file operations, and provides a more intuitive interface than the old `ioctl` based API. For example, changing the checkpointing period previously required the user to use a special `slsctl` utility and pass the new period as an argument together with a flag. The same operation is now possible by writing out the new period into a special `period` file using standard shell utilities like the `echo` command. In fact, most of our userspace utilities are now either unnecessary under the new API.

File semantics are a natural fit for our API, much more so than an `ioctl` based API. A narrow API based on function calls is normally simpler than a text-oriented protocol, but that's because it does not need parsing. Aurora has to do parsing in the kernel when restoring anyway, because it needs to ensure the checkpoint it is restoring is well-formed. Using a file server thus adds minimal complexity while making it easy to modify to the interface. The apparent extra complexity of a 9P server compared to a simple device is thus misleading.

The 9P interface also holds great potential for simplifying the management of entire networks made of Aurora machines. We compose the different SLS servers from different machines to provide a consistent view of our cluster from any host. Remote applications are indistinguishable from local ones. The namespacing feature enables straightforward implementations for scaling and migration. The lack of 9P's client-side caching is a good fit for the application servers, because application files are by nature constantly changing.

The Server Directory Structure

Below we describe the 9P server's API and how it represents common SLS operations. Aurora9P presents all applications in the SLS as a file tree. The root of the server, `/aurora9p`, holds one directory per application. For example, we find the checkpoints for application `app` in `/aurora9p/app`. The application directory has a set of directories with each directory holding a checkpoint from some point in time. Aurora9P retains the external synchrony guarantees of the SLS by only exposing checkpoints that have been sent to persistent storage.

Aurora uses hard links to manage application checkpoints. Each new differential checkpoint is represented as a new subdirectory in the main application directory. New directories are named after their creation timestamp to sort them in chronological order.

Aurora automatically creates a hard link to the most recent checkpoint, which it names `latest` for easy retrieval. Aurora makes room for new checkpoints by deleting older ones, starting from the oldest available one. Users can prevent a checkpoint from being deleted by renaming or creating a hard link to its directory under a name of their choice.

Aurora9P exposes operations like checkpoint and restore using file semantics. Users configure the application's checkpointing parameters by writing to special files in the base directory, e.g. `/aurora9p/app/checkpoint_period` to control the checkpointing period. Restoring is not an explicit operation anymore but is now done by writing `YES` to the control file `/aurora9p/app/running` that denotes whether an application is currently active. The file interface thus accurately captures the SLS model where images are inactive applications and not just data like core dumps.

Users manage checkpoints using file operations on the checkpoint directories. Each directory represents the checkpoint as a collection of files, one for each OS resource. The directory is self-contained and contains all necessary information to restore the application. Reading these files returns a the metadata for the resource in a comma separated value format. Each resource has a unique identifier that other resources use to link to it. For example, a file that represents a process has entries for its PID, its parent PID, and one unique identifier for each of its threads.

Reading from the application's checkpoint files does not require stopping the application because the reader just uses the latest checkpoint. Reading the checkpoint does not return the data of the currently running application like `/proc` does. For this reason Aurora9P's implementation is simpler than that of `/proc` as it does not traverse live kernel data structures when reading the data.

The read file methods internally use the same deserialization logic as the restore operation, making them easy to maintain. The `ioctl` interface for Aurora required a separate userspace tool for reading and writing images, as it would have been very difficult maintaining separate `ioctl` calls for each file type. The tool was eventually abandoned because keeping it in sync with the kernel code was too high-maintainance for the value it provided. As a result it was often out-of-date, improperly parsing images and generally being a liability during debugging. The current interface is low-maintainance and has proven useful both for debugging and development.

Aurora9P's write routines allow the modification of both running and inactive applications, making it useful for tasks like debugging. Users can modify the running application by modifying the `latest` checkpoint file. Aurora responds to writes to the file by destroying the current instance of the application and restoring a new one from the updated checkpoint. All unpersisted changes are safely discarded in the process because external synchrony prevents them from being visible outside the application itself. Updating live applications is useful for debugging because it allows modifying its state to observe how its behavior changes.

Aurora9P allows a single open call for a checkpoint directory to implement atomicity for modifications that span multiple files. This also prevents races between writes and the SLS checkpoint creation process. The SLS itself adds new checkpoints using the 9P protocol, so it is enough to open `/aurora9p/app/latest` for writing to prevent new checkpoints from being persisted. The SLS stops the application from resuming until it can create the new checkpoint. In the above scenario uncommited data is ultimately discarded because of writes to the checkpoint file.

Application Fork

We next describe how we implement application fork to duplicate entire applications. We implement application fork as a file copy operation in Aurora9P from an existing application directory to a new one. We can create applications from earlier checkpoints instead of the most recent one, but for this example we are using the checkpoint `/aurora9p/appserver/latest/` from a server application called `appserver`.

We implement the operation exclusively through file operations. We create a new application directory called `/aurora9p/appclone` and copy `/aurora9p/appserver/latest` to `/aurora9p/appclone/latest`. The SLS internally uses COW to implement the copying of data regions, so the operation does not trigger significant data copies. The main overheads of the copy operation are creating and writing the OS resource files of the new checkpoint. Each such write triggers the creation of the corresponding resource in the kernel. For example, creating a new process file creates a new process, including threads. The end result is a new application named `appclone` identical to `appserver`. All process state, thread state, open file descriptors, and memory are the same.

Application fork does not modify the checkpoint's files with the exception of internet network sockets. The SLS has local namespaces for each kernel resource, including PIDs and file names. The only exception is internet sockets whose addresses are part of a common network-wide namespace. These sockets require their (IP, port) tuples to be unique, so they cannot be cloned outright. Application fork handles UDP sockets and TCP listening sockets by assigning new user-specified values to their (IP, ports) tuples. Connected TCP sockets are replaced with closed sockets, and the cloned application is expected to clean them up after resuming. In our example we assume the application is a server in the middle of servicing remote hosts, with multiple connected sockets per host. The restored application runs with these sockets closed, and runs as if the remote hosts suddenly lost connectivity.

We use application fork to autoscale applications. Creating a new instance this way avoids setup like runtime initialization that the new instance would have to make if starting from scratch. Application fork is particularly efficient for creating short-lived workloads like serverless functions. For such jobs we can keep a checkpoint taken right before when the workload receives its arguments. We create new instances using application fork, then connecting to the workload and sending it its input. We can even use a checkpoint taken right *after* the application receives its arguments, and overwrite them in the image with the new input.

Aurora9P makes it easy to autoscale or migrate workloads between different machines. Mounting a remote Aurora9P server locally and moving or copying over a checkpoint creates a new instance on the remote machine. The two machines must run the same architecture and OS version to ensure ABI compatibility. The SLS' per-application namespaces for all kernel resources prevent naming conflicts between local and remote workloads.

Discussion

The 9P interface has proven remarkably flexible for interacting with complex kernel APIs from userspace. The alternatives, including adding to the system call interface or making `ioctl` calls to a device, proved brittle and required us to reimplement functionality. The experiences we had with the `ioctl` interface and the image analysis tool demonstrate the problems of system-call like binary interfaces when used for complex

operations.

The file interface has proven robust to malformed images because the SLS internally uses existing kernel APIs to create new OS resources. These APIs are well-tested and have strict checks on the arguments they can take. For example, the SLS creates a pipe with data already in it by creating the pipe using `pipe` then writing the data to it using `write`. The Aurora9P code need only split the files in the checkpoint into arguments, because the SLS verifies that these arguments represent valid OS resources.

The 9P API and Aurora's applicaiton introspection combine to minimize the system call ABI of existing systems. Most system calls' purpose is to provide a way for an application to modify its state in a controlled fashion. Modifications include adding a mapping or file, reading or writing data to a file, or setting configuration flags. Our 9P server instead lets us describe these operations as file reads/writes on resource pseudofiles. Even very complex system calls like `rfork` can be described as file operations. Using an introspection-based file API for most privileged operations could potentially reduce the system call bloat present in modern OSes.

Single level store capabilities do not require a radical redesign of the OS but can instead be retrofitted into existing kernels. The SLS/9P interface of Aurora9P is applicable to most OSes even if our current implementation uses FreeBSD. The only FreeBSD specific aspect of our current implementation is the use of Mach-style virtual memory objects [Acce86] to implement differential checkpointing. Other systems would have to either implement differential checkpointing, or use a lower checkpointing frequency. SLS capabilities for smaller kernels additionally require less developer effort as they have to account for fewer OS primitives.

References

[Acce86] Accetta, Mike, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A new Kernel Foundation for UNIX Development.", 1986

[Kill84] T.J. Killian, ''Processes as Files'', USENIX Summer 1984 Conf. Proc., June 1984, Salt Lake City, UT.

[Nigh08] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, ''Rethink the Sync'', ACM Transactions on Computing Systems, New York, NY, 2008

[Pike95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom ''Plan 9 from Bell Labs'', Berkeley, CA: University of California Press

[Tsal21] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, Ali José Mashtizadeh ''The Aurora Operating System: Revisiting the Single Level Store'', Proceedings of the Workshop on Hot Topics in Operating Systems, Ann Arbor, MI, 2021

GEFS, A Good Enough File System

Ori Bernstein
ori@eigenstate.org

ABSTRACT

GEFS is a new file system built for Plan 9. It aims to be a crash-safe, corruption-detecting, simple, and fast snapshotting file system, in that order. GEFS achieves these goals by building a traditional 9p file system interface on top of a forest of copy-on-write Bϵ trees. It doesn't try to be optimal on all axes, but good enough for daily use.

1. The Current Situation

Currently, Plan 9 has several general purpose disk file systems available. All of them share a common set of problems. On power loss, the file systems tend to get corrupted, requiring manual recovery steps. Silent disk corruption is not caught by the file system, and reads will silently return incorrect data. They tend to require a large, unshrinkable disk for archival dumps, and behave poorly when the disk fills. Additionally, all of them do O(n) scans to look up files in directories when walking to a file. This causes poor performance in large directories.

CWFS, the default file system on 9front, has proven to be performant and reliable, but is not crash safe. While the root file system can be recovered from the dump, this is inconvenient and can lead to a large amount of lost data. It has no way to reclaim space from the dump. In addition, due to its age, it has a lot of historical baggage and complexity.

HJFS, a new experimental system in 9front, is extremely simple, with fewer lines of code than any of the other on-disk storage options. It has dumps, but does not separate dump storage from cache storage, allowing full use of small disks. However, it is extremely slow, not crash safe, and lacks consistency check and recovery mechanisms.

Finally, fossil, the default file system on 9legacy, is large and complicated. It uses soft-updates for crash safety[7], an approach that has worked poorly in practice for the BSD filesystems[8]. Fossil also has a history (and present) of deadlocks and crashes due its inherent complexity. There are regular reports of deadlocks and crashes when using tools such as clone[9]. While the bugs can be fixed as they're found, simplicity requires a rethink of the on disk data structures. And even after adding all this complexity, the fossil+venti system provides no way to recover space when the disk fills.

2. How GEFS Solves it

GEFS aims to solve the problems with these file systems. The data and metadata is copied on write, with atomic commits. If the file server crashes before the superblocks are updated, then the next mount will see the last commit that was synced to disk. Some data may be lost, by default up to 5 seconds worth, but no corruption will occur. Furthermore, because of the use of an indexed data structures, directories do not suffer from O(n) lookups, solving a long standing performance issue with large directories.

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

While snapshots are useful to keep data consistent, disks often fail over time. In order to detect corruption and allow space reclamation, block pointers are triples of 64-bit values: The block address, the block hash, and the generation that they were born in. If corrupted data is returned by the underlying storage medium, this is detected via the block hashes. The corruption is reported, and the damaged data may then be recovered from backups, RAID restoration, or some other means. Eventually, the goal is to make GEFS self-healing.

Archival dumps are replaced with snapshots. Snapshots may be deleted at any time, allowing data within a snapshot to be reclaimed for reuse. To enable this, in addition to the address and hash, each block pointer contains a birth generation. Blocks are reclaimed using a deadlist algorithm inspired by ZFS.

Finally, the entire file system is based around a relatively novel data structure. This data structure is known as a Bϵ tree [1]. It's a write optimized variant of a B+ tree, which plays particularly nicely with copy on write semantics. This allows GEFS to greatly reduce write amplification seen with traditional copy on write B-trees.

And as a bonus, it solves these problems with less complexity. By selecting a suitable data structure, a large amount of complexity elsewhere in the file system falls away. The complexity of the core data structure pays dividends. Being able to atomically update multiple attributes in the Bϵ tree, making the core data structure safely traversable without locks, and having a simple, unified set of operations makes everything else simpler. As a result, the total source size of GEFS is currently 8737 lines of code, as compared to CWFS at 15634, and the fossil/venti system at 27762 lines of code (20429 for fossil, with an additional 7333 lines for Venti).

3. Bε Trees: A Short Summary

The core data structure used in GEFS is a Bϵ tree. A Bϵ tree is a modification of a B+ tree, which optimizes writes by adding a write buffer to the pivot nodes. Like B-trees, Bϵ trees consist of leaf nodes, which contain keys and values, and pivot nodes. Like B-trees, the pivot nodes contain pointers to their children, which are either pivot nodes or leaf nodes. Unlike B-trees, the pivot nodes also contain a write buffer.

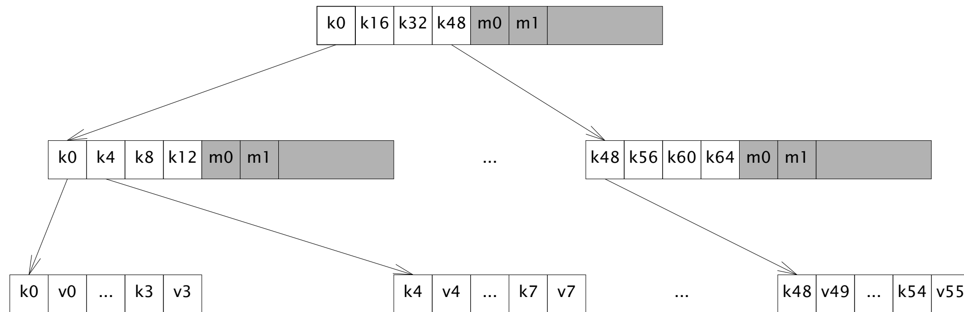
The Bϵ tree implements a simple key-value API, with point queries and range scans. It diverges form a traditional B-tree key value store by the addition of an upsert operation. Upsert operations are operations that insert a modification message into the tree. These modifications are addressed to a key.

To insert to the tree, the root node is copied, and the new message is inserted into its write buffer. When the write buffer is full, it is inspected, and the number of messages directed to each child is counted up. The child with the largest number of pending writes is picked as the victim, and the root's write buffer is flushed towards it. This proceeds recursively down the tree, until either an intermediate node has sufficient space in its write buffer, or the messages reach a leaf node, at which point the value in the leaf is updated.

In order to query a value, the tree is walked as normal, however the path to the leaf node is recorded. When a value is found, the write buffers along the path to the root are inspected, and any messages that have not yet reached the leaves are applied to the final value read back.

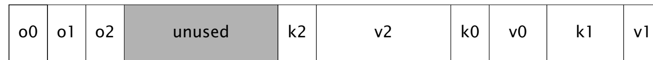
Because mutations to the leaf nodes are messages that describe a mutation, updates to data may be performed without inspecting the data at all. For example, when writing to a file, the modification time and QID version of the file may be incremented without inspecting the current QID; a 'new version' message may be upserted instead. This allows skipping read-modify-write cycles that access distant regions of the tree, in favor of a simple insertion into the root nodes write buffer. Additionally, because all upserts go into the root node, a number of operations may be upserted in a

single update. As long as we ensure that there is sufficient space in the root node's write buffer, the batch insert is atomic. Inserts and deletions are upserts, but so are mutations to existing data.



In GEFS, for the sake of simplicity all blocks are the same size. Unfortunately, this implies that the Bϵ tree blocks are smaller than optimal, and the disk blocks are larger than optimal.

Within a single block, the pivot keys are stored as offsets to variable width data. The data itself is unsorted, but the offsets pointing to it are sorted. This allows O(1) access to the keys and values, while allowing variable sizes.



In order to allow for efficient copy on write operation, the Bϵ tree in GEFS relaxes several of the balance properties of B-trees [5]. It allows for a smaller amount of fill than would normally be required, and merges nodes with their siblings opportunistically. In order to prevent sideways pointers between sibling nodes that would need copy on write updates, the fill levels are stored in the parent blocks, and updated when updating the child pointers.

4. Mapping Files to Bε Operations

With a description of the core data structure completed, we now need to describe how a file system is mapped on to Bϵ trees.

A GEFS file system consists of a snapshot tree, which points to a number of file system trees. The snapshot tree exists to track snapshots, and will be covered later. Each snapshot points to a single GEFS metadata tree, which contains all file system state for a single version of the file system. GEFS is somewhat unique in that all file system data is recorded within a single flat key value store. There are no directory structures, no indirect blocks, and no other traditional structures. Instead, GEFS has the following key-value pairs:

`Kdat(qid, offset) → (ptr)`

Data keys store pointers to data blocks. The key is the file qid, concatenated to the block-aligned file offset. The value is the pointer to the data block that is being looked up.

`Kent(pqid, name) → (stat)`

Entry keys contain file metadata. The key is the qid of the containing directory, concatenated to the name of the file within the directory. The value is a stat struct, containing the file metadata, including the qid of the directory entry.

`Kup(qid) → Kent(pqid, name)`

Up keys are maintained so that '..' walks can find their parent directory. The key is the qid of the directory. The value is the key for the parent directory.

Walking a path is done by starting at the root, which has a parent qid of ~0, and a name of "/". The QID of the root is looked up, and the key for the next step on the walk is constructed by concatenating the walk element with the root qid. This produces the key for the next walk element, which is then looked up, and the next key for the walk path is constructed. This continues until the full walk has completed. If one of the path elements is '..' instead of a name, then the super key is inspected instead to find the parent link of the directory.

If we had a file hierarchy containing the paths 'foo/bar', 'foo/baz/meh', 'quux', 'blorp', with 'blorp' containing the text 'hello world', this file system may be represented with the following set of keys and values:

```
Kdat(qid=3, off=0) → Bptr(off=0x712000, hash=04a73, gen=712)
Kent(pqid=1, name='blorp') → Dir(qid=3, mode=0644, ...)
Kent(pqid=1, name='foo') → Dir(qid=2, mode=DMDIR|0755, ...)
Kent(pqid=1, name='quux') → Dir(qid=4, mode=0644, ...)
Kent(pqid=2, name='bar') → Dir(qid=6, mode=DMDIR|0755, ...)
Kent(pqid=2, name='baz') → Dir(qid=5, mode=DMDIR|0755, ...)
Kent(pqid=5, name='meh') → Dir(qid=5, mode=0600, ...)
Kent(pqid=-1, name='') → Dir(qid=1, mode=DMDIR|0755, ...)
Kup(qid=2) → Kent(pqid=-1, name='')
Kup(qid=5) → Kent(pqid=2, name='foo')
```

Note that all of the keys for a single directory are grouped because they sort together, and that if we were to read a file sequentially, all of the data keys for the file would be similarly grouped.

If we were to walk `foo/bar` then we would begin by constructing the key `Kent(-1, '')` to get the root directory entry. The directory entry contains the qid. For this example, let's assume that the root qid is 123. The key for `foo` is then constructed by concatenating the root qid to the first walk name, giving the key `Kent(123, foo)` This is then looked up, giving the directory entry for `foo`. If the directory entry contains the qid 234, then the key `Kent(234, bar)` is then constructed and looked up. The walk is then done.

Because a Bϵ tree is a sorted data structure, range scans are efficient. As a result, listing a directory is done by doing a range scan of all keys that start with the qid of the directory entry.

Reading from a file proceeds in a similar way, though with less iteration: When writing to a file, the qid is known, so the block key is created by concatenating the file qid with the read offset. This is then looked up, and the address of the block containing the data is found. The block is then read, and the data is returned.

Writing proceeds in a similar manner to reading, and in the general case begins by looking up the existing block containing the data so that it can be modified and updated. If a write happens to fully cover a data block, then a blind upsert of the data is done instead. Atomically along with the upsert of the new data, a blind write of the version number incremnt, mtime, and muid is performed.

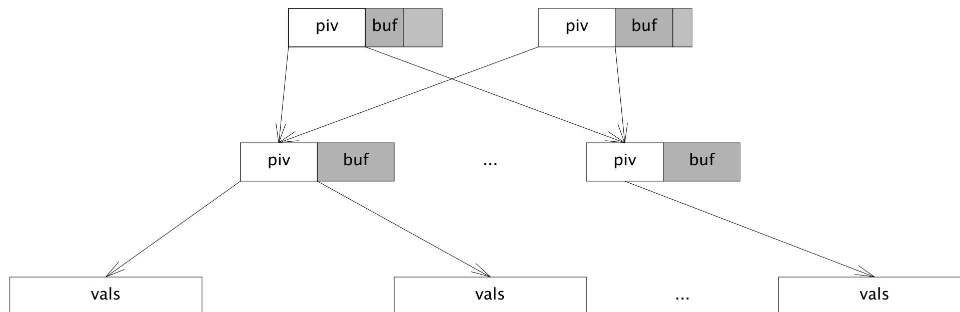
Stats and wstat operations both construct and look up the keys for the directory entries, either upserting modifications or reading the data back directly.

5. Snapshots

GEFS supports snapshots. Each snapshot is referred to by a unique integer id, and is fully immutable once it is taken. For human use, a snapshot may be labeled. The labels may move to new snapshots, either automatically if they point to a tip of a snapshot, or via user intervention. For the sake of space reclamation, snapshots are reference counted. Each snapshot takes a reference to the snapshot it descends from. Each label also takes a reference to the snapshot that it descends from. When a snapshot's

only reference is its descendant, then it is deleted, and any space it uses exclusively is reclaimed. The only structure GEFS keeps on disk are snapshots, which are taken every 5 seconds. This means that in the case of sudden termination, GEFS may lose up to 5 seconds of data, but the data on disk will always be consistent.

If there was no space reclamation in gefs, then snapshots would be trivial. The tree is copy on write. Therefore, as long as blocks are never reclaimed, it would be sufficient to save the current root of the tree once all blocks in it were synced to disk. However, because snapshots are taken every 5 seconds, disk space would get used uncomfortably quickly. As a result, space needs to be reclaimed. An advantage of Bε trees here is that often, only the root block will be copied, and updates will be inserted into its write buffer.



There are a number of options for space reclamation. Some that were considered when implementing GEFS included garbage collection, in the style of HAMMER [3], or optimized reference counting in the style of BTRFS [4], but both of these options have significant downsides. Garbage collection requires that the entire disk get scanned to find unreferenced blocks. This means that there are scheduled performance degradations, and in the limit of throughput, the bandwidth spent scanning must approach the bandwidth spent on metadata updates, as each block must be scanned and then reclaimed. Reference counting implies a large number of scattered writes to maintain the reference counts of blocks.

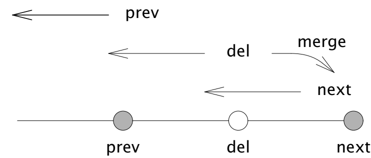
As a result, the algorithm for space reclamation is lifted directly from ZFS [6]. It is based on the idea of using deadlists to track blocks that became free within a snapshot. If snapshots are immutable, then a block may not be freed as long as a snapshot exists. This implies that block lifetimes are contiguous. A block may not exist in a snapshot and be available for reallocation. Thus, when freeing a block, there are 2 cases: Either a block was born within the pending snapshot, and died within it, or it was born in a previous snapshot and was killed by the pending snapshot.

To build intuition, let's start by imagining the crudest possible implementation of snapshot space reclamation. Assuming that block pointers contain their birth generation, we can walk the entire tree. When a block's birth time is <= the previous snapshot, it is referred to by an older snapshot. We may not reclaim it. If the subsequent snapshot refers to this block, then it was born in this snapshot but is still in use. We may not reclaim it. Otherwise, the block is free, and we can reclaim it.

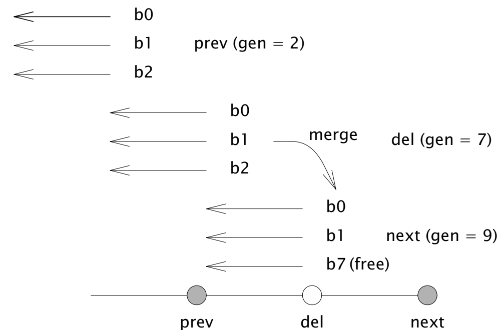
Obviously, this is slow: It involves full tree walks of multiple snapshots. It may walk large numbers of blocks that are not freed.

So, in order to do better, we can keep track of blocks that we want to delete from this snapshot as we delete them, instead of trying to reconstruct the list when we delete the snapshot. When we attempt to delete a block, there are two cases: First, block's birth time may be newer than the previous snapshot, in which case it may be freed immediately. And second, the block may have been born in the previous snapshot or earlier, in which case we need to put it on the current snapshot's deadlist. When the current snapshot is deleted, the current snapshot's deadlist is merged with the next

snapshot's deadlist. All blocks on the deadlist that were born after the previous snapshot are freed.



There's one further optimization we can do on top of this to make deletions extremely fast. The deadlists may be sharded by birth generation. When a snapshot is deleted, all deadlists within the snapshot are appended to the descendant snapshot, and any deadlists with a birth time after the deleted snapshot in the descendant may be reclaimed. With this approach, the only lists that need to be scanned are the ones consisting wholly of blocks that must be freed.



The disadvantage of this approach is that appending to the deadlists may need more random writes. This is because in the worst case, blocks deleted may be scattered across a large number of generations. It seems likely that in practice, most bulk deletions will touch files that were written in a small number of generations, and not scattered across the whole history of the disk.

The information about the snapshots, deadlists, and labels are stored in a separate snapshot tree. The snapshot tree, of course, can never be snapshotted itself. However, it's also a copy on write Bϵ tree where blocks are reclaimed immediately. It's kept consistent by syncing both the root of the snapshot tree and the freelists at the same time. If any blocks in the snapshot tree are freed, this freeing is only reflected after the snapshot tree is synced to disk fully.

The key-value pairs in the snapshot tree are stored as follows

`Ksnap(id) → (tree)`

Snapshot keys take a unique numeric snapshot id. The value contains the tree root. This includes the block pointer for the tree, the snapshot generation of the tree, the previous snapshot of the tree, its reference count, and its height.

`Klabel(name) → (snapid)`

Label keys contain a human-readable string identifying a snapshot. The value is a snapshot id. Labels regularly move between snapshots. When mounting a mutable snapshot, the label is updated to point at the latest snapshot every time the tree is synced to disk.

`Kslink(snap, next) → ()`

A snap link key contains a snapshot id, and the id of one of its successors. Ideally, the successor would be a value, but our Bϵ tree requires unique keys, so we hack around it by putting both values into the key. When we have exactly one next link, and no labels that point at this snapshot, we merge with our successor.

`Kdead(snap, gen) → (headptr, tailptr)`

A dead key contains a pair of snapshot id and deadlist generation. The value contains a head and tail pointer for a deadlist. These are used to quickly look up and merge deadlists, as described earlier in this paper.

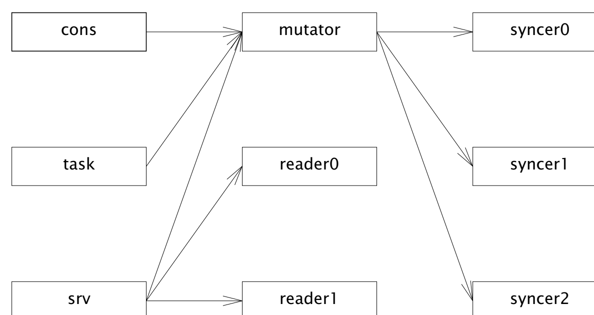
6. Block Allocation

In GEFS, blocks are allocated from arenas. Within an arena, allocations are stored in a linked list of blocks, which is read at file system initialization. The blocks contain a journal of free or allocate operations, which free or allocate regions of disk. When the file system starts, it replays this log of allocations and frees, storing the available regions of blocks in an in-memory AVL tree. As the file system runs, it appends to the free space log, and occasionally compresses this log, collapsing adjacent free or used blocks into larger regions.

Because of the copy on write structure, it's fairly common for metadata blocks to get allocated and deallocated rapidly. Drives (even solid state drives) care a lot about sequential access, so it's beneficial to make a best effort attempt at keeping data sequential. As a result, GEFS selects the arena to allocate from via round robin, offsetting by the type of block. If the round robin counter is 10, and we have 7 arenas, then data blocks (type 0) are allocated from arena 3 ((10+0)%7), pivot blocks (type 1) are allocated from arena 4 ((10+1)%7), and leaf blocks (type 2) are allocated from arena 5 ((10+2)%7). The round robin counter is incremented after every few thousand block writes, in order to balance writes across arenas. Since all arenas are the same, if an arena is full, we simply advance to the next arena.

7. Process Structure

GEFS is implemented in a multiprocess manner. There is one protocol proc per posted service or listener, which dispatches 9p messages to the appropriate worker. Read-only messages get dispatched to one of many reader procs. Write messages get dispatched to the mutator proc, which modifies the in-memory representation of the file system. The mutator proc sends dirty blocks to the syncer procs. There is also a task proc which messages the mutator proc to do periodic maintenance such as syncing. The console proc also sends messages to the mutator proc to update snapshots and do other file system maintenance tasks.



Because the file system is copy on write, as long as the blocks aren't reclaimed while a reader is accessing the tree, writes need not block reads. However, if a block is freed within the same snapshot, it's possible that a reader might observe a block with an inconsistent state. This is handled by using epoch based reclamation to free blocks.

When a proc starts to operate on the tree, it enters an epoch. This is done by atomically taking the current global epoch, and setting the proc's local epoch to that, with an additional bit set to indicate that the proc is active:

```
epoch[pid] = atomic_load(globalepoch) | Active
```

As the mutator frees blocks, instead of immediately making them reusable, it puts the blocks on the limbo list for its generation:

```
limbo[gen] = append(limbo[gen], b)
```

When the proc finishes operating on the tree, it leaves the epoch by clearing the active bit. When the mutator leaves the current epoch, it also attempts to advance the global epoch. This is done by looping over all worker epochs, and checking if any of them are active in an old epoch. If the old epoch is empty, then it's safe to advance the current epoch and clear the old epoch's limbo list.

```
ge = atomic_load(globalepoch);
for(w in workers){
        e = atomic_load(epoch[w]);
        if((e & Active) && e != (ge | Active))
                return;
}
globalepoch = globalepoch+1
freeblks(limbo[globalepoch - 2])
```

This epoch based approach allows GEFS to avoid contention between writes and reads. A writer may freely mutate the tree as multiple readers traverse it, with no locking between the processes, beyond what is required for the 9p implementation. There is still contention on the FID table, the block cache, and a number of other in-memory data structures.

The block cache is currently a simple LRU cache with a set of preallocated blocks.

8. Future Work

Currently, GEFS is buggy, and the disk format is still subject to change. In its current state, it would be a bad idea to trust your data to it. Testing and debugging is underway, including simulating disk failures for every block written. In addition, disk inspection and repair tools would need to be written. Write performance is also acceptable, but not as good as I would like it to be. We top out at several hundred megabytes per second. A large part of this is that for simplicity, every upsert to the tree copies the root block. A small sorted write buffer in an AVL tree that gets flushed when the root block would reach disk would greatly improve write performance.

On top of this, I have been convinced that fs(3) is not the right choice for RAID. Therefore, I would like to implement RAID directly in GEFS. When implementing RAID5 naïvely, there is a window of inconsistency where a block has been replicated to only some devices, but not all of them. This is known as the "write hole". Implementing mirroring natively would allow the file system to mirror the blocks fully before they appear in the data structures. Having native RAID would also allow automatic recovery based on block pointer hashes. This is not possible with fs(3), because if one copy of a block is corrupt, fs(3) would not know which one had the incorrect data.

Furthermore, growing the file system would be extremely useful for taking flashed disk images and growing them to the full size of the underlying storage. The way that arenas work allows for this to happen fairly easily, but right now there's no way to change the list of arenas. Additionally, the naive round robin allocation across arenas works doesn't allow for balancing.

There are a number of disk-format changing optimizations that should be done. For example, small writes should be done via blind writes to the tree. This would allow small writes to be blindingly fast, and give us the ability to keep small files directly in the values of the tree, without allocating a disk block for them.

Deletion of files could also be done via a range deletion. Currently, each block deleted requires an upsert to the tree, which makes file deletion O(n) with a relatively high cost. Moving to a range deletion makes deletion blindingly fast, at the cost of a large amount of complexity: The deletion message would need to split as it gets flushed down the tree. It's an open question whether the benefit is worth the complexity.

Finally, it would be good to find a method of supporting narrow snapshots, where only a range of the file hierarchy is retained.

9. References

[1] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan,

''An Introduction to Bϵ Trees and Write-Optimization,'' *;login:*,
October 2015, Vol. 40, No. 5" ,

[2] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter, ''BetrFS: A Right-Optimized Write-Optimized File System,'' *Proceedings of the 13th USENIX Conference on File and Storage Technologies,* 2015

[3] Matthew Dillon, "The HAMMER Filesystem," June 2008.

[4] Ohad Rodeh, Josef Bacik, Chris Mason, "BTRFS: The Linux B-Tree Filesystem" *ACM Transactions on Storage, Volume 9, Issue 3, Article No 9, pp 1-32,* August 2013

[5] Ohad Rodeh, "B-trees, Shadowing, and Clones",

H-0245(H0611-006) November 12, 2006

[6] Matt Ahrens, '' How ZFS Snapshots Really Work,'' *BSDCan,* 2019

[7] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. ''Soft Updates: A Solution to the Metadata Update Problem in File Systems,'' *ACM Transactions on Computer Systems*, Vol 18., No. 2, May 2000, pp. 127-153.

[8] Valerie Aurora, ''Soft updates, hard problems'' *Linux Weekly News*, July 1, 2009, https://lwn.net/Articles/339337/

[9] kvik, *Clone*, https://shithub.us/kvik/clone/HEAD/info.html

MIPS Rides Again

Andrew D. Gibson
adventuresin9@gmail.com

1. Background

MIPS has been with Plan 9 since the beginning. Multi-processor MIPS hardware was used early on for the first CPU servers[1], which provided a proving ground for both MIPS compilers and for software that could make use of multiple CPUs. Over the years the MIPS intellectual property has changed hands a few times, and in the process went from powerful workstation and server processors for Silicon Graphics, to simple power efficient cores used by a variety of chip manufacturers for home WiFi routers.

The work detailed below was done using 9 Front. This paper will distinguish "MIPS" for the general name of the type of processor, "mips" for the name of big-endian MIPS related software, and "spim" for the little-endian software.

1.1. Motivation

I had some ideas for using small single board computers as CPU servers. But rather than the traditional role of a CPU server providing computing power, it was to do fairly minimal work in formatting and transmitting sensor data, and doing simple automation tasks. In short, I wanted to run a Plan 9 system on "Internet of Things" like devices. In this case, I am using 9 Front.

9 Front provides a lot of solutions to typical problems found in IoT devices.

•Authentication – to prevent others from issuing commands or reading private information from the device.

•TLS – to provide encryption to keep authentication and data secure in transit.

•Diskless booting – as these devices have minimal storage, it helps that Plan 9 has had network booting tools from the very beginning.

•"Cloud" storage – which can be a remote server, but just as easily on a local file server.

My initial work had been done on full size Raspberry Pis, with the intention to eventually use the Pi Zero as there was already a 9Front kernel for those. However, the global chip shortage happened, and the Pi Zero became difficult to find. Searching for alternatives, I started to include MIPS based development platforms knowing that Plan 9 had run well on MIPS systems in the past.

2. The Platform

I found the Onion Omega 2 Plus[2] for a reasonable price and in stock. It is marketed specifically as an IoT development kit, and uses the MediaTek MT7688, which uses the MIPS24KEc core and includes 128MB of RAM. The MT7688 is a fairly old chip, originally developed as to run WiFi routers, and being built around the idea of being a network attached appliance, it had everything I needed for my purposes. Something to note

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

right away about the MIPS24KEc core is that it lacks a floating point unit, but adds a digital signal processor. So that means floating point math will have to be emulated.

I later found several similar system-on-module and development kit router boards using the same MT7688, and also purchased a Hi-Link HLK7688A[3] router board. Unlike other off the shelf home WiFi routers, the Hi-Link board came with pins to access the UART and other interfaces.

2.1. Connectivity

Being designed as a system-on-chip for WiFi routers, the MT7688 comes with several networking peripherals along with some other typical interfaces.

- WiFi 802.11b/g/n
- Gigabit Ethernet
- 5 port 10/100 Ethernet switch
- I^2C
- SPI
- GPIO
- UART (x3)

The Onion Omega 2 module is designed around using the WiFi for networking, and has a small built in antenna. An add-on board for an Ethernet port is available, and all other interfaces are exposed on a header on the various "docks". The Hi-Link router board has 5 Ethernet ports, a U.FL connector for a WiFi antenna, and holes to add pins for the other interfaces.

3. Information

Onion provides a data sheet from MediaTek with information on accessing most of the interfaces. They also provide other specs on the modules and docks they sell, and git repositories for u-boot and other software they use.

Building up a working environment in 9 Front took a little work, as both the mips and spim code had not been used in quite a while. There is one MIPS compiler `vc` which is mips by default, and the spim compiler is a shell script `0c` that runs the mips compiler with an `-l` flag to produce spim output. Some minor patches were needed in `libc` and `ape` to get libraries and the rest of the programs to build without error.

3.1. Plan 9 rb Kernel

My first inspiration to try a MIPS based system was finding a kernel[4] written for a now out of production Mikrotik rb450g router board. The notes contained some useful information pertaining to some known issues with the MIPS24K cores, and a fix for the compiler. However, the rest of the kernel was not as useful, given the differences between the Atheros chip used on the Mikrotik board and the MT7688, and the many changes that had been made to 9 Front over the years.

3.2. 9 Front sgi Kernel

A 9 Front kernel[5] was written for an SGI Indy, and it had borrowed heavily from rb kernel. While the architecture specific kernel code had kept pace with the changes made in the portable 9 Front kernel code, the hardware and peripherals of the SGI Indy were very different from those of the MT7688.

3.3. Plan 9 loongson Kernel

A kernel[6] written for an old Loongson system was the only information I found that specifically dealt with little-endian spim issues. Thankfully some comments were left behind to point to where the endian differences caused alignment issues.

3.4. NetBSD

Since no previous work had been done with MediaTek MIPS CPUs on Plan9 or 9 Front, NetBSD[7] provided a valuable resource to supplement the available data sheet for the addresses of various registers, the secondary interrupt controller, the set up of transmit and receive descriptors for the Ethernet device, and additional notes on work arounds for known bugs.

4. Planning

The main considerations for making the kernel came down to two issues.

•How Plan 9 and 9 Front have diverged.

Several functions had been moved from platform specific code to the portable code in 9 Front, and 9 Front had moved to using rc for more things later in the boot process. There were also several differences in the contents and layout of `mach` and `proc` data structures.

•The features that were universal among MIPS systems.

While there have been slight variations over the years, several things were effectively universal among the MIPS kernels I had access to. Status registers, the on board counter and interrupt controller, and the MIPS style of memory layout and memory management had largely remained the same.

4.1. Universals

The majority of the machine specific assembly came from the rb kernel. Not only because of the features specific to the MIPS24K platform, but also because of a known bug with cache writes that required additional no-op instructions to avoid losing data. The MIPS24K platform also had a change in the WAIT instruction to aid in power saving for the small devices it was planned to be used in.

The trap code among the MIPS kernels was mostly the same. That, along with the similarities in the counter and the interrupt code, was enough to use the counter interrupt to call the scheduler and get the boot process moving along. The MT7688 also lacks a floating point unit, so I used the FP trap code form the rb kernel, along with the floating point emulator code.

The MMU code was also largely the same. The main difference being the number of TLB enties available on each make of CPU. Notes from the rb kernel had mentioned an issue with a large number of TLB misses on the MIPS24KEc based Atheros chip, and the MT7688 had a similar issue.

The UART was a 16550 compatible, so existing UART drivers were used for that.

4.2. Particulars

The UARTs and most of the other interfaces had interrupt handling done on a secondary interrupt controller. This is programmable to do "high" and "low" priority interrupts, and those are then sent to interrupts 2 and 3 on the primary MIPS interrupt controller. Since no previous MediaTek kernel had been done, this was written from scratch.

The Ethernet interface made for a unique challenge. While the functioning of the Ethernet by itself was unremarkable, it was physically wired directly into an Ethernet switch that was part of the SoC. This meant communication outside the system required initializing the switch. NetBSD provided an example of a basic configuration for the switch.

5. Building

Both the Onion Omega 2 and the Hi-Link router came installed with u-boot. The Onion Omega 2 expansion dock had a USB port to allow for loading kernels off a thumb drive using fatload, and another micro USB port that provided power and a UART to USB interface. The Hi-Link board came with two DB9 serial ports, but I opted for using a USB to UART adapter and soldered pins into the exposed UART through holes. The u-boot installed on the Hi-Link board initialized the external Ethernet ports and the switch, which allowed for loading kernels and plan9.ini files via tftp.

5.1. Building the Environment

The first thing was to build all the spim libraries and programs, which uncovered some neglected code in both the mips and spim libraries. For the most part, spim just uses the mips code, except for a few cases were the endian order is an issue. So errors when building could be either bugs in mips or spim.

The first issue was that some of the code for spim in the APE library was just missing. This was quickly, but not properly, fixed by copying mips code into spim directories, like `/spim/include/ape` and `/sys/src/ape/lib/9/spim`.

While checking through the standard library files for any obvious issues, I found a couple in `/spim/include/u.h`. Since u.h was just a copy of the mips version, it helpfully noted that `FPdbleword` was in a big-endian configuration. Reversing the placement of `lo` and `hi` would match other little-endian systems:

```
union FPdbleword
{
    double  x;
    struct {    /* little endian */
        ulong lo;
        ulong hi;
    };
};
```

The other was the `va_arg()` macro, which was also changed to match what other 32 bit little-endian systems did:

```
#define va_arg(list, mode)\
    ((sizeof(mode) == 1)?\
        ((list += 4), (mode*)list)[-4]:\    /* changed -1 to -4 */
    (sizeof(mode) == 2)?\
        ((list += 4), (mode*)list)[-2]:\    /* changed -1 to -2 */
        ((list += sizeof(mode)), (mode*)list)[-1])
```

5.2. Initial Kernel

The existing code base was enough to get the system to boot through the initial `start()` in `l.s` , through `main()` and `init0()`. After that, the boot process would die somewhere after the hand off to userspace with a malformed error message. Adding stack and register dumps ended up pointing to the first `open()` syscall in

/sys/src/9/port/initcode.c. The loongson kernel, being the only other published kernel to use spim, noted a difference in spim's libc and the alignment of syscall arguments.

This difference stems from /sys/src/libc/9syscall/mkfile, where mips simply does SYSCALL, while spim adds 4 to the stack pointer (R29) before SYSCALL:

```
case mips
    echo TEXT $i'(SB)', 1, '$0'
    echo MOVW R1, '0(FP)'
    echo MOVW '$'$n, R1
    echo SYSCALL
    ...
case spim
    echo TEXT $i'(SB)', 1, '$0'
    echo MOVW R1, '0(FP)'
    echo MOVW '$'$n, R1
    echo ADD '$4',R29
    echo SYSCALL
    echo ADD '$-4',R29
```

However, in mips syscall() it would advance the stack pointer by 4 (BY2WD):

```
up->s = *((Sargs*)(sp+BY2WD));
```

While in the spim example I found in loongson, since the addition happened before, the spim syscall() saved the stack pointer as is:

```
up->s = *((Sargs*)(sp));
```

I wasn't able to find any documentation on why this was done in two different ways, so I decided to leave /sys/src/libc/9syscall/mkfile as it was, and stay with the differences in syscall().

5.3. Another Endian Issue

After the syscall argument fix, another problem was found when calling bind() in initcode.c. The arguments for the kernel console '#c' and env '#e' devices being passed to it were being truncated to just '#'. This led through strlen() to a bug in the machine specific code for spim in libc.

The file /sys/src/libc/spim/strchr.s had been partly changed for little-endian use, but part of the loop that handled word alignment was still set up for big-endian:

```
l3:
	MOVW	$0xff000000, R6		/* byte 4 */
	MOVW	$0x00ff0000, R7		/* byte 3 */

l4:
	MOVW	(R3), R5
	ADDU	$4, R3
	AND R6,R5, R1		/* byte 4 */
	AND R7,R5, R2		/* byte 3 */
	BEQ R1, b0
	AND $0xff00,R5, R1	/* byte 2 */
	BEQ R2, b1
	AND $0xff,R5, R2	/* byte 1 */
	BEQ R1, b2
	BNE R2, l4
```

The order needed to be reversed:

```
l4:
	MOVW	(R3), R5
	ADDU	$4, R3
	AND $0xff,R5, R1	/* byte 1 */
	AND $0xff00,R5, R2	/* byte 2 */
	BEQ R1, b0
	AND R7,R5, R1		/* byte 3 */
	BEQ R2, b1
	AND R6,R5, R2		/* byte 4 */
	BEQ R1, b2
	BNE R2, l4
```

With that done, `strlen()` worked correctly, and the system would now continue to boot past `initcode.c` and through the rest of the boot process.

5.4. UART

The UART drivers had assumed that they were being used in a more full featured Plan 9 environment. Notably that there would be something to handle an output queue. After setting up cons, printing to the UART would halt after `serialoq` was filled. Setting `serialoq` to point to nil instructed the portable cons code to continue using the basic `uartputs()` function to print directly into the UART transmit register.

At this point, using the 9 Front feature `!rc` at the boot prompt allowed for completing the boot process, and the system to run with what was provided in the paqfs compiled with the kernel.

5.5. Ethernet and Switch

A Plan 9 system isn't much without a network, so the next step was to get the onboard Ethernet working. There were 3 things that were noteworthy about writing the Ethernet driver.

The first is particular to MIPS and how the kernel memory is mapped. 0x80000000 – 0x9FFFFFFF is `KSEG0` and is cached kernel space. 0xA0000000 – 0xBFFFFFFF is `KSEG1` and matches `KSEG0` but is uncached. `xspanalloc()` was used to get a contiguous space in memory for the transmit and recieve descriptor ring and the pointer returned by `xspanalloc()` was then shifted to the coresponding `KESG1` address. These shifted pointers were used throughout the rest of the driver to avoid the need for specific cache flushing instruction anytime the descriptors were writen to.

Second was that the onboard Ethernet was wired directly into an onboard Ethernet switch. The documentation seems to show 7 ports total, but the HLK7688A router board has 5 external ports, and 1 internal port connected to the internal Ethernet interface. Configuring the switch is pretty straight forward, as the registers are mapped into the same block of memory as the other devices on the SoC. However, a standard "switchdev" is not available in 9 Front's portable kernel code base, so there is not currently a way to do a Plan 9 style attach or file system to do configuration with. That would be a much larger project, and in the meantime I used an example from NetBSD to simply set up the standard 4 LAN ports to act as a basic unmanaged switch with the Ethernet, allowing communication to the outside world.

The third issue is fairly minor, which is that the Ethernet switch provides the physical interface (PHY) for the onboard Ethernet (MAC), so the registers for reading and writing the MII are in the switch. Other than that, the existing portable code for MII works fine.

With that done, and with a valid nvram file added to the kernel, I was able to get the kernel to boot, initialize the ethernet and switch, authenticate with the auth server and connect to the file server. Aside from some bugs in the floating point emulation, it behaves like any other cpu server on my 9 Front grid.

5.6. Floating Point Emulation

The floating point emulation code that came with the rb kernel was written with a big-endian MIPS system in mind. This is compounded by the emulator storing everything internally as a double, so the order in which pairs of `lo` and `hi` words are read and stored is important. While the code compiles, and is called upon by the kernel, it often produces bad results. Conspicuously, the results are often 0, or endless loops of multiplying by 0. This is likely a results of reading an empty `hi` word as the `lo` word. This will take further testing to find and check all the ways in which the emulator stores and reads various combination of single and double word values.

The most notable issue this causes is that `awk` fails to work, often getting stuck in a loop. It is possible to `rcpu` from another 9 Front terminal and run `rio`, but the drawing functions in some application would fail. An example is that `games/catclock` would display the background image of the cat, but fail to draw the eyes, tail, or hands of the clock.

6. Future

Finishing up drivers for the remaining interfaces is pretty straight forward, with the exception of the Ethernet switch. There is no existing portable code for exposing the switch hardware settings as a file system, so that will need to be designed from scratch. More thought will need to be put into that, as even the rather low end switch in the MT7688 still has global and per-port settings, VLAN and priority settings, and some ability to react to broadcast storms. 9 Front also includes a bridge device that does VLAN, so a future switch interface would need to be compatible with that.

While the current owner of the MIPS property has announced[9] a switch to RISC-V for future designs, existing MIPS license holders continue to produce chips, and they often find their way into IoT like devices. So while there is not a long future for the MIPS ISA, embedded systems tend to stay in production well past their prime, and MIPS in particular is a low cost and well documented platform found in a variety of devices that work well in Plan 9's well integrated network environment.

References

[1] R. Pike, D. Presotto, K. Thompson, H. Trickey, "Plan 9 from Bell Labs", UKUUG Proc. of the Summer 1990 Conf. , London, England, 1990.

[2] Onion Corporation, https://onion.io/

[3] Shenzhen Hi-Link Electronic Co., Ltd. https://hlktech.net/

[4] Plan9 rb kernel, https://github.com/0intro/plan9-contrib/tree/main/sys/src/9/rb

[5] 9Front source code, /sys/src/9/sgi/, http://git.9front.org/plan9front/plan9front/HEAD/info.html

[6] Plan9 loongson kernel, https://github.com/0intro/plan9-contrib/tree/main/sys/src/9/loongson

[7] NetBSD ralink kernel, https://github.com/NetBSD/src/tree/trunk/sys/arch/mips/ralink

[8] MIPS Tech LLC, "MIPS32 24K Processor Core Family Software User's Manual" , p. 335 & 339, https://www.mips.com/products/classic/

[9] MIPS Tech LLC, "MIPS Pivots to RISC-V with Best-In-Class Performance and Scalability", https://www.mips.com/news/mips-pivots-to-risc-with-best-in-class-performance-and-scalability/

Plan 9 on 64-bit RISC-V

Geoff Collyer
geoff@collyer.net

ABSTRACT

We have ported Plan 9 to several RISC-V 'Unix-capable' (RV64GCSU) implementations: the Microchip™ Polarfire Icicle™, the *tinyemu* emulator, a pre-release Beagle V, and the SiFive™ HiFive Unmatched™. The Beagle V and Unmatched ports are currently usable, if you can find the hardware, but consume more power than necessary when idle. The Unmatched's SBI (Supervisor Binary Interface) lies randomly about which harts are running, so Plan 9 only boots successfully once in a while. The only plausible hardware is currently the Icicle.

This paper describes the porting process and makes recommendations. Some of this work is still in progress, as of 19 March 2023.

1. Position Statement

The RISC-V architecture is elegant; I don't have any serious criticism of it, including at least the standard IMAFD extensions for Unix-capable systems. (These are base Integer instructions, Multiplication and division, Atomic memory operations, and single- and double-precision floating point. IMAFD is also written as 'G'. 'C' indicates ability to execute the compressed instructions.) *However*, proposed and approved extensions beyond IMAFD, and other additions, are often flawed or downright rubbish.

> *People admire complexity.*
>
> — Rob Pike

> *Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defence against complexity.*
>
> — David Gelernter

Much of this is hardware and software adopted unthinkingly from PCs and ARM devices, regardless of technical merit, probably to re-use existing designs and IP and Linux code. Several large corporations (e.g., Intel, Alibaba) seem unable to comprehend the 'R' in 'RISC' (i.e., keep it simple, stupid), but visible complexity is not actually required for good performance, even if that's easier for the hardware designers. *Unnecessary visible complexity is a failure of design.* I don't really expect a company with an instruction set with over 800 instructions (over 2,000 by some reckoning), hundreds of MSRs, and a 2,680-page Ethernet controller manual to understand this. We'll revisit this in *Recommendations and Observations*.

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

1.1. Demented Standards

How do you tell a bad standard? If it begins with "I", e.g, I2O, IPMI. If it ends with "I", e.g., ACPI, EFI, IPMI, etc. If it has the word "intelligent" in it, e.g., I2O, IPMI. Or, the best, if it has all three, e.g., IPMI.

— Ron Minnich

'Device trees', ACPI, (U)EFI and GUIDs are problems, not solutions, and we try to avoid them at all costs. (Why use meaningful names when you can use long, meaningless strings of hex digits?) The RISC-V Platform Specification subcommittee is just flat-out wrong to adopt virtually every mistake ever made on the IBM PC or ARM. The RISC-V Platform Specification is disappointing; RISC-V presented an opportunity to rethink and replace this string of disasters, most of which provide very little benefit.

1.2. Wretched Hardware

Throw out the hardware, let's do it right.

— Steely Dan, *Aja*

Life is too short to deal with SD and MMC cards, GPIOs, PHYs, I2C, SPI and other single-bit interfaces to guaranteed-model-specific hardware, and this crud shouldn't be necessary; hardware should be usable immediately after coming out of reset. If the BIOS or *U-boot* initialize devices sufficiently to use them, that's good enough.

2. Background

In July 2020, the Microchip Polarfire Icicle board [Mic] was due to be the first available RISC-V [Pat2017, Int] system that looked capable of running a Unix-like operating system, including paging hardware, a gigabyte of RAM (which turns out to be actually 2GB in 2 banks), and gigabit Ethernet, [Xil] with multiple CPUs (also called cores and RISC-V *harts*) capable of 64-bit and (in theory) 32-bit operation. It has one SiFive E51 RV64IMA core that lacks supervisor mode (a 'hobbled' hart) and starts the other four, which are SiFive U54 RV64GC cores at 600 MHz. The board contains no graphics hardware. We assume conformity to a minimum Privileged ISA Specification of 1.10.

Richard Miller had a 32-bit RISC-V Plan 9 [Pik1990] C compiler suite [Mil2020] already, and was willing to create a 64-bit compiler suite. Without this, I would not have started this porting effort.

I originally planned to port the Plan 9 *9k* kernel, which already ran on 64-bit `amd64` systems and could exploit a large address space, to RV32GC mode on the Icicle while Richard worked on the RV64 C compiler, then use the RV32GC port as a basis for an RV64GC port when the RV64 compiler was ready. *9k* implements the same system calls and a subset of the devices that *9* implements. The major device drivers are now identical in *9* and *9k*.

2.1. Timeline

The hardware was due to arrive in mid-September 2020, which eventually slipped to mid-October. In the meantime, we developed using the *tinyemu* emulator, which emulates both RV32 and RV64 architectures on any processor architecture, though only a single emulated processor. *Tinyemu* supplies no SBI and starts the kernel in machine mode; all the other implementations have an SBI and start the kernel in supervisor mode (except the RVB-ICE, which appears to start it in machine mode). Richard ported it to Plan 9 and added a serial port and *virtio* Ethernet. I made small changes to it to improve debugging capabilities and it has proven helpful in finding bugs where the hardware's response has been to just sit there dumbly. Perhaps the SBI could accept requests on a UART to dump a hart's registers.

When the hardware arrived, we had a *9k* kernel running on *tinyemu*, but we then discovered some things that would require changes. *Tinyemu* starts our kernel in RISC-V 'machine' mode, in which paging is disabled, but all machine facilities may be configured. The lack of paging encourages starting RAM at `0x80000000` or higher, to match Unix kernel conventions. By early December, we had a 64-bit kernel running on one CPU of the Icicle board using 39-bit virtual addresses ('Sv39'). By late December, all U54 CPUs were scheduling processes. A few C compiler fixes arrived through mid-January. After that, various mysterious misbehaviour disappeared. By early February, graceful reboot was working and by mid-February, paging with 48-bit virtual addresses ('Sv48') was working. The system seems to be complete and solid enough to use as a CPU server, and should be relatively easy to adapt to future RV64G systems.

Since then, we have made minor fixes, code and performance improvements, and adapted to various RISC-V systems, including improving SoC (system-on-chip) configurability. In particular, self-checking code has been added to verify sanity in various conditions, and to attempt to tolerate the unexpected.

2.2. Hardware and Firmware

A note on terminology: the *CLINT* is the per-CPU simple interrupt controller; the *PLIC* is the system-wide more-complex interrupt controller. The PLIC feeds into the CLINTs as the external interrupt signal. The *SBI*[Int2022] is a sort of BIOS, but unlike a PC BIOS, it cannot be circumvented.

On the hardware, the boot ROM/flash starts (typically) OpenSBI which then starts *U-boot*, which starts our kernel in 'supervisor' mode, from which there is no escape, with additional undocumented restrictions:

- read-only (or no) access to the CLINT's timer registers;
- have to use SBI calls to set the CLINT timer (and maybe send and clear IPIs);
- SBI v0.2 HSM (hart state management) calls are not implemented in the provided Icicle OpenSBI;
- *U-boot* on the Icicle only starts all CPUs (*hart*s in RISC-V terminology) if one uses the `bootm` command with an FDT to run a *uImage* claiming to be a Linux kernel.

A result is that we can't switch the Icicle into RV32GC mode with the stock boot 'ROM', though it is possible in machine mode. So we abandoned the 32-bit port since it can't run on the available hardware, though it still ran in *tinyemu*, as the current wave of Unix-capable systems are all RV64GC, as will be any Unix-capable systems with more than 2GB of RAM.

There are conflicting accounts of the details of how RISC-V harts are started, particularly at what PC. *U-boot* on Icicle starts them all at once at the entry point in the *uImage* file, while the Beagle V's and Unmatched's *U-boot* starts only hart 1.

2.2.1. Polarfire Icicle

There are other bits of ill-documented hardware:

- there's an L2 cache which adheres to RISC-V cache coherence principles, so can be largely ignored;
- the PLIC context ids apparently have consecutive values starting at 0: E51 hart 0 M (machine) mode, U54 hart 1 M, hart 1 S (supervisor) mode, hart 2 M, hart 2 S, hart 3 M, hart 3 S, hart 4 M, and hart 4 S. These should be predictable or discoverable without consulting a 'device tree'.

2.2.2. SiFive U740

On SiFive-U740-based systems, use of the `WFI` instruction, instead of `PAUSE`, to save power when idling produces strange and varied behaviour: console serial output gets stuck, or time gradually stops advancing, or the system becomes very busy, possibly servicing interrupts. Without the Unmatched hardware reference manual, it's difficult to understand what is going wrong.

2.2.2.1. Beagle V

The now-cancelled Beagle V has 8 GB of RAM, and an L2 cache that is *not* coherent with DMA, thus requiring manual cache flushing, unlike the other systems. (This was claimed to be a bug that would be fixed in production hardware, the JH7110 SoC) It also has a newer OpenSBI implementation that provides the HSM operations.

2.2.2.2. HiFive Unmatched

OpenSBI's *sbi_get_hart_status* appears to often report the wrong hart as the sole started hart.

2.2.2.3. Starfive™ Visionfive™ 2

The newly-arrived successor to the Beagle V, incorporating the JH7110 SoC, currently gets as far as `/boot/boot` running `ipconfig` with all 4 U74 harts running. The Synopsys™ DWMAC is version 5.20, which is newer than the Beagle V's version 3.7, and incompatible. Many clock signals had to be enabled and components taken out of reset via CRGs (system control registers) in order to communicate with the DWMAC at all. Work continues.

2.2.3. XuanTie™/T-Head RVB-ICE

This uses the XuanTie C910 CPU, which is claimed to be quite fast. After enabling paging, something goes off the rails. Linux runs on this hardware, so presumably there's some extremely obscure magic needed, despite T-Head's claim of RISC-V compatibility. English documentation not generated by Google Translate would help.

2.3. RISC-V Peculiarities

Memory alignment requirements are stricter than most people are used to: natural alignment for scalars up to and including `vlong`†. Otherwise, we get alignment exceptions. The 64-bit compiler promotes most scalars to `long` when pushing them as function arguments, only `vlongs`, `doubles`, pointers, and some `structs` are wider. However, there can be gaps on the stack, e.g., when pushing an `int` then a pointer.

Except on the Beagle V, all CPUS, memory caches and DMA accesses are coherent, which is a delight. The RISC-V specifications encourage this, but it is nevertheless unusual, surprising and noteworthy for RISC designs.

3. Plan 9 Changes

These are largely confined to the architecture-dependent source directories.

3.1. Removed Assumption of Memory at Address Zero

The original *9k* assumed that RAM started at physical address 0, and it took some trial-and-error to find and repair the myriad dependencies, notably in initial memory discovery and allocation.

† On Plan 9, `vlong` is `long long`, which is always 64 bits.

3.2. No Virtual Page Table

The technique of the 'virtual page table' [MIT] (VPT), which injects the page table into itself as a top-level PTE, is used in the 386 and amd64 ports, but appears to be inapplicable to RISC-V. Lifting a level 1 PTE into the root (level 2) PTE would vastly increase the address space that it covers, since size is implied by level. So some page table updates had to be made explicit and do their own allocations, which is clearer anyway (the existing VPT code is obscure).

3.3. Variable Page Sizes and Page Table Levels

The system implements *Sv39*, *Sv48*, *Sv57*, and *Sv64* paging, where available. The supported hardware implements only *Sv39* in RV64, but *tinyemu* implements *Sv48* too. *Sv57* and *Sv64* are untested to date, but are straight-forward extensions from *Sv48*.

3.4. SoC Configuration

Configuration for a new SoC requires editing the conf sections of kernel configuration files, which now include descriptions, in C, of the SoC's devices, and fundamental addresses needed early or in mkfile are specified in the /sys/src/9k/rv directory, in the file *arch*/defs, where *arch* is a short name for the subarchitecture (e.g., te for *tinyemu*). The appendix contains an example of the 64-bit *tinyemu* configuration. See tecpu and pfcpu for complete examples.

3.5. Starting CPUs During Bootstrap

On x86 systems, a single CPU starts at bootstrap, and it then starts the others. RISC-V systems may start CPUs (*hart*s) at any time. The Icicle starts them all at once when *U-boot*'s bootm command starts the kernel, which is necessary because its SBI lacks the HSM commands that would otherwise be needed. The Beagle V and HiFive Unmatched both start a single CPU (or at least try to), which uses the SBI HSM calls to start the others. The start-up code now copes with those possibilities, and the situation of having just been restarted via /dev/reboot.

3.6. A C Idiom

In a C expression such as in the following, using Plan 9 types:

```
uvlong uvl, va;
uvl &= ~((1<<5) - 1);       /* zero low 5 bits */
uvl = va & ~((1U<<12) - 1); /* get pure page number */
```

the result will probably not be what was intended. The ~ operator will have an int or uint operand, yielding a result of the same type, 32 bits wide. This result will be widened for the & or &= operator, but it may be zero-extended, thus ensuring that the result in uvl will have zeroes in its upper 32 bits. In particular, 64-bit physical addresses of RAM on RISC-V were being truncated. *6c* and now *jc* detect this inadvertent zero-extension in the uint case.

Ensuring that the operand (and thus result type) of ~ is vlong or uvlong avoids this problem. We have made this change throughout *9k*.

4. Performance

These are times to build the Plan 9 `rv` kernel from scratch mostly on RV64GC systems with 1Gb/s Ethernet using the same 10Gb/s Ethernet file server, except as noted. These were all effectively diskless, as is normal for Plan 9 systems. To load caches before measuring, these commands were executed:

```
mk clean; mk; mk clean; time mk >/dev/null
```

and yielded these results:

user	sys	real	cores	system
1.07u	1.73s	2.04r	4-core	amd64 Xeon 3.8GHz, 10GbE
1.96u	1.67s	2.58r	4-core	intel nuc5i7 3.1GHz (32-bit)
5.66u	3.60s	9.35r	4-core	arm raspberry pi 4 1.5GHz (32-bit)
10.91u	7.79s	11.46r	4-core	dual-issue hifive unmatched 1.2GHz
14.60u	7.09s	14.47r	4-core	single-issue Icicle 600MHz, no ipis
10.14u	13.63s	19.10r	2-core	arm v7 trimslice 1GHz (32-bit)
14.99u	9.11s	50.91r	2-core	dual-issue pre-release beagle v 1GHz *
38.52u	20.44s	64.98r	1-core	mips 24k routerboard 680MHz, no fp (32-bit)
169.53u	45.98s	260.60r	1-core	tinyemu on nuc5i7 3.1GHz HZ=200
281.80u	90.86s	464.07r	1-core	tinyemu on nuc5i7 3.1GHz HZ=100

See this earlier paper [Col2010] for comparison with older Plan 9 systems of various architectures.

5. Recommendations and Observations

Microchip's documentation seems to be unclear if it's intended for someone repackaging the hardware or for the ultimate end user. It often specifies that some value is programmable but doesn't provide the choice of value used in the Icicle. It would be helpful to have end user documentation.

The RISC-V architecture tries to leave some things unspecified to allow implementations some leeway, requiring that platform documentation provide the actual values implemented, but the platform makers don't always do so. Concern for RISC-V implementors should be balanced with concern for users; vagueness is rarely useful to system programmers. It would be more helpful to be able to query such values programmatically without consulting a 'device tree'.

All the timers provided require *a priori* knowledge of their frequencies. To let software determine the actual frequencies, it would be very helpful to have a real-time clock that ticks at a known, fixed rate (e.g., 100 times per second) or a register containing the (fixed) CLINT timer frequency. As it stands, the frequencies have to be supplied to software.

Detecting and reporting infinitely-recursive traps (perhaps in SBI) would be quite helpful during development, for example, if the STVEC CSR (Control and Status Register) contains a no-longer valid virtual address. We have modified *tinyemu* to do this.

Requiring all RISC-V systems (or at least Unix-capable ones) to have an 8250-compatible console UART at a common, fixed physical address and a common frequency would help with porting. SiFive has its own non-8250-compatible UART.

Micro-USB connectors need to be braced very firmly; a slight tug on an attached cable should not yank the connector off the board. The Unmatched board is quite flimsy. Its SD card slot isn't much better.

The Icicle's power cable is fragile and prone to interrupting power when flexed.

Suppliers need to implement both PXE booting and the `saveenv` command in their *U-boot* variants from the very start. These are important capabilities for kernel

* using a different, 1Gb/s file server

developers and must not be pushed off into the future. The Icicle at least has working PXE booting on one Ethernet, but no saveenv command, so automatic booting of Plan 9 kernels at reset won't work. The otherwise-promising Sipeed Nezha board's *U-boot* lacks PXE booting entirely, which makes it too much of a hassle to be worthwhile.

5.1. Assessment of RISC-V

In general, RISC-V seems to be a pleasant architecture with a few minor infelicities. (Implementing graceful reboot on the Icicle was a challenge.) Some additions and extensions add the sort of unnecessary and clumsy complexity that has made X86 the dog's breakfast that it is (e.g., the XuanTie C910). The XuanTie processors seem to have reintroduced all the mistakes that ARM made and that RISC-V carefully omitted. SBI is another story altogether.

The CSRR* instructions hard-code the CSR number; they would be easier to invoke from C if the CSR number were held in *rs2* instead, thus allowing use of less assembly language while avoiding executing code generated on-the-fly.

If the kernel's stack pointer contains an invalid address (e.g., a change in page tables makes it invalid), the trap to report the invalid address will trap endlessly due to an invalid stack pointer. SBI could perhaps note and report this.

A register that returns the PLIC context for machine mode on the current CPU would ease PLIC use without requiring external assistance.

It would be useful in a few cases to be able to determine the nominal privilege mode, even if it's virtualized. Being able to probe for a given CSR without causing a trap would help too.

Machine mode seems dubious. Supervisor mode should be able to control the (possibly virtual) machine, and a mode without the possibility of paging is not helpful. Running Plan 9 or a UNIX kernel in machine mode with reasonable efficiency is infeasible; the kernel needs to use virtual memory. When running on *tinyemu*, we initially configure some M-mode-only facilities, delegate any possible M-mode traps and interrupts to S-mode, and switch to S-mode. Thereafter, we catch and forward M-mode traps to S-mode.

The focus on undetectable virtualization seems excessive. Being able to programmatically at least confirm various attributes of the environment in machine and supervisor modes would be helpful.

PMP (Physical Memory Protection) is probably unnecessary on systems with MMUs, and is a bit of a pain to configure.

5.1.1. SBI

SBI seems largely unnecessary yet it insists on disabling some hardware features that a kernel could use directly and requiring use of SBI instead. I don't want or need another layer of software between the hardware and my kernel. The SBI specification is imprecise. For example, what are the units of the timer functions? Which timer do they set? What is that timer's frequency? Is the timer global or per-hart? Under what conditions can *sbi_send_ipi* (FID 0, EID 4) fail? It has been seen to fail with valid hart ids on OpenSBI. Which supervisor-mode facilities has it disabled?

There is some evidence of bugs in OpenSBI calls, e.g., *sbi_get_hart_status*.

6. Future Work

Bootstrapping is clumsy; a future upgrade to the Icicle's *U-boot* should yield an automatic way to PXE boot at power-on or reset. (Until then, *fshalt*(8) provides graceful reboots.)

There is Icicle and Unmatched hardware that we do not (yet) drive: an open PCI-E

slot, an FPGA on the Icicle, and USB controller(s). Icicle documentation for USB is not obviously locatable. Icicle PCI-E requires newer HSS (hart software services) firmware.

7. Availability

A reasonably-stable distribution of the RISC-V kernel and the compiler used to build it, along with support files, is maintained in

`https://9p.io/sources/contrib/geoff/riscv/dist.9k-rv.tgz`.

8. Acknowledgements

Richard Miller developed the 32-bit and 64-bit RISC-V C compiler suites for Plan 9. He has been very helpful, fixing (minor) bugs, helping to find my obscure bugs, contributing *sdio/mmc* drivers, and extending the assemblers. The late Jim McKie created the 64-bit *9k* kernel and Charles Forsyth created the `amd64` compiler suite for the first architecture. We are building, of course, on years of work at Bell Labs creating and developing Plan 9.

References

Col2010. Geoff Collyer, ''Recent Plan 9 Work at Bell Labs,'' *Fifth International Workshop on Plan 9*, Seattle, invited talk, `http://www.collyer.net/who/geoff/ports.pdf` (October 2010).

Int2022. RISC-V International, *RISC-V Supervisor Binary Interface Specification*, `https://github.com/riscv/riscv-sbi-doc/blob/master/riscv-sbi.adoc`, 2022.

Int. RISC-V International, *RISC-V home*, `http://riscv.org`.

Mic. Microsemi, *Icicle*, `https://www.microsemi.com/products/fpga-soc`.

Mil2020. Richard Miller, *A Plan 9 C Compiler for RISC-V RV32GC and RV64GC*, `https://ossg.bcs.org/wp-content/uploads/criscv64.pdf`, 19 Oct 2020.

MIT. MIT, *Address translation and sharing using page tables*, `https://pdos.csail.mit.edu/6.828/2007/lec/l5.html`.

Pat2017. David Patterson, Andrew Waterman, *The RISC-V Reader,* Strawberry Canyon (7 November 2017). `http://riscvbook.com`

Pik1990. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, ''Plan 9 from Bell Labs,'' *Proc. of the Summer 1990 UKUUG Conf., London*, pp. 1-9 (July, 1990).

Xil. Xilinx, *Zynq 7000 SoC Technical Reference Manual (UG585)*, `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`.

Appendix

```
/* 64-bit tinyemu configuration from tecpu kernel config */
#include "riscv64.h"
int cpuserver = 1;
int idlepause = 1;
uvlong cpuhz = 156*1000*1000;  /* from timesync, emulated on 3ghz nuc */
uvlong timebase = 10*1000*1000;  /* clint ticks per second */
Membank membanks[] = {      /* (address, size) pairs */
      PHYSMEM, BANK0SIZE,
      0
};
char defnvram[] = "/boot/nvram";
uintptr uartregs[] = { PAUart0 };
int nuart = nelem(uartregs);
vlong uartfreq = 384000;
uchar ether0mac[] = { 2, 0, 0, 0, 0, 1 };
/* the emulated plic doesn't seem to follow the spec; we ignore it. */
Soc soc = {
      .clint     = (char *)PAClint,
      .uart = (char *)PAUart0,
      .plic = (char *)0x40100000,
      .ether[0] = (char *)0x40011000,
      .hobbled= 0,                /* only 1 hart */
};
Ioconf socconf[] = { /* devices without drivers that vmap their regs */
      { "clint", 64*KB, &soc.clint, },
      { "uart",  PGSZ, &soc.uart, 1, },
      { "plic",  4*MB, &soc.plic, },         /* common but smaller */
      0
};
Ioconf ioconfs[] = {  /* devices whose drivers vmap their regs */
      { "ether", 2*PGSZ, &soc.ether[0], 2, },
      0
};
```

Hell Freezes Over:
Freezing Limbo modules to reduce Inferno's memory footprint

David Boddie
david@boddie.org.uk

ABSTRACT

While Inferno is capable of running on systems with a modest amount of memory, it is sometimes useful to be able to reduce the memory footprint of the running system. This is particularly true of embedded systems with only a few hundred kilobytes of RAM. This paper explores an approach where Dis virtual machine code is retained ("frozen") in flash memory instead of being loaded into precious RAM.

Introduction

Inferno was designed to have minimal hardware requirements, with a stated baseline of 1 MB of memory and no memory mapping hardware [1]. It was suggested in early publicity [2] that about 512 KB each of RAM and ROM would be enough to "run an interesting system." Ports of Inferno to several systems [3] were made and explored, including some with low amounts of memory [4]. It was also shown to be possible to run Inferno with only 512 KB of RAM using a custom memory allocator [5].

Today, microcontrollers with several hundred kilobytes of RAM and flash memory can be readily obtained. For example, microcontrollers using cores in the ARM Cortex-M4 series can be found with up to 256 KB of RAM, and those in the Cortex-M7 series can be found with 1 MB or more, making them potential targets for Inferno ports [6]. The discussion below focuses on targets at the lower end of this range, specifically the STM32F405RGT6 microcontroller [7] which has 192 KB of RAM and 1 MB of flash memory.

Challenges

The first challenge a port to a microcontroller faces is the availability of a compiler. In this case [8], the Thumb compiler was retrieved from the commit history of the Inferno repository and updated slightly. Beyond this, the two main challenges are the amounts of flash memory and RAM available to use.

The amount of flash memory constrains the size of the operating system and limits the features it can include. In the worst case it may not even be possible to fit a system into the space available. Around 256 KB may be required for a minimal set of features, so the target under discussion has plenty of space for a reasonable system.

The amount of RAM, however, is more limiting. Most Inferno systems tend to run in several megabytes of RAM, and many are copied into RAM on boot-up. Although it was found [5] that 512 KB could be enough to run a useful system if the kernel is run from flash memory, problems can still occur when memory allocation occurs in the running system. In the case of the STM32 target hardware these problems appear in the form of heap allocation errors when trying to run programs.

There are two obvious strategies for dealing with the shortage of RAM in this case:

1. Reduce the amount used by the operating system to begin with.
2. Reduce the amount of memory requested when loading Limbo [9] modules in the form of `.dis` files.

The second approach seemed to be a useful starting point and would still be a beneficial approach for systems that have enough free RAM to begin with, but which use large Limbo modules, since they will still encounter limits to the amount of memory they can allocate.

One strategy for dealing with the shortage of RAM suggested itself when investigating how the Dis virtual machine loads Limbo modules.

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

Loading Limbo modules

When the Dis virtual machine [10] loads a Limbo module from a file, it performs a number of steps:

1. In the `readmod` function, raw module data is loaded into a newly allocated memory buffer. The `parsemod` function is then called with a new `Module` object to create the suitable data structures needed to run the module code.
2. The `parsemod` function checks for a suitable magic number at the start of the data, returning if unsuccessful.
3. Each section in the module data is decoded and the amount of memory required to represent it is allocated. The first section contains code, which is decoded and expanded into a sequence of `Inst` structures suitable for execution.
4. The members of the `Module` object are filled in and the module is linked to an existing list of modules.

Dis normally patches the code as it decodes it into its runnable form, adjusting the instructions that perform branches to refer to the addresses of their target instructions in memory. This patching is necessary because it is not possible to know ahead of time where the code will be allocated.

When processing the code section of the module, two things stand out. The amount of memory used to store the instructions is typically larger than the size of the corresponding code in the original file. In addition, once the instructions have been decoded, they never change. This suggests that they could be retained in flash memory in a pre-decoded form through a process of "freezing" at build-time.

Freezing code

Support for frozen modules requires three things: a tool to create frozen module data at build-time, a way for the virtual machine to recognize frozen modules, and an extension to the virtual machine that can load frozen module data.

A `freeze` tool was created to perform the task of reading an existing module file and writing data structures that can be compiled into the operating system. Reusing code from the virtual machine, the tool decodes the instructions in the module's code section, ready to be executed at run-time, but writes the other sections in their original form for the virtual machine to handle in the usual way.

The necessary patching of code might appear to be a problem. However, the address of frozen code is fixed at build-time, allowing the branch addresses to be pre-calculated and avoiding any need to modify the instructions at run-time. Each sequence of instructions is represented by a sequence of 32-bit words written in assembly language. Branches within the code are represented as references to offsets from the start of each sequence.

Another purpose of the `freeze` tool is to create a placeholder module file for each original module. This contains the magic number, `0xFD15`, and the path of the original file within the root file system, allowing the virtual machine to recognize that it refers to frozen data for a particular module. There is no need to include any of the original data.

At run-time, the virtual machine recognizes a frozen module by its magic number, causing it to divert control to the `loadfrozen` function which looks up the address of the frozen data using the path encoded in the file. If frozen data is found, it is processed in much the same way as regular module data, except that the address of the code in flash memory is used unchanged and the `frozen` flag is set in the `Module` object.

The build system for the STM32 port was updated with `mk` rules to run the `freeze` tool and build the generated code. The kernel's `main` function was updated to call a function to register the frozen modules with the virtual machine. The modules to be frozen are specified in the root section of the configuration file, using sources that end with the `.fdis` suffix, as in these examples:

```
/dis/env.dis     /dis/env.fdis
/dis/ls.dis      /dis/tiny/ls.fdis
```

These intermediate files are created then processed by the `mkroot` script as normal.

Issues with freezing code and other data

The virtual machine assumes that the resources held by modules are allocated at run-time, so issues can occur when these resources are no longer needed. The most obvious place where a change was needed is in the `freemod` function which was modified to only free sequences of instructions for non-frozen modules. However, problems still occurred when the `ps` command was run. This was tracked down to the `devprog` device that tries to obtain the size of a running program, with assumptions made in the allocator's `poolmsize` function about where the instructions are stored. Checks for the module's `frozen` flag were used to work around issues like these.

Experiments with freezing other sections of modules were only partially successful. As with frozen code, problems occur when data structures created by the loader are assumed to be managed by the allocator. An analysis of how each section is used by the virtual machine is needed to ensure that there are no surprises at run-time. Some sections are less amenable to freezing than others, with potential savings being only slight for those sections that contain only small amounts of data. A minimal approach that only freezes instructions offers most of the benefits of the technique with relatively few changes to the surrounding virtual machine.

Summary of memory savings

The use of frozen modules results in two main ways in which memory usage is reduced. The obvious one is the lack of a need to allocate memory for instructions at run-time. Additional savings are made by the use of placeholder files to represent frozen modules instead of including full-size module files in the root file system, leading to smaller allocations when reading them.

A side effect of the use of placeholder files is the reduction in the size of the root file system itself. Since this data is copied into RAM in the STM32 port, moving the contents of modules into the text section of the kernel means that only references to them in the root file system use up RAM. This means that using frozen modules also reduces the initial RAM requirements of the system. A different approach to handling immutable data would lead to similar benefits for all kinds of data, making this particular benefit obsolete.

Conclusion and future work

The work described above shows how a conceptually simple approach, supported by small changes to the Limbo module loader in the virtual machine, can reduce the size of memory allocations at run-time and reduce the overall memory footprint of an embedded Inferno system.

Further improvements could be made by applying the same approach to other sections of modules, where applicable, noting that further memory savings may be smaller than those already obtained.

It may be possible to extend the technique to compile virtual machine code to native machine code ahead of time. This approach may work well for devices with slow processors and reasonable amounts of flash memory, and could be used as an alternative to just-in-time (JIT) compilation for frequently used modules.

The source code associated with this paper can be found in the following repository:
https://github.com/dboddie/inferno-os/tree/stm32f405

References

1. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, P. Winterbottom, "The Inferno Operating System", *Bell Labs Technical Journal*, Vol. 2, No. 1, Winter 1997, pp. 5-18
   https://www.vitanuova.com/inferno/papers/bltj.pdf
2. "Lucent brews Inferno", *InfoWorld*, October 1996
   https://www.infoworld.com/article/2077261/lucent-brews-inferno.html
3. Vita Nuova, "Inferno Ports: Hosted and Native", *Vita Nuova Limited*, 27 April 2005
   https://www.vitanuova.com/inferno/papers/port.pdf
4. Salva Peiró, "Inferno DS: Inferno port to the Nintendo DS", *3rd International Workshop on Plan 9*, University of Thessaly, Volos, Greece, 2008
   http://3e.iwp9.org/iwp9_proceedings08.pdf

5. Brian L. Stuart, "Inferno in Embedded Space: Porting to the Sun SPOT", *8th International Workshop on Plan 9*, Athens, Georgia, USA, 2013
http://8e.iwp9.org/sunspot.pdf

6. Petter Duus Berven, "Porting Inferno OS to ARMv7-M and Cortex-M7", *Master's thesis in Computer Science*, Norwegian University of Science and Technology (NTNU), January 2022
https://hdl.handle.net/11250/3034870

7. "STM32F405xx STM32F407xx Datasheet", *STMicroelectronics*, 2020
https://www.st.com/en/microcontrollers-microprocessors/stm32f405rg.html

8. David Boddie, "STM32F405 Port of Inferno", accessed on 8 February 2023
https://github.com/dboddie/inferno-os/tree/stm32f405

9. Dennis M. Ritchie, "The Limbo Programming Language", *Inferno Programmer's Manual, Volume Two, Vita Nuova*, 2005
https://www.vitanuova.com/inferno/papers/limbo.pdf

10. Lucent Technologies Inc, Vita Nuova Limited, "Dis Virtual Machine Specification", *Lucent Technologies*, 1999, *Vita Nuova Limited*, 2003
https://www.vitanuova.com/inferno/papers/dis.pdf

An $O(1)$ Method for Storage Snapshots

Brian L. Stuart

ABSTRACT

The capability of taking snapshots is approaching ubiquity as a feature of file systems and data storage arrays. Here, we present an approach to structuring and managing snapshots in a storage space that provides for rapid creation and roll-back. This approach has been realized in the form of a Plan 9 kernel device that can be interposed between any pair of storage service and application. The kernel device has, in turn, been used to support an experimental file system. In this paper, we discuss the basic snapshot model, its implementation, and its application to a file system. Finally, we consider extensions to the model for supporting the deletion of snapshots.

1. Introduction

The ability of a file system or storage array to take snapshots has become de rigueur in the modern data storage world. As we discuss in Section 2., many of the recently developed file systems include snapshot capabilities as integral parts of their designs. Similarly, many Storage Area Network (SAN) products also include snapshot features as part of the block storage services they provide.

For the purposes of this work, we define a snapshot by the following characteristics:

1. Snapshots are immutable. We consider only the case of a snapshot which is preserved in time. In particular, clones (views of a data store which allow further modification) are outside the scope of this work.

2. Snapshots preserve the state of a data store at a particular moment in time. To view the state of a specific file as it existed at a specific point in history, the file as it exists in any snapshot taken between the specified point in time and the time of the next modification of the file will give that view. Collectively, a series of snapshots show a sampled view of the evolution of a data store over time.

3. Snapshots do not interfere with future use of the data store. Although the contents of a file are preserved by the act of taking a snapshot, the snapshot does not freeze the file for all time. Further modifications of the file can be made in the data store itself independent of the snapshot of its earlier state.

4. Snapshots can be accessed through the same mechanisms as the data store itself. This characteristic of snapshots distinguishes them from more traditional backups in which the data store is represented in a format that is in some ways different from the original data store preventing the normal file access tools and function calls from being able to act on the backup. Although the internal details of data representation may or may not be different in snapshots, we expect the implementation to provide the same interface as that of the primary data store. For data stores accessible via a network protocol such as 9P or NFS, we expect that the protocol and the implementation provide a way for a client to specify that it wishes to view the set of snapshots (or perhaps a particular snapshot).

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

There are two fundamental techniques whereby snapshots are usually implemented. The first of these techniques is block copying. As with a backup, a snapshot is taken by copying the blocks of the data store to a separate medium where the snapshots are stored. Normally, as an optimization, only the blocks (or files) that have changed since the last snapshot are copied. Of course, when using such an optimization, some form of indexing distinct from that used in the active store must indicate the location of files and blocks that are carried over from one snapshot to the next. This approach is particularly useful when the nature of the storage medium for snapshots is substantially different from that of the active data store. For example, the snapshots might reside on spinning disks where the active data store resides on solid-state flash memory devices.

The second major technique for implementing snapshots is copy-on-write (COW). Using this technique, a snapshot is taken by marking all currently writable blocks as copy-on-write. Later, when an attempt is made to write to such a block, a copy is made and the write is carried out on the copy. As with the block copying technique, indexing in the data store must identify which copy of a block is being referenced in a particular snapshot.

As detailed in Section 3., the present approach to snapshots is based on a COW mechanism where the physical location of a block determines its COW status. Specifically, this approach implements write once, read many (WORM)-like behavior such that blocks older than the most recent snapshot are frozen in time. In this way, we can implicitly and rapidly mark a large number of blocks as needing to be copied on the next write. In Sections 4.–7., we examine various considerations regarding the realization and application of this technique. The WORM-like characteristic of this approach isn't, however, immediately amenable to deleting snapshots. Section 8. discusses some potential methods of supporting snapshot deletion within the present snapshot technique.

2. Related Work

As mentioned earlier, snapshots in a file system or a storage space have become quite common. In this section, we take a look at several examples that are relevant to the present work. The particular examples chosen here are intended to be representative, rather than exhaustive.

2.1. Fossil/Venti and KenFS

Thompson's Plan 9 file server [5, 13] includes the ability to dump the state of the file system to WORM storage on demand or on an automated schedule. Later Plan 9 work includes the venti [6, 7] storage server which provides WORM-like behavior. It is most often used by fossil [8] for dump snapshots. The details of how our file system [11] presents its snapshots is modeled on that of the original Plan 9 file system and subsequently by fossil using venti. In both of these cases, snapshots are stored on what is conceptually (and in the earliest incarnations of the Plan 9 file system, literally) WORM media. The effect of "carving snapshots in stone" is a characteristic shared by the snapshots in the present work.

To avoid potential confusion, we point out that fossil supports two types of snapshots: ephemeral and archival. The mechanism for ephemeral snapshots is similar to that discussed in the present work, but the resulting snapshots are not persistent and their space is reclaimed in time. Fossil's archival snapshots are implemented by writing all active blocks to venti, providing the same persistence as the techique discussed here. Thus for purposes of comparison, one should read all references to snapshots in fossil as referring to the archival operation of block-wise copies to venti.

Both of the traditional Plan 9 file systems maintain caches where writes take place. When a snapshot is taken, the blocks in the cache that are to be part of the snapshot are scheduled for writing to a separate snapshot storage area. At the highest level of abstraction, this is very much like the active region distinct from the snapshots we discuss in Section 3.. There is, however, a fundamental difference of design. In the former cases, the active area is in a fixed location on potentially different media from the snapshots. In the case of the present system, the active region and the snapshots share common media, and the active area moves across the media like a sliding window. Although we continue to

investigate the relative merits of the two architectures, one benefit the present work does give is very rapid snapshot and roll-back operations.

2.2. Ext3cow

The Linux operating system has also provided a platform on which a number of storage research projects have been based. Among these is ext3cow [4] which implements a versioning file system based on similar copy-on-write techniques to those we use here. Because all file system updates in ext3cow induce copies, it effectively creates a snapshot on each write with one second resolution. Although the present work can be used for fine-grained snapshots, we have focused on applications where snapshot granularity on the order of one day is appropriate, and intra-day versions are unnecessary.

Ext3cow also illustrates a different approach to making snapshots available in the name space. Where our file system that uses the snapshots described here makes them available as individual hierarchical trees, ext3cow allows each component of a path name to be modified with version information. This difference is directly related to the granularity and intended application of the snapshots in the two different systems.

2.3. BTRFS

BTRFS [10] is one of the more recent major file systems developed for Linux. It is based on data extents arranged as B-trees with copy-on-write semantics [9] and uses reference counts to determine when extents can be reused. Although it uses COW for all updates (unless the nocow option is set), it does not keep all versions as ext3cow does. Instead, BTRFS uses a snapshot mechanism that creates a copy of the B-tree root, thus increasing the reference count on all nodes in the tree. BTRFS avoids walking the entire tree updating reference counts by deferring the reference count updates beyond a single generation of descendants in the tree. The effect of copying the root in this way is the creation of a clone, allowing updates to both "copies" of the tree.

The core technique of creating a snapshot by making a copy of a single metadata structure is similar to the techniques we report here. However, the details of our approach differ from those of BTRFS in a number of respects. For simplicity, we have chosen to use a large fixed-size block as the unit of COW rather than extents. Furthermore, we only perform a COW on those blocks that have not changed since the last snapshot following much the same policy as BTRFS does with the nocow option set.

2.4. ZFS

The SUN file system, ZFS, also provides a snapshotting facility [12]. At a superficial level, the relationship between snapshot support in ZFS and the approach described here is much like a combination the relationships described in this section. By that, we refer to the design of snapshots in ZFS as part of a file system and that clones can be created and snapshots deleted. Like several of the systems discussed in this section and the work reported here, ZFS uses a copy on write approach to implementing snapshots and clones. However, unlike the traditional Plan 9 file systems and the file system enhancements described here, ZFS doesn't make the set of snapshots collectively available in a unified name space.

2.5. Coraid VSX

For storage managed at the block level, the Coraid VSX product [1] includes a number of snapshotting features. Indeed, it provides a more rich set of snapshotting features than the system described in this report. In particular, it directly implements snapshot removal and the creation of clones (writable snapshots).

The basic technique of treating all pre-snapshot blocks as copy-on-write and using the new copy as the active block is modeled directly on the VSX approach. However, because the VSX implementation directly supports removal and clones and because it is integrated with the other functions of the VSX,

the process for creating a snapshot is relatively complex. As a result, its metadata updating as part of the snapshotting process is a $O(n)$ operation.

The divergent objectives between the two efforts are also apparent in the architectural approaches taken. Whereas the VSX implementation of snapshotting is aimed at providing a subset of the features of an integrated product, the present work is aimed at providing a snapshotting component that can be used in a number of contexts. It is this distinction that led to the implementation of this system as a kernel device, rather than integrating snapshot functionality into one or more applications.

2.6. NetApp WAFL

The Write Anywhere File Layout (WAFL) from NetApp also provides snapshotting based on copy on write. In a 2008 blog [3], the original developer of WAFL, Dave Hitz, stated "My current view is that WAFL *contains* a filesystem, multiple filesystems actually, but that's different from *being* a filesystem." This conclusion is based on the observation that WAFL has a top half which handles traditional file system functions, and a bottom half which handles disk and data management. The snapshot facility is described as being part of the bottom half. That certainly sounds similar to the architecture described here where the file system runs in a snapshotted space provided by devsnap. However, if we look deeper, we find that they are quite different. In [2], WAFL snapshots are described as being implemented by copying the root i-node. Furthermore, we find that the representations of a block's status within the file system and its status within the storage pool are directly coupled. Finally, all of the block management is based on physical block numbers, lacking the logical to physical block translation used in the system described here.

3. Snapshot Design

In the course of investigating how best to provide snapshotting in an experimental file system to be reported in a Coraid technical report [11], we have developed a technique and a mechanism for snapshotting the underlying storage space. The technique applies a COW approach to the raw blocks in a data store, residing in a layer underneath any file system. Thus, snapshots for a file system or a block storage structure fall out almost for free. In this paper, we report the details of the snapshotting design and implementation.

The core mechanism for this approach to snapshotting has its genesis in a question on an operating systems qualifying examination. The problem asked how one might approach supporting a UNIX-like file system directly on WORM media. One approach to this problem is the use of a block forwarding table. When a new block is allocated from the free list and written, no action outside of the normal operations is taken. However, when an existing block is modified, a new block is allocated from the free list and the modified block is written there. In conjunction with that, the original block's entry in the forwarding table is changed to point to the newly allocated block. We note that this technique implements the COW semantics. When reading a block, we consult the forwarding table and follow the chain of forwarding pointers until reaching one that is not forwarded. It is in that block where the current contents can be found.

We adapt this idea for implementing snapshots by recasting the forwarding table as a logical block number to physical block number translation table and use it to implement the COW semantics. The translation table here, denoted $\Pi : l \mapsto p$, differs from the WORM block forwarding table in two key respects. First, free blocks are not allocated from physical space, but from a logical space of blocks. This means that on the first allocation of a block an entry must be added to Π whereas in the forwarding table newly allocated blocks are left identified as unforwarded. Second, we assume that we are working with re-writable media. Thus prior to freezing by a snapshot, blocks and entries in the translation table can be modified multiple times. However, once frozen by a snapshot, blocks acquire COW status.

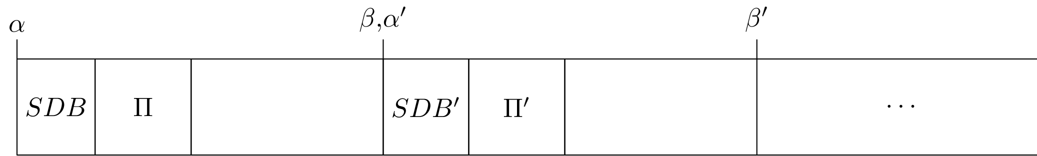


Figure 1: Version 1 Snapshot Structure

3.1. Snapshot Structure: Version 1

With this structure in place, we can see that a snapshot can be described uniquely with a translation table and a small amount of additional descriptive information. We place this additional information in a Snapshot Descriptor Block (SDB) which sits just before the snapshot's translation table. A snapshot is created, then, by writing an appropriately initialized SDB and copying the currently active Π table. The new SDB and table become the active ones and new blocks are allocated after the table. The effect is that snapshots are laid out sequentially in the storage media with the last one always being the active data store. This layout is illustrated in Figure 1.

The SDB stores several items. The value n gives the size of Π in blocks, α is the block number of the SDB, and β is the first free block in an active partition (as well as the SDB block number of a subsequent snapshot). Independent of β, the SDB also contains a pointer to the next SDB which is nil in the currently active data store. It also contains a textual string naming the snapshot.

The translation table, Π, is a simple array of block pointers, indexed by logical block number. Unallocated logical blocks are identified by a nil block pointer. Only the Π table in the currently active region can be written, because of the immutability of snapshots. Entries that point to physical blocks located to the left of the currently active SDB refer to blocks that are part of snapshots, which implicitly possess the COW property. Those that point to physical blocks to the right of the currently active Π table refer to blocks that are mutable, having been allocated or copied since the most recent snapshot was taken.

3.2. Snapshot Operation: Version 1

We now turn to the operation of the snapshot system. This section focuses on the four primary operations of reading blocks of data, writing blocks of data, creating a new snapshot, and reverting to an earlier snapshot. Throughout this discussion, the notation Π_l refers to the lth entry of the translation table Π. In other words, it is the physical block number where logical block l is stored in the relevant snapshot. Similarly, the notation Σ_p refers to the contents of physical block p in the storage media. If physical block p is part of a Π table, then the ith entry in that block of the table is identified by $(\Sigma_p)_i$. Finally, $\Sigma_{[n,m)}$ refers to the contents of blocks n through (but not including) m. The discussion is mostly structured in terms of single blocks, but support for partial blocks and multiple blocks is added in the obvious way.

With the structure described above, it is clear that we do not need much of the block meta-data that would normally be present. For example, from the perspective of snapshot handling, we know that a block is free if and only if it's physical block number is greater than or equal to β. Similarly, we know that a block has the COW status if and only if its physical block number is less than α. These observations make the following algorithms substantially simpler than they would otherwise be.

3.2.1. Block Reads

Reads are the simplest operations we perform in the snapshot system. Conceptually, we simply look up the block we want in the relevant translation table and then perform the read from the physical block identified there. More formally, we can describe the operation as in Algorithm 1.

ALGORITHM 1: Block Read

Input: Logical block number l to read
Output: Physical block contents
$p = \Pi_l$, where Π is the translation table for the snapshot
if $p = \Lambda$ **then** logical block l never allocated
    return 0 Block
else
    return Σ_p
end

Note that because contiguous logical blocks may be mapped to non-contiguous physical blocks, requests that span logical block boundaries must be broken down and processed as multiple operations, each fitting within one logical block.

3.2.2. Block Writes

As one might guess, writes are a bit more complicated than reads. This is because they require allocating blocks and copying when necessary. The algorithm for a write works as in Algorithm 2.

ALGORITHM 2: Block Write

Input: Logical block number l to write and data D
$p = \Pi_l$, Π must be the translation table for the currently active store, because snapshots are immutable
if $p = \Lambda$ **then** writing a block never yet written
    $\Pi_l \leftarrow \beta$
    Store D into block β
    $\beta \leftarrow \beta + 1$
end
else if $p \geq \alpha$ **then** writing a block already copied since last snapshot
    Store D into block p
else need a COW
    $\Pi_l \leftarrow \beta$
    $\Sigma_\beta \leftarrow \Sigma_p$
    Store D into block β
    $\beta \leftarrow \beta + 1$
end

Although we have accounted for all values of p, note that not all values of p are possible. If $\alpha \leq p < \alpha + n + 1$, then we have a logical block mapped to a physical block where an SDB or a Π block is stored. This is not possible in a correct implementation of these algorithms.

3.2.3. Taking a Snapshot

As illustrated in Algorthm 3 the act of taking a snapshot is relatively simple. We need only close the currently active region as a snapshot and open a new active region by making a copy of the current SDB and Π table. One effect of this is resetting α to the old value of β. This, in turn, implicitly marks all allocated blocks as COW. Thus we mark them all in a single step, rather than having to go through a table of blocks, marking each one.

Note that these operations are all $O(1)$ with respect to the number of dirty, unique, or allocated blocks. Thus the time to take a snapshot does not depend on the number of dirty or unique blocks in any earlier snapshot or the currently active store.

ALGORITHM 3: Taking a Snapshot

$\Sigma_{[\beta,\beta+n+1)} \leftarrow \Sigma_{[\alpha,\alpha+n+1)}$, Copy the SDB and Π table
Update the name of the old SDB and set its next pointer to β
Update the new SDB as: $\alpha' \leftarrow \beta$ and $\beta' \leftarrow \alpha' + n + 1$

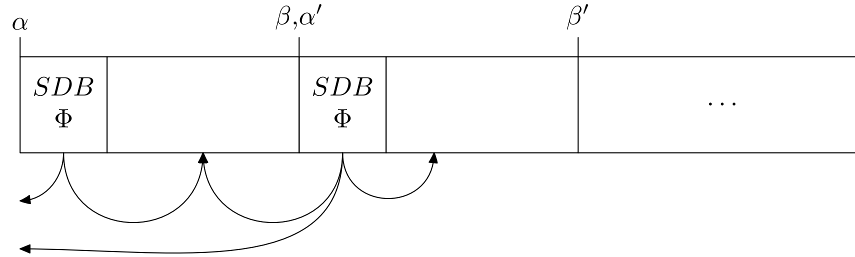


Figure 2: Version 2 Snapshot Structure

3.2.4. Reverting

The effect of reverting to Snapshot x is the removal of all changes made since x was taken and the resetting of x as the active region. This also results in x and all subsequent snapshots being removed. Implementing reverting is simply a matter of relabeling the SDB of x as the active region and as no longer pointing to any subsequent snapshots.

3.3. Snapshot Structure: Version 2

The initial prototype of this snapshot mechanism followed the structure and operation outlined in the previous subsections. However, it was noted that there are certain inefficiencies in that approach. Although the number of blocks copied in a snapshot operation (and, by extension, the number of blocks of overhead consumed by a snapshot) are constant with respect to the number of modified blocks, it is proportional to the overall size of the storage space. Furthermore, the subset of Π that changes from one snapshot to the next is relatively small. Therefore, a large subset of Π is redundant from snapshot to snapshot.

This deficiency is addressed in a second version of the basic design. In particular, a second level of indirection is added, distinguishing between logical and physical block numbers of the blocks that make up the translation table Π. The same COW strategy we use for data block also applies to the blocks making up Π. For the purposes of notation, we refer to this second-level translation table as Φ. With a suitable choice of block size, we can ensure that Φ is fully contained within the previously unused portion of the SDB. This, in turn, means that the snapshot procedure involves writing a single block, and that the storage overhead is a single block per snapshot. This revised structure is illustrated in Figure 2. The arrows leaving the SDBs in the figure are pointers in Φ identifying the physical blocks that make up Π. Those pointing to the left are entries of Φ that are inherited from the previous snapshot. Those pointing to the right are entries for blocks of Π that are either newly allocated or that are modified since the last snapshot was taken.

3.4. Snapshot Operation: Version 2

As expected, the details of the specific operations are a little more involved with this revised structure. However, as we see in this section, they are straightforward extensions of the previous algorithms. In the following discussion, B denotes the number of block numbers that are stored in a block.

3.4.1. Block Reads

The only thing different for reads in the new structure is the need for a two-step block number translation. This is illustrated in Algorithm 4

ALGORITHM 4: Block Read

Input: Logical block number l to read
Output: Physical block contents
$i = \lfloor \frac{l}{B} \rfloor$
$j = l \bmod B$
$m = \Phi_i$
if $m = \Lambda$ **then** unallocated block of Π
  return 0 Block
else
  $p = (\Sigma_m)_j$
  if $p = \Lambda$ **then** logical block l never allocated
    return 0 Block
  else
    return Σ_p
  end
end

3.4.2. Block Writes

Similarly, block writes require a two-step translation. Because not all blocks of Π are necessarily allocated, we might have to allocate a block for the translation table as well as for the data. We also note that a write operation might result in two COW operations. If the most recent version of the data block itself is in a frozen snapshot, then it is copied. Also if this is the first block mapped in a particular Π block since the last snapshot was taken, then the Π block must also be copied. Algorithm 5 shows the details of this.

3.4.3. Taking a Snapshot

Although the revised snapshot algorithm as shown in Algorithm 6 looks nearly identical to the first version, it involves copying only a single block as opposed to $n + 1$ blocks.

3.4.4. Reverting

Reverting to a previous snapshot is not changed by these revisions in the structure of the translation table.

4. Snapshot Kernel Device

This design for storage snapshots has been realized in the form of a Plan 9 kernel device. The structure and design of Plan 9 makes the inclusion of this type of mechanism particularly straightforward. Because all storage space servers present the spaces as simple data files and all clients expect to work through data files, we can construct a simple kernel device that both uses and presents the same data file interface. Such a device can then be interposed between any client and storage space, providing snapshotting for any storage application. Furthermore, this approach makes it easy for the snapshotting device to present all snapshots as data files too, allowing the clients to interact with snapshots in the same way that they interact with the active store. In the remainder of this section, we discuss the kernel device we

ALGORITHM 5: Block Write

Input: Logical block number l to write and data D
$i = \lfloor \frac{l}{B} \rfloor$
$j = l \bmod B$
$m = \Phi_i$
if $m = \Lambda$ **then** unallocated block of Π
    $\Phi_i \leftarrow \beta$
    $(\Sigma_\beta)_j \leftarrow \beta + 1$
    Store D into block $\beta + 1$
    $\beta \leftarrow \beta + 2$
end
else
    $p = (\Sigma_m)_j$
    if $p \geq \alpha$ **then** writing a block already copied since last snapshot
        Store D into block p
    else p is either never written or needs a COW
        if $m < \alpha$ **then** need to COW the relevant block of Π
            $\Phi_i \leftarrow \beta$
            $\Sigma_\beta \leftarrow \Sigma_m$
            $k = \beta$
            $\beta \leftarrow \beta + 1$
        end
        else
            $k = m$
        if $p \neq \Lambda$ **then** needs a COW
            $\Sigma_\beta \leftarrow \Sigma_p$
        $(\Sigma_k)_j \leftarrow \beta$
        Store D into block β
        $\beta \leftarrow \beta + 1$
    end
end

have implemented for this. Because the current code implements Version 2 of the design, the following discussion is based on that version.

The source file for the driver is devsnap.c and it uses the device letter, $\mathbb{P}$. (Plan 9 supports the full Unicode character set for device identifiers.) The current prototype version is less than 700 lines of code. When the system first comes up, devsnap presents a single file in its name space. The file, `ctl`, serves the conventional role of providing a control and status interface for the snap device. There are five commands that can be written to the control file:

bind The command, `bind`, is used to attach an existing snapshotted store to the snap device. It takes a single argument which is the data file representing the storage space where snapshots are stored. E.g.

```
bind '#ℙ' /n/snap
echo bind '#S/sdE1/data' > /n/snap/ctl
```

Devsnap scans the data file and creates a data file for each snapshot (including the active store) it finds there.

ream The `ream` command is used to initially format a storage space where snapshots are to be managed.

ALGORITHM 6: Taking a Snapshot

$\Sigma_\beta \leftarrow \Sigma_\alpha$, Copy the SDB and Φ table
Update the name of the old SDB and set its next pointer to β
Update the new SDB as: $\alpha' \leftarrow \beta$ and $\beta' \leftarrow \alpha' + 1$

It takes two arguments, a name given to the active store and the path name of the data file where the snapshots are to be managed. E.g.

```
echo ream fs '#S/sdE1/data' > /n/snap/ctl
```

Reaming is implemented by simply writing the first SDB and an empty Φ table at the beginning of the storage space.

snap Taking a snapshot is triggered by the `snap` command which takes two arguments. The first argument is the name of the active store to be snapshotted, and the second argument is the name to be given to the newly created snapshot. E.g.

```
echo snap fs fs.20131119 > /n/snap/ctl
```

In response to this command, devsnap applies Algorithm 6 given in Section 3.4.3..

unbind The `unbind` command takes a single argument which is the path name for the storage space data file previously connected with a `ream` or `bind` command. E.g.

```
echo unbind '#S/sdE1/data' > /n/snap/ctl
```

The effect of an `unbind` is the closing of the connection to the underlying device and the removal of the associated files from devsnap's name space.

revert As with taking a snapshot, reverting takes two arguments, the name of the active region and the name of the snapshot to which we should revert. E.g.

```
echo revert fs fs.20131119 > /n/snap/ctl
```

The effect of this command is to roll back the store to the same state it was in at the time that Snapshot `fs.20131119` was taken.

Reads of the `ctl` file return a list of the snapshots and active stores currently known to devsnap. E.g.

```
0 fs #S/sdE1/data 1 10720 10849 10863
1 fs.20131119 #S/sdE1/data 3 10683 0 10683
2 fs.201311191 #S/sdE1/data 3 10720 10683 10849
```

The items on each line are: the internal slot number, the name of the snapshot, the path name to the underlying data store, the internal flags, the number of blocks in use, the value of α, and the value of β.

For each snapshot, devsnap provides a data file in its name space. E.g.

```
--rw-rw-r-- ℙ ...             0 ... ctl
--rw-rw-r-- ℙ ... 1000204886016 ... fs
```

```
--r--r--r-- ℙ ...   11201937408 ... fs.20131119
--r--r--r-- ℙ ...   11240734720 ... fs.201311191
```

Note that the snapshots are given read-only permissions, and the active store is given read-write permissions. Also the sizes differ among the data files. The size of the active store is the size of the underlying data file. The size of a snapshot is the number of bytes used by that snapshot.

In devsnap, we have chosen to use a block size of 1MB and to specify logical and physical block numbers as 64-bit integers. It should be noted that devsnap's block size is only used as the unit of allocation and COW and in no way constrains the block sizes used either by clients or by the underlying storage. We do, however, assume the ability to write 512-byte sectors to the underlying storage. This happens when updating a value in the Π table. Instead of writing the whole 1MB block, we write only the 512-byte sector containing the change. I/O as a result of client read/write requests is carried out in the units requested by the client. Thus the behavior continues to be the same as it would be without the addition of devsnap.

The parameters we have chosen for devsnap mean that each block in the Π table contains 128K entries. Since each block is 1MB, a block of the Π table describes 128GB of logical space. Because very little of the SDB is used for descriptive information, almost all of it is available for the Φ table. Thus a two-level table with 1MB blocks can manage about 16PB. To support a larger storage space, we can use 4MB blocks which allow for up to 1EB in total space. Similarly, a three level table can be used which would support up to 2ZB with 1MB blocks. As well as the currently active SDB and Φ table, we keep a one block cache for the Π table for each snapshot, allocated when we first access a snapshot. Therefore, as long as jumps between 128GB regions of the logical space are relatively rare, the snap device introduces very little overhead.

5. Boot Modifications

To assist in booting from a storage space managed by devsnap, we have augmented /sys/src/9/boot/local.c. If the bootargs string or the string entered at the "root is from" prompt begins with "#ℙ" indicating that the root is to be taken from devsnap, the second token, if any, indicates the data file to bind to devsnap. For example, if we enter the string:

```
local!#ℙ/fs #S/sdE1/data
```

at the "root is from" prompt, the boot code will issue the message

```
bind #S/sdE1/data
```

to devsnap before running the file system code. This would result in the system running with its root taken from a snapshotted file system stored on a device handled by devsd.

6. File System Support

Because each snapshot is provided in its own data file, it would be relatively easy to add snapshot support to any file system by pointing the file system in a read-only mode at the appropriate snapshot data file. However, this would become quite cumbersome as the number of snapshots grew, particularly with respect to any attempts to export the set of snapshots collectively via 9P or NFS. Therefore, when adding support for snapshotting to an experimental file system, we chose instead to give the file system knowledge of the set of snapshot files and let it present them in a useful way.

Our support for snapshots takes advantage of support for the creation of multiple roots similar to that supported by ZFS. (Alternatively, these can be thought of as multiple independent file systems sharing the same storage space.) Access to them is specified by the attach name in 9P and by the mount path name in NFS.

For snapshot use, we create a root called /dump which provides semantics compatible with existing Plan 9 history tools. Contained within this root directory are a set of directories, each named by the year for which at least one snapshot was taken. Inside the year directories are directories named by the date on which the snapshot was taken. Each entry is of the form $mmdds$ where mm is the month, dd is the day of the month, and s is a sequence number which is nil on the first snapshot of the day and the numbers 1, 2, 3,... for subsequent snapshots. What makes these entries special is the presence of a meta-datum named snap. The value of the snap meta-datum is a string giving the name of the data file for that snapshot.

Continuing with the example we have been showing, we could access the snapshots as follows:

```
term% 9fs dump
term% lc /n/dump
2014
term% ls -l /n/dump/2014
d-rwxrwxr-x M 42 bootes bootes 0 Feb 12  2014 /n/dump/2014/0212
d-rwxrwxr-x M 42 bootes bootes 0 Feb 12  2014 /n/dump/2014/02121
d-rwxrwxr-x M 42 bootes bootes 0 Feb 13  2014 /n/dump/2014/0213
d-rwxrwxr-x M 42 bootes bootes 0 Feb 14  2014 /n/dump/2014/0214
...
term% history papers/acm/snapshot.tex
Aug 28 14:38:50 EDT 2014 papers/acm/snapshot.tex 44208 [stuart]
Aug 27 21:57:22 EDT 2014 /n/dump/2014/0827/.../snapshot.tex 46048 [stuart]
Aug 26 22:02:12 EDT 2014 /n/dump/2014/0826/.../snapshot.tex 45365 [stuart]
Aug 25 17:54:02 EDT 2014 /n/dump/2014/0825/.../snapshot.tex 45258 [stuart]
Aug 24 21:09:27 EDT 2014 /n/dump/2014/0824/.../snapshot.tex 43448 [stuart]
Aug 23 00:25:37 EDT 2014 /n/dump/2014/0823/.../snapshot.tex 42991 [stuart]
Aug 22 23:03:47 EDT 2014 /n/dump/2014/0822/.../snapshot.tex 43035 [stuart]
Aug 21 16:52:40 EDT 2014 /n/dump/2014/0821/.../snapshot.tex 38902 [stuart]
term% ls -lp /n/dump/2014/0821/usr/stuart/papers/acm/snapshot.tex
--rw-rw-r-- M 42 stuart stuart 38902 Aug 21 16:52 snapshot.tex
term% ls -lp /n/dump/2014/0827/usr/stuart/papers/acm/snapshot.tex
--rw-rw-r-- M 42 stuart stuart 46048 Aug 27 21:57 snapshot.tex
```

where the ellipsis is used to shorten the path names for formatting purposes. When walking a directory tree, we walk as normal until we reach a node with a snap value. At that point, we open the snapshot data file and continue the walk starting from its default root. The net effect is much like a mounted file system.

Because the details of the directory traversal are very different between 9P and NFS, so too are the details of our snapshot "mounting." In the case of 9P, we keep track of which tree we're in by attaching an open file descriptor to the internal Fid structure. The file descriptor is reference counted and on a destroyfid call where we are dropping the last reference to it, we close the file. In the case of NFS, we keep track of where we are by extending the file handle passed back to the client for files in a snapshot tree. In particular, we append a unique identifier of the file in the /dump tree whose meta-data includes the relevant value for snap. In both cases, accesses to snapshots are read-only and uncached.

To support creating snapshots, a new command has been added to the console interface of the file system. Writing the command snap to the console causes it to command devsnap to create a snapshot. The name is based on the dump conventions discussed earlier. Specifically, if the active store name is x, then the dump will be named x.yyyymmdds. The snapshots shown in the examples of this report were all created by a periodic scheduled triggering of this mechanism.

Similarly the `revert` command has been added to support rolling the file system back to an earlier

snapshot. The command takes a single argument which identifies the snapshot to which we are rolling back. For convenience, it can be specified either in the form it appears in the snapshot name, yyyymmdds or in the partial path name seen in the dump file system, yyyy/mmdds.

7. Applications in Block and Object Storage

The snapshotting provided by this technique is not restricted to applications in file systems. Because it operates directly on an array of blocks and presents the same array of blocks, it can just as easily be used with a storage space used to provide block or object storage. However, as with 9P and NFS exports of the file system, we must establish some mechanism by which a client can gain access to the snapshot. Most SAN designs identify exported block and object storage spaces by logical unit numbers (LUNs). Such a system can export a snapshot by assigning a LUN to the snapshot but with read-only permissions.

We should point out one interesting effect of snapshotting LUNs in this way. Just as the implementation of devsnap as a kernel device creates a very general snapshotting mechanism within the system, applying it to LUNs creates a very general snapshotting mechanism for clients. Whether the client is running Linux, Solaris, vmware, or Windows, it will have access to snapshots of its native file systems as stored on a block storage appliance.

8. The Snapshot Deletion Problem

From the foregoing discussion in this report, it is clear that this approach to snapshotting is fundamentally based on a WORM-like model of storage. In particular, once a region has been closed by taking a snapshot and by starting a new active region, the space in the first region is never reused. Another way of viewing this is that we never write to any physical block whose block number is less than α in the active system. Although this characteristic of our approach substantially simplifies and streamlines many of the implementational details, it does impede any desire to remove snapshots. In this section, we consider some ways of providing snapshot removal.

8.1. Redacting Snapshots

We identify the two primary motivations for removing snapshots as data obsolescence and space recovery. In the case of data obsolescence, the intent is usually the removal of references to sensitive data. Merely removing the mapping to old snapshots is actually quite easy. If we copy the succeeding SDB and Φ table on top of the ones we want to remove, we lose any mappings to the blocks unique to the snapshot we are removing while maintaining references to other blocks. More formally, we copy $\Sigma_{\alpha_n} \leftarrow \Sigma_{\alpha_{n+1}}$. Although the references to those blocks are removed, the data blocks themselves remain intact and are therefore subject to forensic recovery. To make the data less susceptible to recovery, we would need to overwrite the blocks that are uniquely allocated to the snapshot being removed. This would, in turn, require the reference counting technique discussed in the next subsection, and could be piggy-backed on the space compaction discussed there.

8.2. Space Compaction

If our purpose in removing snapshots is recovering space not used in recent snapshots, then there are a couple of approaches we can use to reclaim those blocks no longer referenced after overwriting an SDB and Φ table. The first approach we consider is compaction, sometimes referred to as defragmentation. The basic idea is a simple one. We shift logical blocks that are in use down into the physical blocks that are freed by the snapshot removal. Compaction has the benefits of not requiring any additional mapping of block positions and of maintaining (and even increasing) locality of reference. The disadvantage of compaction, however, is the large amount of block copying that it incurs. Consequently, we lose the $O(1)$ snapshot behavior mentioned earlier.

The primary technique for knowing which blocks are freed in a snapshot removal is to maintain a reference count of each physical block. Each reference count is incremented on block allocation and snapshot creation, and it is decremented on COW and snapshot removal. Blocks whose reference counts are decremented to zero are now free. Because the reference counts are global to the whole storage space, only a single copy is kept avoiding adding more overhead to the snapshot regions.

8.3. Unbounded Space Simulation

The second approach we consider for reclaiming space from removed snapshots is simulating an unlimited storage space. The basic idea is that when we remove a snapshot, we logically tack all of the blocks unique to it to the end of the storage space expanding it by that number of blocks. Then as β grows and new snapshots are taken, we reuse the blocks that had been part of earlier snapshots.

As with the compaction approach in the previous subsection, we need to maintain reference counts to know which blocks are freed. Simulating the unbounded logical storage space can be done relatively simply by maintaining another logical-to-physical translation table. This table is a little more involved than the ones we use as part of the snapshots. The reason is that the logical space is unbounded and thus potentially much larger than the physical space to which it maps. Consequently, a simple table indexed by logical block number would not be practical. An effective alternative would be a hash table keyed on the logical block number.

9. Conclusion

In this report, we have examined a novel technique for providing storage snapshots. The technique is both simple and general, and it places very little in the way of demands on the storage applications using it. Our prototype implementation is less than 700 lines of code and supports both file-structured and block-structured storage space. It can be used with individual raw disks, RAID configurations, and network attached block storage. This technique has proven to be quite powerful as evidenced by the simplicity of including support for it in an existing file system. The bulk of the necessary changes are a few hundred lines that make the set of snapshots available in a single hierarchical name space. Finally, as suggested by the example in Section 6., the text editing and formatting of this paper were carried out on a file system operating on the snapshot mechanism described here.

10. Acknowledgements

We would like to express our appreciation to Coraid for its support during this research. In particular, Brantley Coile established the research environment in which the work was conducted. Erik Quanstrom provided a valuable sounding board and helpful questions and suggestions, particularly in the transition from Version 1 to Version 2 of the design discussed here.

References

[1] Coraid Inc. Ethernet san storage virtualization etherdrive vsx3500, 2010.

[2] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[3] David Hitz. Is wafl a filesystem? https://web.archive.org/web/20140715102135/https://-communities.netapp.com/community/netapp-blogs/dave/blog/2008/12/08/is-wafl-a-filesystem, archived on July 15, 2014, 2008.

[4] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Trans. Storage*, 1(2):190–212, May 2005.

[5] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[6] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. Technical report, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA, May 2002.

[7] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[8] Sean Quinlan, Jim McKie, and Russ Cox. Fossil: an archival file server. World-Wide Web document, 2003.

[9] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Trans. Storage*, 3(4):2:1–2:27, February 2008.

[10] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, August 2013.

[11] Brian L. Stuart. Toward unification of storage granularities. Coraid Technical Report, 2014.

[12] Sun Microsystems. ZFS On-Disk Specification. https://web.archive.org/web/20081230170058/-http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf, archived December 30, 2008, 2006.

[13] Ken Thompson. The Plan 9 file server. World-Wide Web document, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA, 2000.

Ghostscriptbusters

Noam Preil (paper author, past and future maintainer / tester / fixer, probably a person)
Sigrid (pdffs author, amazing 9fronter)

<noam@pixelhero.dev>

ABSTRACT

pdffs is a Plan 9-native, clean, and maintainable PDF reader and writer, which is almost ready for upstreaming. A few key features need to be implemented, and document compatibility testing is still needed, but pdffs is already showing great promise.

Motivation

The Plan 9 port of ghostscript [3] is old and was never truly integrated with the system. Updating it is not a good option; GhostScript upstream is notorious for security problems (Gentoo disables usage of GhostScript in web services by default, for this reason [4]), and is overengineered and overcomplicated. Integrating it with the system and, effectively, hard-forking it would require a monumental undertaking, as its design is unfriendly and it is over half a million lines of code.

Additionally, over the years, PostScript has slowly fallen by the wayside in favor of PDF. PDF is a deceptively simple format, and can be handled much more sanely than a glance at GhostScript's code base would suggest.

Lastly, there are some basic features which GhostScript doesn't provide, such as PDF searching, text and image extraction, links, selecting and copying text, which are trivial to implement on top of a cleaner code base.

GhostScript served us well for a long time, but the time has come to kill some ghosts. The PDF reader is dead, long live the PDF reader!

History

In the beginning, there was nothing. That was quite boring, so the nothing created PostScript, and lo! The nothing saw that it was good.

PostScript is a graphical programming language. Programs – documents – can set variables (such as fonts), render text and graphics to the screen, and all the other fancy-shmancy stuff a graphical program needs to be useful. PostScript has global context: important variables, such as fonts, all exist in the global scope. If a document sets the font size to 12 on page 3, and nothing changes the font size, all the following pages will inherit the font size. This has some unpleasant results, not least of which is that, to render any page, all previous pages must first be rendered. It's impossible to know the font used on page 1000 without rendering every page from 1 through 999 first!

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

Flash forward a few billion years, and nothing has changed. Eventually, Adobe decided that Something Must Be Done. They took PostScript, added page scoping, and structure. Then, of course, they decided that wasn't enough, and tossed in XML, JavaScript, and various other violations of the Geneva Convention and the Treaty of Versailles. Fortunately, most of the bad ideas have been deprecated, so pdffs doesn't need to worry about them.

PDF Design

The PDF specification [2] is over a thousand pages, but – at least as of PDF 2.0 – is a lot simpler than its length would suggest. Many of the features are optional [and, in practice, unused], and the format – while far from perfect – is *extremely good* compared to all of its competitors. For a full grasp of the PDF format itself, I highly recommend either looking at the PDF1.7 specification [2] – the last version which was publicly available – or finding a copy of the PDF2.0 final draft, which is probably identical to the finalized spec and remains legally available for free. The overview here is just meant to give enough background to understand pdffs.

PDFs are randomly accessed via various index structures stored in the document. PDFs are composed of objects of various types: booleans, integers, doubles, strings, arrays, dictionaries, and so on. PDFs also contain indirect objects, which are essentially pointers into the document. Since indirect objects point to unparsed data, objects can be parsed on access, which allows for only tracking the parts of the document that are in active usage – this is essential for random access.

PDFs store data in a typed, key–value tree, in a mix of raw binary data and Unicode text. The trailer at the end of the document contains pointers to the root of the main tree, an information tree, and a document ID. For instance, the trailer for the current draft of this document looks like this:

```
<<Size = 32, Root = @1[gen=0], Info = @2[gen=0], ID =
[<468E598CDE0B53AC84B1182089F83079>,
<468E598CDE0B53AC84B1182089F83079>]>>
```

pdffs indicates indirect references with an @ prefix. The root of the tree is at address 1, the information tree is at address 2, and so on. PDFs can also be edited, and PDF has special mechanisms for updating the document in-place without needing to fully reserialize: objects [and references!] have *generations,* so a document change which only affects one object can e.g. leave most objects at generation zero, append the changed object with generation one, and bump the generation in the pointers to reference the new object.

pdffs' object trawler can be used to descend the tree. Shown here is some quick exploration of the current draft of this document:

```
% pdffs /sys/doc/pdffs.pdf Info
<<Producer = 'AFPL Ghostscript 8.53', CreationDate =
D:20230213012906, ModDate = D:20230213012906>>

% pdffs /sys/doc/pdffs.pdf Root
<<Type = /Catalog, Pages = @3[gen=0]>>
% pdffs /sys/doc/pdffs.pdf Root Pages
<<Type = /Pages, Kids = [@4[gen=0], @16[gen=0], @20[gen=0]],
Count = 3>>
% pdffs /sys/doc/pdffs.pdf Root Pages Kids
[@4[gen=0], @16[gen=0], @20[gen=0]]
% pdffs /sys/doc/pdffs.pdf Root Pages Kids 0
```

```
<<Type = /Page, MediaBox = [0, 0, 612, 792], Rotate = 0,
Parent = @3[gen=0], Resources = <<ProcSet = [/PDF, /Text],
Font = @15[gen=0]>>, Contents = @5[gen=0]>>
% pdffs /sys/doc/pdffs.pdf Root Pages Kids 0 Contents
<<Length = @6[gen=0], Filter = /FlateDecode>>+stream
```

PDFs have a typed hierachy. Dictionaries in the tree always contain a Type field, which indicates what type of dictionary is present, and thus what other fields exist. For instance, Pages are a group of Page entries, and can also contain nested Pages. A 30-page document can be split into 6 groups of five, for instance, with the root Pages containing six Pages, with each group containing five Page entries. For documents generated by ps2pdf on 9front, the root of the pages catalog always contains one entry per page, and no subgroups.

A Pages dictionary, for instance, always has a Kids field, which indicates the children nodes, and a Count field, which says how many subpages it contains. A Page dictionary contains a Parent field, which is the Pages dictionary containing it, a Contents field, which is a stream of PostScript-ish instructions for rendering the page. There are also optional fields, with default values; a Page has a CropBox field which defaults to the value of the page's MediaBox field, for instance.

pdffs design

The core of pdffs is quite simple: find the trailer at the end of the PDF, which indicates where to find all of the key information about the document, and then expose a file system representing the document, parsing objects and dereferencing pointers as requested by the user. A trivial garbage collector can clear up resources when the user no longer needs them.

Since file systems and PDF documents are both hierarchical, it's quite easy to map the document to a file system tree. pdffs really just needs a few components: the document parser, and the 9p implementation to expose the data. Additionally, pdffs provides other convenience functions: there's an internal renderer, for instance, so that pages can be exposed in the file system as rendered images.

Status

The parser is basically complete. The renderer is able to position and draw text correctly, including colors and sizing. All text is rendered with the default Plan 9 font. Path support is missing, as is image rendering. pdffs has been tested against the /sys/doc papers, the Intel technical reference manuals for x86, the Rockchip SoC papers, and an assorted dozen other papers.

Most of the core functionality is basically complete, and just needs to be wired together. At the moment, the file system hierarchy is nonexistent, and resources are instead requested by specifying a path on the command line, as shown earlier. To render a page to an image, the pseudo-field ! can be used on a Page.

`pdffs /sys/doc/pdffs.pdf Root Pages Kids 0 !`, for instance, will render the first page of /sys/doc/pdffs.pdf to a Plan 9 image, and write the image to standard output. Piping the output into page allows for viewing a single page at a time.

There are existing prototypes for searching within a PDF, and converting PDFs to text, both function on top of the CLI, but need to be switched over to the fs. The pseudo-field " of a page renders a page to text, and writes it to stdout:

```
% pdffs /sys/doc/pdffs.pdf Root Pages Kids 0 " | read -c50
Gh ostb u stersNoam  Preil (paper author, current
```

Text conversion, at present, uses heuristics for determining where to split words, which *sometimes* works – but, as seen above, has more than a few flaws.

Improvements over GhostScript

Even in its current state, `pdffs` already has some advantages over `ghostscript`. `pdffs` is able to search through a PDF for text, for instance:

```
% ./pdfpages.rc
Usage: pdfpages.rc file.pdf 'search string'
% ./pdfpages.rc /sys/doc/pdffs.pdf PDF
1
2
3
4
% ./pdfpages.rc /sys/doc/pdffs.pdf 9front
1
3
```

At present, without the file system, this is done by looping through Page structures, rendering to text, and `grep` ping for the search string, reporting which page it's found on.

Additionally, compile times are much, much more reasonable, and not just because features are missing. All of the following metrics were taken from my T420s:

```
GhostScript:
37.17u 9.65s 21.67r   mk all
pdffs:
0.32u 0.23s 0.29r     mk all
```

Building the entire rest of 9front takes under 48 seconds. Replacing ghostscript with pdffs cuts down my overall build times from 70 seconds to 48 seconds.

GhostScript is *not* native code, and it shows. It contains its own parsers for the various image formats, for instance, and plenty of other code which is completely redundant with existing native tools. It's also largely overengineered, and its PDF support is implemented within PostScript, not C.

Lastly, but most importantly, pdffs is actually maintainable. While pdffs currently has fewer features than ghostscript, there are already documents that ghostscript fails to render that pdffs handles more elegantly.

`pdffs`[1] is currently just over five thousand lines of code. The ghostscript source included in 9front is almost 670,000 lines of code, more than one hundred times larger. GhostScript's PNG parser, alone, is almost 50,000 lines of code. By contrast, pdffs uses the same PNG reader as the rest of 9front, which is under 1K lines. Even counting that code as part of 9front – which is a little bit silly – would still make GhostScript's PNG parser nearly *ten times larger* than the entirety of pdffs!

Intended usage

Some of the primary motivation for pdffs is the ability to search for text in a document, and to snarf text from within documents. Other desirable features are navigation, indexing (via the table of contents, for instance), To facilitate this, a few endpoints are needed for pages in the document. With a PDF mounted at `/mnt/pdf` , here's an example of interaction from a terminal:

```
% cd /mnt/pdf
% ls
toc
pages
% ls toc
Prologue
% cat toc/Prologue
4
% ls pages/4/
contents
fonts
mediabox
rendered
region
text
% cat <pages/4/mediabox >pages/4/region
% sha1sum pages/4/region pages/4/text
d6e03e03d6770adcd29a0427dd88e16f5286816a  pages/4/region
d6e03e03d6770adcd29a0427dd88e16f5286816a  pages/4/text
```

When selecting a region of text to copy, we can simply write the region to pdffs and read back the text in the selected region. Searching is simple, as we can just look through the pages for one whose /text contains the string we're searching for. We can also convert PDFs to other formats easily:

```
% cd /mnt/pdf/pages
% mkdir /tmp/output
% for(page in *){
    topng $page/rendered >/tmp/output/$page.png
}
%
```

By exposing a PDF as a file system, and providing a few core tools, pdffs can provide a base on which to build all the PDF tooling we could ever need, in a very 9 manner, including but not limited to:

- Normal pdf viewer – page(1) integration: rendering, searching, links, table of contents, snarfing, ...

- Text-only PDF client – cat $page/text

- PDF conversion – $page/rendered as a Plan 9 image, existing tooling

- PDF editor – writable endpoints

- PDF to html conversion – $page/text, $page/fonts, ...

- PDF splitter – for each page, create a new PDF,

```
% echo new >/mnt/new/pages/ctl &&
        cp /mnt/old/pages/$page/all /mnt/new/pages/1/all
```

- PDF combiner –

```
% for(i in $SOURCES){
        mount /srv/pdffs /mnt/old $i
        for(page in /mnt/old/pages/*){
                echo new >/mnt/new/pages/ctl &&
                        cp $page/all /mnt/new/pages/latests/all
        }
}
```

Open questions

A few design questions remain.

How much work should be done in page, and how much in pdffs?

Once pdffs is finished, do we remove postscript – and replace all tools which generate postscript, such as dpost, with PDF output? Or, do we write a PostScript interpreter, too?

What APIs do we want to crystallize? Once an interface is exposed in the file system and used, it will be very hard to replace it.

Future work

There are a few critical features missing, most obviously the actual exposure of a file system. While a file system is, as indicated by the name, one of the core features of `pdffs`, it's actually the easiest part to add, and thus one of the final steps to completing pdffs.

Integration with page(1) is essential, and much of the driving motivation for pdffs' existence. This depends on the file system: we want to be able to e.g. write a rectangle from page to pdffs, and read back the text in the region – and, even more importantly, the ability to search within pdfs is essential.

Much more pressingly, the core logic and internal APIs need to be finalized, and the glue code finished. Font support is critical, for instance.

Then, all the major outstanding bugs need to be fixed. When all of /sys/doc can be rendered without issues, and some other documents – such as the Intel TRMs – are readable, it'll be time to add the 9p support and page(1) integration. At that point, we'll be ready to merge with 9front upstream, and start deprecating GhostScript. We'll need to, in stages, make pdffs the default – with gs available as a fallback and for postscript documents – and, at long last, finish busting the ghost.

Acknowledgements

While I wrote the current rendering code, and fixed bugs here and there, Sigrid wrote the majority of the infrastructure code pdffs uses, and deserves basically all of the credit for pdffs. All *I* did was write some matrix multiplication code, and lose twenty hours of my life to reading the PDF specification.

Thanks, as well, to a semi-sapient string of hexadecimal that would like to remain anonymous. I've only ever met one or two semi-sapient hexadecimals, though, so deanonymization is an ever-present threat.

References

[1] pdffs repository SourceHut, the hacker's forge `https://sr.ht/~ft/pdffs`

[2] Document Management – PDF1.7, ISO–32000 , Adobe Systems Inc™

[3] GhostScript, `gs(1)` , /sys/src/cmd/gs

[4] ImageMagick , Gentoo Wiki

Porting the Netsurf web browser to Plan 9

Jonas Amoson
jonas.amoson@hv.se

University West
461 86 Trollhättan, Sweden

ABSTRACT

Netsurf has recently been ported to Plan 9 (9front), bringing a more powerful browser to the Plan 9 users. This paper describes the steps done in the porting effort and the solutions considered for challenges encountered in the tool chain, the environment and the build system. Solutions fall in different categories, depending on where they can be implemented: in upstream Netsurf, in 9front or if they have to be maintained as patches outside the official Netsurf sources.

1. Introduction

The available web browsers (Mothra and Abaco) in Plan 9 and 9front are small and elegant but lack much of the functionality of the modern web, at the same time as full-featured open-source browsers such as Firefox are big, complex, and to a big part written in C++. It would therefore be interesting to port a more powerful, but still small, browser to Plan 9. Netsurf [1] is a relatively modest open-source web browser written in C, with support for many quite different platforms, among them legacy operating systems with limited hardware resources, such as RISCOS, Amiga OS and BeOS. Yet, Netsurf has good support for CSS and a growing support for JavaScript, which makes it a good candidate for porting. Netsurf has been in active development since 2002 with major releases every few years.

This paper describes the port [2] of Netsurf 3.10 to 9front, focusing on problems and selected solutions in the process of compiling the upstream Netsurf sources. The experience might be useful for porting other software to plan 9, or generally for porting software between platforms that differ a lot.

2. The porting process

The porting was started by first compiling all support libraries and platform independent code in Netsurf, using a mk-based build system, fixing the gaps until everything compiled and linked. This way of working was not clear from the start, and one idea was to compile the source on Linux, where it already works, and gradually add support for plan 9 graphics and input handling, and only thereafter move the source to Plan 9. It already from the beginning became clear that APE, the ANSI/POSIX environment [3] was needed to compile the upstream Netsurf code. It was also helpful to have the Linux port to compare with, both in building the source as well as in run-time debugging.

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

2.1. Framebuffer frontend

Netsurf supports native frontends for many different operating systems, as well as a generic ‘framebuffer’ frontend that draws on a memory bitmap and contains its own graphical components such as buttons, scroll bars as well as its own text rendering. The second step was to write a framebuffer driver (called surface) written especially for plan 9, based on the generic framebuffer code in Netsurf. It had to translate Plan 9 keyboard and mouse events for the Netsurf framebuffer library, and copy the bitmap image drawn by Netsurf onto a Plan 9 libdraw image, in a similar way as VNC† works. After some debugging, Netsurf was running on Plan 9.

2.2. Native frontend

The framebuffer version of Netsurf on Plan 9 had several drawbacks. It was rather slow, especially if running over a network, was lacking many features of the native frontends for other platforms supported by Netsurf and didn’t look and behave much like a Plan 9 program. A native frontend was then written from scratch by Philippe Mechai, with Plan 9 scrollbars, mouse context menus, plumber support and many more features. The work on the native frontend surpasses the initial porting effort, and the framebuffer frontend was discarded after some time to simplify the build system and have less code to maintain. The native frontend still lives in an APE environment. Figure 1 shows Netsurf displaying the official Netsurf webpage.

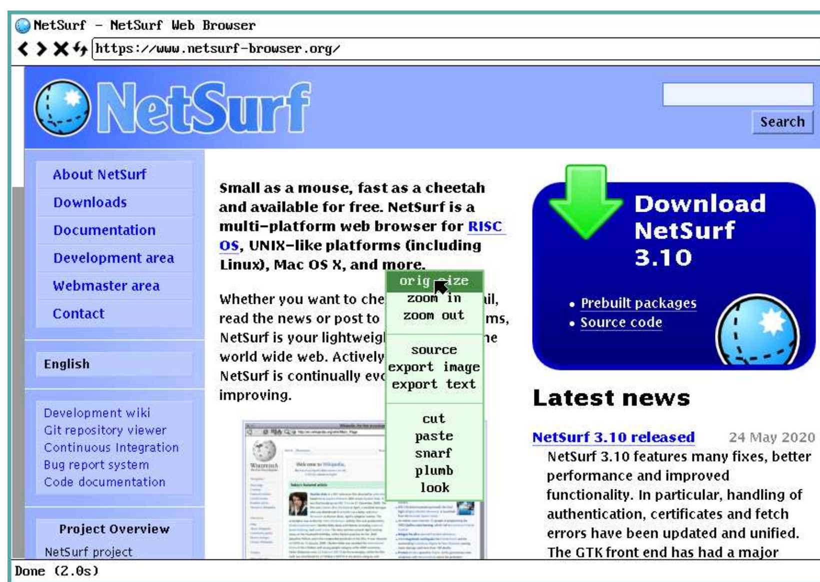


Figure 1: Screenshot of Netsurf with the native frontend

3. Porting challenges

Although Netsurf is written to be portable, and is available on many platforms that are quite different from each other, porting to Plan 9 is a special challenge for multiple reasons, one being that its C compilers differ from GCC. Another reason is that the build system depends on GNU Make, Flex and Bison. Then there is the general share of compatibility issues rising from differing environments in which the process runs in on various platforms.

† Virtual Network Computing, see vnc(1).

Table 1: Porting issues and possible solutions. An asterisk (*) indicates a preferred solution, and a dagger (†) the current solution.

Issue	Solution		
	Keep patches in the port	Changes in 9front	Changes in upstream Netsurf
Compiler			
Variable length arrays	†	Implement in kencc	Allocate on heap
Macro varargs (gcc extension)	†		Use C99 varargs
Statement expressions in macros (gcc extension)	†		Use static inline functions
Bitfields in static initialisers	†	Implement in kencc *	Use basic data types (short/int) on Plan 9
Zero–length arrays	†		Use C99 array[]
Environment			
Missing Posix functions in APE	Add missing functions in linking †	Add missing functions in APE *	
Libcurl (network resource fetching library)	Implement webfs(4) fetcher *†	Use libcurl port	
Build system			
Symbolic links in the upstream repository	9git makes physical copies of symlinks when cloning †		
Perl/Python (generating files)	Ship prebuilt files from a Linux build †	Port modern Perl and Python	
Flex/Bison (lexical and parser generators)	Ship generated lexer and parser files † Alt. write flex to lex & bison to yacc transpiler	Port GNU Flex & Python *	Rewrite rules to conform to both flex/bison and lex/yacc

Table 1 shows porting issues and alternative solutions. Solutions in the first solution column are patches or added functionality that have to be maintained in the plan 9 port. The second solution column denotes potential changes to the operating system, compilers and libraries to compile the upstream Netsurf sources without patching. Solutions in the last column suggest changes that might be accepted by the official Netsurf team in order to compile the sources more easily on plan 9.

3.1. Compiler

Netsurf generally follows the ISO C89 standard, with some C99 and some GCC extensions. The Plan 9 C compiler in 9front [4] is mostly C89 compliant with partial C99 support, along with some non-standard features. The native Plan 9 code base uses C libraries and header files that differ from the standard C libraries, but APE gives an emulation of the standard libraries as defined by Ansi and Posix.

Until now, all compiler differences have been bridged by patching the netsurf source, seen by the daggers (†) in the compiler section of table 1, making it hard to keep up to date with new releases of the browser. The goal is to solve all compiler differences either by suggesting changes to the upstream netsurf source or by implementing the missing features in 9front. If patches are not accepted, an alternative would be to write a transpiler that would convert the netsurf C to a C that can compile on Plan 9.

3.1.1. Variable length Arrays

Arrays in C are normally fixed in size, but ISO C99 allows a function to allocate an array which size is unknown at compile time, but this is not supported in Plan 9 C. GCC implements VLA:s by allocating space on the stack [5]. A simple solution, if not to implement VLA in kencc, is to reserve the memory on the heap using malloc(), which also is the current work-around, but requires patching.

3.1.2. Macro varargs

Preprocessor macros with variable number of arguments (varargs or variadics) is supported in kencc in accordance with C99, but the netsurf source uses an older GCC extension that predates the standard [6]. The solution would be to change over the upstream source to use the newer standard, as both ways should work with gcc.

3.1.3. Statement expressions

Netsurf uses statement expressions [7] in some preprocessor macros to speed up execution as the code get inlined. This is a non-standard GCC extension. These macros could be changed to 'static inline' functions instead, keeping the performance on existing platforms, while being ignored by the Plan 9 compilers.

3.1.4. Bitfields in static initializers

The upstream source of Netsurf uses bitfields to make some data structures smaller. Bitfields are generally supported in the Plan 9 compilers, but it is not possible to use them in static initializers. The current solution in the port is to replace bitfields with basic datatypes like short or int where needed, making the data structures a bit bigger, and adding a patch to forward. The better solution would be to add support for bitfields in static initializers in kencc.

3.1.5. Zero-length arrays

Declaring arrays of zero-length (array[0]) is allowed in GNU C as an extension, and is used in places in the Netsurf sources. This is not supported by the Plan 9 compilers, but they instead have support for C99 flexible array members (array[]) [8].

3.2. Environment

An application always runs in an environment, and a requisite to get Netsurf to compile was to use APE. APE is not a complete implementation of Posix, and some functions had to be added in the linking of the executable: pread(), pwrite(), ceilf(). It can be noted that pread() and pwrite() are available in the native libc, just not through APE.

An application can be linked either against the APE libraries (in /$objtype/lib/ape) or against the native Plan 9 libraries (in /$objtype/lib), as many functions have the same

names in both implementations and are designed to work with the other functions in the same set of libraries. The Plan 9 graphics and event handling can be accessed while using APE, but some things like the event timer etimer() is not available under APE, which was a problem in the event handler in the framebuffer frontend.

3.2.1. Webfs fetcher

Netsurf by default uses libcurl [9] to fetch content from various networked resources, and while a port of libcurl does exist for 9front, it brings a tail of dependencies on further libraries. It also doesn't follow the paradigm for accessing the web on Plan 9. Kyle Nusbaum instead implemented a Netsurf webcontent fetcher which uses the webfs(4) filesystem in Plan 9, wich was further developed by Philippe Mechai.

3.3. Build system

The official Netsurf source uses an elaborate system of GNUMake files, while the Plan 9 build system uses 'mk' instead. Both the upstream Makefiles and the mk-files compile library by library, but the mk-files generate the object files and library files directly in the 'src' directories whereas the Makefile build writes them to special build directories.

3.3.1. Symbolic links

The upstream sources use 'symbolic links' to avoid duplicates in the runtime resource files among different architectures and different languages. As plan 9 doesn't support symbolic links [10], the git client git(1) makes copies of those files for the plan 9 sources, and 'mk install' installs the needed resource files in /sys/lib/netsurf.

3.3.2. External tools for building

Apart from the C compiler, the build also depends on Flex, Bison, Perl and Python. For now, the files autogenerated by those tools are copied over from a Linux build.

3.3.3. Image libraries

The netsurf code base includes libraries for handling images in BMP, GIF and SVG formats, but external libraries are used to help display JPEG and PNG. Netsurf for Plan 9 vendors APE-compiled versions of libpng and libjpeg, not requiring to install any external libraries.

3.4. Merging with upstream Netsurf

Porting any larger software package raises the question on how to keep up to date with new releases. For the plan 9 port of Netsurf to become an official Netsurf port, it must be considered what to do with each of the change needed for the code to compile and run under Plan 9. In the first round all fixes were applied to a copy of the upstream source, which is hard to maintain.

Some of the fixes that stem from lacking support in the C compiler or Posix libraries in 9front can with time be fixed by adding in those features, given that they are accepted by 9front upstream. Some of the changes are needed because the upstream Netsurf utilizes GCC extensions or code that runs on GCC but is not following C89 or C99, and in those places it might be possible to convince the Netsurf core team to change their source so that it compiles both on their current targets as well as on Plan 9. The remaining changes must be maintained as a series of patches and extra files added to the build.

4. Discussion

Porting a software package, in the sense of bringing it to run on another operating system or hardware architecture, can be done at different levels, where one extreme is to run the existing software in a virtual machine emulating the operating system it was written for, and the other extreme is to change the source in all needed places for it to compile and run natively on the target system. The emulating approach most likely results in a program that doesn't fit very well into the target system, and that might not interact with it efficiently. The second approach has the major disadvantage that the porting largely must be redone for updates to the ported software. The Netsurf port tries to stride the middle–way, trying to change the original source as little as possible, but to compile and link natively on the target system using its native toolchain.

The current port avoids a number of dependencies on external tools (Flex, Bison, Perl, Python) by copying files from a Linux build. Would porting these tools be a better solution to be able to build from the source? The downside is that anyone who wants to compile the browser also must install all the tools required.

One might also discuss whether it is beneficial to maintain a Plan 9 specific build system with it is own 'mk' files mirroring the functionality of the upstream hierarchy of GNU Makefiles. The native build system obviously fit better with the system and, to its defense, is not updated as often as the source of the browser core and its libraries. Another solution would be to auto-generate Plan 9 mk-files from the official Makefiles.

4.1. Conclusion and future work

The Netsurf browser is steadily evolving, and an important goal is to keep up with changes and additions in the upstream source tree. The straighter the original sources compiles on Plan 9, the easier it becomes to keep up with changes and new releases.

While the technology of the web is in constant development, Netsurf is moving slower than the main players in the browser market, and it is a reasonable amount of work to keep the port up to date given that manual patching can be eliminated.

An alternative to manual patches for circumventing differences in the Plan 9 build environment and compilers might be to use an automatic code rewriting tool such as semantic patches [11] used by the Harvey operating system to convert Plan 9 code to C11 [12].

The Netsurf port to Plan 9 has already generated some fixes of hidden bugs in upstream Netsurf, and adding missing functionality to APE in 9front might benefit porting other useful Posix tools to Plan 9. A future goal might be to include the Plan 9 port as an official port in the upstream source repository, but it might be hard using a custom build system.

Another goal is to have the browser feel native in the Plan 9 user experience and support Plan 9 ways of doing things, a goal that has already come a long way with the native frontend, and having the native frontend makes it easier to improve that experience even more in the future.

5. Acknowledgements

Netsurf for Plan 9 with the framebuffer driver was initially ported by Jonas Amoson. Philippe Mechai wrote the native frontend and has been maintaining the source tree on Github. Kyle Nusbaum implemented the first versions of the webfs fetcher. Ori Bernstein initially set up the Github repository, has been rebasing changes for new versions of upstream Netsurf, suggested patches upstream, and more. Thanks also to Michael Forney, Sigrid Solveig Haflínudóttir, Lucas Fransesco and others for their contributions.

6. References

[1] Netsurf Web Browser, https://www.netsurf-browser.org.

[2] Netsurf port to Plan 9, https://github.com/netsurf-plan9.

[3] Howard Trickey, *APE — The ANSI/POSIX Environment*, Plan 9 Programmer's Manual, Volume 2.

[4] Ken Thompson, *Plan 9 C Compilers*, Plan 9 Programmer's Manual, Volume 2.

[5] Using the GNU Compiler Collection: *6.20 Arrays of Variable Length*, https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html.

[6] Using the GNU Compiler Collection: *6.21 Macros with a Variable Number of Arguments*, https://gcc.gnu.org/onlinedocs/gcc/Variadic-Macros.html.

[7] Using the GNU Compiler Collection: *6.1 Statements and Declarations in Expressions*, https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html.

[8] Using the GNU Compiler Collection: *6.18 Arrays of Length Zero*, https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html

[9] CURL, Command line tool and library for transferring data with URLs, https://curl.haxx.se.

[10] Rob Pike, *Lexical file names in Plan 9 or getting dot-dot right*, Proceedings of 2000 USENIX Annual Technical Conference, June 18—23, 2000.

[11] Yoann Padioleau, Julia L. Lawall, Gilles Muller. *Semantic Patches*, Proceedings of the 2007 Linux Symposium, Volume 2, Ottowa, Canada, 2007, pp. 107—118. https://www.kernel.org/doc/ols/2007/ols2007v2-pages-107-118.pdf

[12] Harvey OS, directory of semantic patches (Coccinelle), https://github.com/Harvey-OS/harvey/tree/GPL-C11/sys/src/spatches.

Plan 9 and Inferno Go to School

Brian L Stuart

ABSTRACT

Although not originally designed for educational purposes, Plan 9 and Inferno are excellent tools for use in Computer Science education. Here I discuss how I use them in my classes at Drexel University. The discussion includes CS314: Computing in the Small using Plan 9, the Operating Systems courses CS370 and CS543 using Inferno, and the computer I take to the classroom for presentations running Plan 9.

1. Introduction

Education in the realm of computing has long been influenced by the work at Bell Labs. The CARDIAC developed by David Hagelbarger and Saul Fingerman was the first exposure for many people to the inner workings of a computer.[3] My CS164: Introduction to Computer Science course still uses the CARDIAC as an architectural example, and students get their first taste of machine language programming on a CARDIAC simulator that runs in their web browser. An entire generation of Computer Science students learned how an operating system really works by studying John Lions' commentary on 6th Edition UNIX.[4]

Drexel University has a long history of innovation and contribution to Computer Science Education. In the early 1960s, when the name was Drexel Institute of Technology, one of the widely distributed FORTRAN manuals for the IBM 1620 was written by a Drexel faculty member.[2] In 1984, the university partnered with Apple Computer becoming one of the first schools to provide Macintoshes for its students. Today, the Computer Science Department as part of the College of Computing and Informatics is one of the strongest units in the university. Each year, we have hundreds of students enrolling as Computer Science and Software Engineering majors. Almost all of our students participate in cooperative education, gaining substantial industry experience prior to graduation, and our graduates are highly respected in the industry.

For all the courses discussed, one of the key principles that guides my teaching is that it is critical to understand how things work, and not just how to use them. One of the best expressions of this was found on the office chalkboard of Richard Feynman after his passing: "That which I cannot create, I do not understand." Without understanding the mechanisms under the covers, seeing example code that one just copies is no better than the children at Hogwarts memorizing spells. If I understand how something works, I can find lots of ways to use and apply it, but if I only know a way to use it, then I'm limited to that procedure.

2. CS314: Computing in the Small

Like most programs in Computer Science and Software Engineering, ours doesn't require our students to get any direct experience with hardware. Yet, there are more CPUs running in embedded systems than any other type of computer system. To provide students with an opportunity to have some experience with embedded systems, we offer CS314: Computing in the Small.

Historically, embedded systems ran on relatively small processors without a conventional operating system. This type of embedded design is still the right way to go for many applications. Today, however,

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

we frequently find systems being built around ARM or MIPS processors with enough horsepower to run operating systems like Linux. It is this latter approach to embedded applications that this course addresses.

For this class, I chose the Raspberry Pi as a good platform. Certainly the price point is ideal for students (except for the recent price-gouging). The Pi has a good variety of I/O support, giving students a chance to work with a number of different devices. It provides GPIO lines, SPI ports, and I^2C interfaces.

Perhaps the best reason to use the Raspberry Pi for this class is that there exists a solid Plan 9 port for the Pi. There are several reasons to prefer Plan 9 over Linux in this sort of class. One reason is that the Plan 9 kernel is written in a much clearer and more straightforward way. This makes it much easier for the student to understand how a user-land operation makes its way to setting the device registers. Being able to see that connection is critical for the level of understanding that I'm trying to get across.

Another important benefit of using Plan 9 is the fact that it is the path less taken. This often seems contradictory to students who believe that learning in programming is always about examples that they can basically copy and tweak. They expect to use web searches to find all possible lines of code that they just assemble. However, this does not represent a significant level of understanding. By requiring that the students construct their code using essentially only the man pages for reference, I can guide them to understanding what they are doing at a deeper level. Related to this is the fact that unlike what is often presented for Pis in the Linux and Python world, programming is not all about using a library that does everything for you.

The third reason I prefer Plan 9 for this course is to expose students to some of the newer and cleaner ideas found in Plan 9 over Linux and other UNIX-likes. For example, the file-oriented interfaces provided by the drivers in Plan 9 using ASCII text make it much easier to demonstrate things in class. They help the students see that APIs with complex structures and many arguments aren't the only way to do things. The hope is that they will come to understand that just because they see something in industry doesn't mean it's the right way to do it. Far too much of what is done under the label of "best practice" is, in fact, quite bad practice.

The following subsections discuss the assignments I give in this class. As a matter of full disclosure, I did have a hand in developing these drivers as part of the original course development. Much of the driver testing was done on devices that are listed as recommended for the students to use.

2.1. GPIO

Giving an initial assignment that just asks for a simple input and output using GPIO lines provides the students with an easy assignment that makes sure they have the system up and running on their Pi, that they understand how to connect devices, and that they understand how to use the compiler. The actual assignment is generally simply to connect a switch of some sort and an LED of some form to a pair of GPIO lines and to control the LED with the switch. I encourage them to implement a three-state machine with states of on, off, and flashing, but if they want to just implement a flashlight, that's okay for this assignment.

Although it's a rather easy assignment on the surface, there are a few details that the student must think about and understand. On the output side, the student needs to first understand whether they have a raw LED that needs a current limiting resistor or a device that integrates the resistor with the LED. There was a time when we could assume that all Computer Science majors knew Ohm's Law and could understand the role of the resistor, but that's no longer the case. As a result, I do explain that in class.

On the input side of the GPIO assignment, there are also questions that the student must address. One of the devices that students might purchase from Adafruit and use provides a logic output level based on a capacitive touch sensor. If they use a device like that, then the input GPIO line is best configured without either an internal pullup or pulldown. On the other hand, if the student has a simple mechanical switch, they need to understand the role of the pullup or pulldown and how to decide which

to configure. Overall, this relatively simple assignment gives the student an early and quick taste of the world of embedded systems.

2.2. SPI

SPI is one of two commonly used synchronous serial interfaces with hardware support on the Pi. I typically give them an assignment to write code that uses an LCD panel driven with SPI. Particularly with the small color LCD panels, most of the controller chips they use are quite similar. To use them, the student must understand the protocol for accessing a device's internal registers. Once the students understand how to do that, they are then faced with a myriad of pixel formats to choose from. This gives them a chance to understand the tradeoff between color detail and performance. Using a display device also exposes the student to the effect of the SPI clock rate. Particularly when writing a whole screen to clear it, the difference made by different clock rates is quite evident. It's also a good exercise for them to see how a maximum clock rate specified in a datasheet can often be exceeded, but usually not relied on.

Of course, SPI is used for other types of devices besides just LCD screens. I discuss in class one of my more recent uses of the SPI interface. A number of LED arrays are currently available that are constructed with 8×8 LED modules attached to a controller board. The controller board drives the LEDs with what amounts to a large shift register. SPI is a particularly convenient interface for devices like this as its different modes allow one to select the combination of rising and falling edges appropriate to the device. I recently used one of these as part of an ENIAC simulation running on a Plan 9 Raspberry Pi.

2.3. I^{2}C

Most of the I^{2}C devices the students use are sensors of various sorts. Among the more popular ones are various light sensors and accelerometers. One part of I^{2}C that is especially challenging from all perspectives is the subaddressing. Devices often use this feature to identify which device register is being accessed. However, the details of its implementation are quite tricky, and the mechanism exposed to applications is perhaps unintuitive. In particular, the offset in a pread or pwrite call is used (some might say abused) as the subaddress. The justification for this design choice is that if we view the register file of the device as exactly that, a file, then the idea that the offset is another way to specify the address of the register is natural. This functionality gives the student a chance to better understand the role of the offset, and the use of pread and pwrite.

2.4. 1-Wire

The 1-Wire interface is an especially interesting one. It is so named because it uses a single line for both input, output, and power. Of course, there still must be a ground connection, so calling it "one" wire is a little euphemistic. The Pi hardware doesn't have direct hardware support for the 1-Wire interface, but by playing with the GPIO line function, it is possible to implement.

The Pi acts as the bus master and all devices share a single line with a pullup resistor. Whether reading or writing, the bus master controls the clocking of the data. It does so by driving the bus line low for a short period (approximately 2μs). If neither the bus master nor any device drives the bus line low, the pullup pulls the bus high. When the Pi is transmitting, it initiates a bit by driving the line low. If transmitting a 0, the line is kept low for longer (typically 20 to 60 μs). If transmitting a 1, the bus is set to high impedance, letting the pullup pull it to a high logic level. When the Pi is reading data, it still initiates each bit transmission by driving the line low for a short time. The Pi then waits about 10 to 15μs and samples the bus line. If the device is sending a 0, it drives the line low, and if a 1 the bus is left high impedance. By setting the output to a 0, we can switch the GPIO line function between input and output to speak the 1-Wire protocol. When the line is set to output, the 0 output drives the line low, but when set to input, it goes to high impedance. Although we (my students and myself) have been able to run the 1-Wire protocol strictly from userspace, I've added an additional function to

the GPIO driver to support 1-Wire. Writing "function n pulse" to /dev/gpio switches the line to input, delays 2μs, and switches it back to output.

2.5. Device Driver

Because of the importance of understanding how everything works and all the layers of abstraction, I do make a point of teaching the structure of drivers in Plan 9. It gives the students a sense of what's involved with doing embedded programming without an operating system supporting them. It also gives the student a chance to see an exemplary system design.

Given the exposure students receive to drivers, it makes sense to give them a chance to experience modifying and rebuilding the kernel. Most often for this project, I give them a relatively simple modification to an existing driver, rather than expect them to create a new driver. Both the 1-Wire assignment and the device driver assignment depend on available time in any given term.

2.6. Final Project

Because I like to give students a chance to stretch their wings and show creativity, I give them the opportunity to specify their own final project, subject to my approval. It's always quite interesting to see what ideas students come up with. One that particularly comes to mind is the student who connected some sensors to his child's wooden train set and then drove motors to create gates at crossings. Another project that I remember was a student who purchased a cell phone module and connected it to the Pi's serial port. By the end of CS314, he was able to send text messages. Because he was interested in taking it further, he followed the course up with an independent study, adding more cell phone functionality.

3. CS370/CS543: Operating Systems

The first thing to address when talking about an operating systems course is whether it's taught from the perspective of applications with a focus on using system calls, or whether it's taught from an internals perspective. Both the undergraduate and graduate operating systems courses I teach at Drexel are very much internals courses. While many courses are taught with a primary focus on standard algorithms, my preferred approach is strongly influenced by Lions' commentary on 6th Edition UNIX.[4] I believe that students get the most out of a course that provides in-depth exposure and experience with one well-designed system.

There are, of course, several options to choose from when teaching a course centered around the internals of a particular system. Like many, I have taken and TA'ed a graduate course based on XINU, and I've taught using MINIX a number of times. When Vita Nuova open sourced the code to Inferno, the stage was set for the way I really wanted to teach the course. This development resulted in the textbook *Principles of Operating Systems: Design and Applications.*[5] Five of the chapters in the book attempt to present aspects of Inferno internals in a way inspired by Lions.

Inferno has a number of characteristics that I want in a system for teaching operating systems. When teaching with MINIX in the '90s, I could expect students to partition their drives and install MINIX so that they could dual boot, but that quickly became too much to ask. Although running MINIX on a simulator would have been an option, Inferno's ability to run hosted by another OS made it even more convenient. However, Inferno is not an artificial system that only runs in simulation. The fact that it also runs natively on bare hardware means that we have device drivers and their interrupt handlers that can be studied. Another important aspect of Inferno is that it is written in the beautiful and elegant style of Bell Labs, giving students a chance to see very well-written code. Finally, Inferno incorporates ideas such as per-process name spaces that are newer than what is usually seen in operating systems classes.

As with CS314, the following subsections discuss various programming assignments that I give in CS370 and CS543. Unless specified otherwise, the projects are ones that might be given in either the

undergraduate or the graduate version of the course. Even when the core of the assignment is the same, I generally require the graduate version of the class to work with native code, rather than hosted.

3.1. First Project

The first project in both these courses is generally a pretty easy one that is mostly about the students getting the system installed and them getting used to working in a larger code base and building the system. One of the most common things I ask them to do is print out the time that the kernel was built in hours, minutes, and seconds. I've stopped asking them to give month, day, and year, because they all just look up some algorithm online. As a hint, I tell them that they should look for KERNDATE in the output of the build process. The idea is that the student should recognize that KERNDATE is defined with a -D option to the compiler when compiling $CONF.c. If they look for how KERNDATE is used in that file, they will find that it's used as an initializer for a global variable kerndate. Finally, students need to recognize that to manipulate kerndate and print the results from a file like dis.c, they need to add an external declaration. Overall, it's an easy assignment, but it gets them to start working with the code.

3.2. Process Management

The second project is generally related to the process handling code of the system. There are two types of processes in Inferno, kernel processes and user processes that run compiled Limbo code in a Dis VM. In this course, we put most of our attention on the user processes.

Relatively easy forms of this assignment involved adding some form of instrumentation to the scheduler. More involved assignments ask the student to modify the scheduler. Sometimes I have asked for a UNIX-style priority scheduler. In other cases, I've asked the student to add mechanisms for adjusting the length of a quantum. One other process management project that I've sometimes assigned is to add a simplified suspend and resume mechanism roughly similar to the BSD stop and continue mechanism.

3.3. Memory Management

The Inferno memory management is built around variable sized allocation from a free block data structure consisting of a binary search tree where each node is a circular doubly linked list. This is a wonderful data structure to test whether students really understand how to deal with data structures. At the same time, it does open the door to experimenting with different structures here. To that end, the most common assignments I give on memory management involve replacing the tree of linked lists with something else. One common version is to have the students replace it with a single linked list. Another is to implement something like the slab allocator in Linux. The idea here is that some request sizes are very common, and it would be more efficient to just put blocks that get freed into lists of common sizes, rather than repeatedly splitting and coalescing them. The single list is more typical for the undergraduate section and the slab-type allocator is more typical for the graduate section. Although often frustrating to the student, one of the most educational experiences about this assignment is how bugs in the memory allocation can prevent the system from booting at all. Debugging that situation is often a new experience for many students.

3.4. I/O Devices

For the undergraduate section of the class, I generally settle for going over code from kb.c and sdata.c in class and explaining why talking directly to controllers is something that we only see in the native versions. However, because the undergraduate sections generally work with hosted builds of the system, there's not really an opportunity for them to work directly with devices. On top of that, because our terms at Drexel are 10-week quarters, four kernel assignments are probably enough for undergraduates.

For the graduate section, I do often add a fifth kernel project (but require fewer problem set submissions). The additional assignment is usually related to the keyboard driver, as that's something that's relatively

Figure 1: Raspberry Pi400 Presentation Set-Up

easy and can be done with a minimal image booted from a virtual floppy. Most often, I ask them to support multiple key mappings that can be switched with a special key sequence.

3.5. File Systems

Although I do cover the implementation of kfs in class, I don't expect the students to come up to speed with Limbo enough to do a project with it. Instead, I have my students develop a new kernel server, and to make the job easier, I give them a devskel.c to work from. One of the simpler versions of this is a simple string manipulation. A string is written to the served file and when the file is read, some modification of the string is retrieved. Another form is a simplified encryption layer that encrypts and decrypts data as it's passed to a backing store. One of the more complex versions of the assignment is for the student to implement a RAID driver layer. Complete and thorough implementations of the encryption or RAID layers can be tested by running kfs on top of them.

4. Raspberry Pi400 Presentation Machine

The final use of Plan 9 discussed here is the machine that I take into the classroom to do my presentations. It is a Raspberry Pi400. This machine is basically a Pi4 built into a keyboard, making it very compact and convenient for use in the classroom.

All of the classrooms in our building are equipped with podiums that are connected to projectors and wall monitors. Each podium does have a computer running Windows installed, but many faculty bring their own laptops to use in the class. However, the podium gets a little crowded with a full laptop in addition to the monitor attached to the podium. On the other hand, using the existing monitor with the Pi400 is much easier to work with and leaves room for both a mouse and a drawing pad to use for a virtual white board.

The Pi400 I use in the classroom runs Plan 9, with both rio and acme configured to use larger fonts than usual to aid readability on the projector screen. Part of the reason for using Plan 9 is that it is, of course, a much more pleasant platform for developing tools for my classes. It also sets a good example

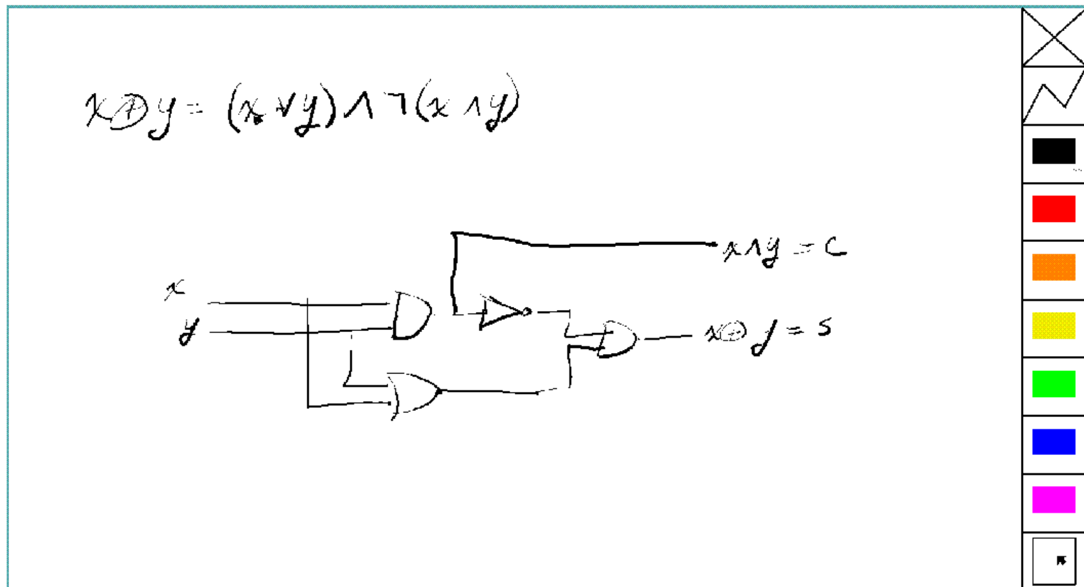


Figure 2: Virtual White Board Example

for students to see that not all of computing need be tied to mainstream and commercial systems. Figure 1 shows the system set up on one of our classroom podiums as I use it in class.

There are a few details of the use of the Pi in that context that deserve some discussion. The first of these is the virtual white board mentioned above, an example of which is shown in Figure 2. The slate device I've been using is one that our IT group had, but no one had ever used. When I asked about recommendations for such a device, they said "take this one; nobody ever used it." It looks like a generic USB HID, and its protocol was fairly easy to discern. My initial approach was written as a program using the P9P library, running on a BSD UNIX sending output to an instance of wish to handle the graphics. When migrating to the Pi, there were two major shifts. First, usbd didn't automatically present a generic HID interface, but handling that was a simple matter of a small program using the USB library to locate and present the slate device. With it, reading from /dev/slate gets a single protocol message from the device.

Another change in the virtual white board highlights one of the best design decisions of Plan 9. In other environments, writing to the graphical display typically requires access to special system calls or special libraries, making it at best a nuisance when working in small experimental languages. In those setting, I have often gotten around the problem by running an instance of wish as a child process and sending Tk commands down a pipe. However, this still requires the existence of another program such as wish written in another language, reducing the stand-alone potential of the experimental language. On the other hand, by providing access to the draw device using ordinary file operations, a language's interpreter or run time environment aren't required to have access to these special interfaces. The relevance here is that the virtual white board is written in just such a small experimental language.

As second issue that must be dealt with when using Plan 9 in such a context arises from the unfortunate push toward the web browser as the universal UI. As with all other bureaucratic organizations, universities are far from immune to this phenomenon. In particular at Drexel, the primary vehicle of distributing material to students (syllabi, assignments, grades, etc.) is a Learning Management System (LMS) called Blackboard. Along similar lines, the mechanism by which both faculty and students learn of the final exam schedule is the Registrar's web site. As a result, there are days when I cannot escape

using a browser in the classroom. All of these web interfaces are heavily JavaScript-laden and often depend on the frameworks du jour. Although recent work on porting netsurf[1] shows promise, we are still a long way from a native Plan 9 web browser that can function with the assumptions made in the web development. To deal with this situation, I euphemistically "declare victory and depart the field of battle" by running a NetBSD machine in my office on which I run a web browser and to which I connect from the classroom using VNC.

The third issue that I've dealt with in my use of Plan 9 in this role is ssh. Currently, the SSHv2 implementation in the 9legacy distribution falls rather short of ideal, a situation for which I accept personal responsibility. It was originally written with the various cryptological algorithms that were required by the RFCs included, but not many beyond that. In the years since, several of these algorithms have been dropped from the default configuration of OpenSSH (and presumably others). When I first started using my Pi in this way, it was unable to connect with our departmental cluster because the key exchange algorithms were out of date. After adding newer versions of Diffie-Helman key exchange algorithms, I was able to use the system successfully during our fall term using ssh directly and as a carrier for sam -r. More recent updates to the cluster, however, have revealed that the original ssh-rsa and ssh-dss public key algorithms have now also been dropped, and I am in the process of adding replacements for them.

5. Conclusion

Without Plan 9 and Inferno, my approach to teaching, especially in systems courses, would be very different, and my students would be the worse for it. The tradition that Bell Labs established long ago of connecting their research to education and reaching out to the classroom has had an important impact on many of us. Although the Bell Labs of that time is no longer with us, its impact continues on through the influence of some of its later work on today's students. Those who continue the use and develop Plan 9 and related software are making a meaningful contribution to the improvement of the overall calibre of people entering the field.

References

[1] Jonas Amoson. Porting the netsurf web browser to Plan 9. In *Proceedings of the 9th International Workshop on Plan 9*, 2023.

[2] Decima M Anderson. *Basic Computer Programming: The IBM 1620 FORTRAN*. Appleton-Century-Crofts divison of Meredith Publish Company, New York, NY, USA, 1964.

[3] David Hagelbarger and Saul Fingerman. *An instructional manual for CARDIAC*. Bell Telephone Laboratories, 1968.

[4] John Lions. *A Commentary on the UNIX Operating System*. Republished 1996 by Peer-to-peer Communications, 1977.

[5] Brian L. Stuart. *Principles of Operating Systems: Design and Applications*. Cengage, 2009.

NinePea — A Small 9P Library for Arduino and Plan 9

Eli Cohen
echoline@gmail.com

ABSTRACT

NinePea is a tiny library for 9P servers specifically designed for use with Arduino Mega boards connected to Unix-like operating systems and Plan 9.

1. Introduction

NinePea [1] was originally a project for Arduino. It implements the 9P [2] filesystem protocol for servers running on the Arduino board. Compared to other 9P libraries it is more compact. NinePea sacrifices many of the features they would have in favor of minimalism. It only supports the 9P2000 version of 9P and does not support auth. It does not do much error checking, and is only meant to be used locally. This is for the advantage of being able to work on systems with only a few kilobytes of RAM. *NinePea* is intended for Arduinos and packaged as an Arduino library, but it is portable C which can easily be compiled on other systems.

2. Synthetic Filesystems

Linux users may be familiar with the `/proc` filesystem which is a synthetic filesystem from the Linux kernel exposing information about running processes and other system information. *NinePea* is used to create synthetic filesystems such as these on separate hardware devices or as programs running in user-space. Examples of the usage of *NinePea* include pinfs which presents the digital GPIO pins of an Arduino as files, randomfs which implements a hardware random number device file using a mountable Arduino, V4LFS [3] for Linux which presents the most recent picture taken by a webcam as a JPEG, a hardware cons device using a television monitor and a PS/2 keyboard connected to an Arduino, and the unfinished etherESP32 [4] implementation which is intended to turn an ESP32 board into a mountable WiFi network device.

3. Arduino Boards

Arduinos are simple open-source electronics prototyping boards. Programs written in the Arduino IDE for use with these boards are called sketches. Arduino started as a board with simple Atmel AVR chips. The Arduino Mega 2560 is based around an Atmega2560 chip, a programmable MCU with only 8 KB RAM, 256 KB of programmable flash, and 4 KB of EEPROM storage. It has 16 analog-to-digital inputs, 70 pins that can be used for digital I/O, and a lot of other functionality for prototyping. These boards can drive pulse-width modulation signals, be used as controllers for SPI or I2C, and have 4 TTL serial ports. The Arduino project also encompasses a simple IDE in Java, which compiles C++ sketches to software for the AVR chip. This of course doesn't run on Plan 9, at least at the time of this writing. Plan 9 doesn't support Java or C++, and there is

M. Engel (Ed.): 9th International Workshop on Plan 9 (IWP9)
Waterloo, Ontario, Canada, 21–23 April 2023, published at http://iwp9.org

little else available on Plan 9 for programming Atmel chips. All of these examples were compiled, uploaded, and used on Linux.

4. Pinfs Example

Pinfs is a simple example Arduino sketch. It serves a single directory that shows each of the digital pins as a file. Here is a usage case on Linux using the 9mount convenience program and netcat to listen on TCP/IP:

```
$ mkfifo pipe
$ ./tty9p /dev/ttyUSB0 < pipe | nc -l -p 2000 >> pipe &
$ 9mount tcp!localhost!2000 /mnt/arduino
$ ls /mnt/arduino
d00 d05 d10 d15 d20 d25 d30 d35 d40 d45 d50 d55 d60 d65
d01 d06 d11 d16 d21 d26 d31 d36 d41 d46 d51 d56 d61 d66
d02 d07 d12 d17 d22 d27 d32 d37 d42 d47 d52 d57 d62 d67
d03 d08 d13 d18 d23 d28 d33 d38 d43 d48 d53 d58 d63 d68
d04 d09 d14 d19 d24 d29 d34 d39 d44 d49 d54 d59 d64 d69
```

This is the pinfs example running on the Arduino. First a fifo pipe is made with the standard mkfifo command. The tty9p utility is used to set serial port parameters and packetize the 9P messages on the serial port, and it is looped through the fifo pipe and netcat to listen on TCP port 2000. 9mount is a convenience program for making 9P mounting simpler on Linux, and here it is being used to mount the serial port through the above command. Then the filesystem can just be accessed with the standard Linux tools. Shown here is a listing of all the synthetic files being generated by the Arduino board. This example is running on Linux, but a Plan 9 computer can easily mount the service on the Linux computer over TCP. The user can write a 1 or 0 to d13 to turn on or off the LED on digital pin 13 of the Arduino board, and can read or write any of the 70 files to access the digital I/O state of any pin.

On Plan 9, the serial port can be mounted directly. 9P doesn't care about the underlying transport, so on Plan 9 this works with any file giving a 9P server:

```
% mount /shr/usb/eiaU50b4c /n/arduino
```

5. Randomfs Example

Another example of *NinePea* is the randomfs sketch for Arduino. It presents one file, `random`, which can be mounted over `/dev` to replace Plan 9's `/dev/random`. This sketch is a bare example of a 9P fileserver, it only presents one file. On read it reads each of the 16 analog inputs on the Arduino Mega and XORs them with the system milliseconds counter:

```
for (pin = A0; pin < (NUM_ANALOG_INPUTS + A0); pin++) {
    seed <<= 1;
    seed ^= analogRead(pin) ^ millis();
}
```

This is to seed a Chacha20 pseudo-random number generator. The Chacha20 cipher code [5] in this sketch is borrowed from 9front. The file it presents can be bound before `/dev/random` to replace it with the random number generation from the Arduino. `AnalogRead(pin)` returns the ambient voltage on that pin which is XOR'ed with `millis()` which is used as the seed for a PRNG. `Millis()` returns the number of milliseconds the Arduino has been booted, and the ambient voltage is similar on each pin, so the randomness of the seed might not be great, but then the sketch uses a Chacha20 stream cipher from that seed. The manual page for `/dev/random`

specifically says it is only to be used on Plan 9 as a seed for better pseudo-random number generators in the programs themselves. This sketch has not been thoroughly assessed for how well it generates randomness or how cryptographically secure it is. Additional hardware could be attached to the analog inputs to better seed the hardware random number generator device. For example, decaying particles from a block of radioactive metal could be used to input random voltages to the pins for seeding the hardware random number generator.

6. Design Choices and Further Examples

The *NinePea* library was designed to be compact. It leaves out a lot of functionality that other 9P libraries have. It doesn't have anything in it specifically for any transport layer. It only operates on buffers which are filled and read from by the implementation. It doesn't multiplex connections; this has to be handled by the implementation, if at all. *NinePea* doesn't do anything special for delayed RReads or TFlushes. This can be handled separately by the implementation. TReads can be buffered and replied to only when they are ready. By default, everything is handled in-order by the proc9p function. An incoming TRead is processed immediately and an RRead is sent. One project using this library was a hardware cons device using an Arduino Mega 2560, a keyboard, and a television monitor. In that project, RReads sometimes needed to be delayed, and TFlushes needed to be handled. The way this was achieved was by buffering TReads and handling them separately from other message types elsewhere in a loop.

Another unfinished project used an ESP32 board as a 9P-speaking wifi dongle. The goal of this project was to make a wifi dongle for Plan 9 that only required a USB-to-serial driver, and then could simply be mounted as a 9P device. In that project delayed RReads were handled by every incoming 9P message spawning a separate thread. When threads were complete, they locked a shared lock and sent their response.

The fileserver introduced above is an example of *NinePea* being used on Linux, instead of an MCU. It layers *NinePea* over Linux's video4linux subsystem for webcams. It presents only one synthetic file, `jpeg` , which is a synthetic jpeg generated by taking a picture from the webcam. The contents of this jpeg change over time, always showing the latest image from the webcam. Most people who have used a modern computer are familiar with what a JPEG file is. It's a picture which is generally in disk storage. The user can open it, delete it, etc. A synthetic file's contents are generated, rather than stored on disk. Imagine opening a JPEG and it's the most recent image from a webcam. An implementation of exactly this is V4LFS for Linux which uses *NinePea* directly. *NinePea* was designed to be portable C which can be included on many different types of systems to create fileservers such as this.

7. Comparisons to Other 9P Libraries

NinePea is intended to be very portable and compact. Running `wc-l` to count the lines of code in all the .c files in /sys/src/lib9p shows that Plan 9's 9P library is only 2781 lines of code. Libixp for Unix-like systems shows 4045 lines of code in all the C files of the main library. *NinePea* however, is only 496 lines of code in NinePea.cpp. Arduino labels the file as C++, but it is completely portable C which can be used on many other systems. Plan 9's Lib9p does a lot of things that NinePea does not. NinePea does not handle authentication, it doesn't have built-in queueing, and it doesn't have any functionality to post a service or to handle threading or any type of listening. *NinePea* only operates on one buffer in memory, which the programmer must handle separately on their own. Outgoing response messages overwrite incoming messages in the same buffer, which is

only 4 KB by default.

8. NinePea Library Usage

NinePea is a minimal 9P server library which was originally intended for use with Arduino Megas. Some code in the header file for the structs, message types, and bit-field flags was borrowed from Plan9port [6]. Everything for a 9P server runs on the Atmega chip on the Arduino board. To use the library, first callback functions for each of the 9P message types [7] must be written and pointed to in a callbacks struct. Fids are handled manually with helper fid hash table functions.

```
Serial.begin(115200);

fs_fid_init(64);

callbacks.attach = fs_attach;
callbacks.flush = fs_flush;
callbacks.walk = fs_walk;
callbacks.open = fs_open;
callbacks.create = fs_create;
callbacks.read = fs_read;
callbacks.write = fs_write;
callbacks.clunk = fs_clunk;
callbacks.remove = fs_remove;
callbacks.stat = fs_stat;
callbacks.wstat = fs_wstat;
```

After that, a 9P message is read from the serial device and buffered into RAM. The buffer and callbacks structure are then passed to the proc9p function which processes the message. Proc9p calls the callbacks with at least an Fcall struct as a parameter, and also buffers for reads and writes. The 9P message buffer is overwritten by proc9p with 9P data to send back to the client, and proc9p returns the total length of the resulting 9P response. An Arduino sketch can then send that buffer back over the serial port.

```
r = 0;
while (r < 5) {
  while (Serial.available() < 1);
  msg[r++] = Serial.read();
}

i = 0;
get4(msg, i, msglen);

if (msg[i] & 1 || msglen > MAX_MSG || msg[i] < TVersion || msg[i] > TWStat) {
  // error
}

while (r < msglen) {
  while (Serial.available() < 1);
  msg[r++] = Serial.read();
}

msglen = proc9p(msg, msglen, &callbacks);

Serial.write(msg, msglen);
```

Handling fids is a bit tricky. Some helper functions make this easier:

```
struct hentry {
    unsigned long id;
    unsigned long data;
    struct hentry *next;
    struct hentry *prev;
    void *aux;
};

struct htable {
    unsigned char length;
    struct hentry **data;
};

struct hentry* fs_fid_find(unsigned long id);
struct hentry* fs_fid_add(unsigned long id, unsigned long data);
void fs_fid_del(unsigned long id);
void fs_fid_init(int l);
```

9P typically has a maximum of 8 KB per message, and these chips have only 8 KB of RAM. Linux's 9P support has a minimum of 4 KB message size, which barely fits here. The iounit is set to 4 KB by default. On a Linux computer the included tty9p program ensures that an entire 9P message is sent or received one at a time, but other than that everything runs on the Arduino. The serial port itself becomes a 9P fileserver endpoint.

9. Previous Work

Inferno's Styx protocol, which is compatible with 9P, was previously used [8] on Lego Mindstorms RCX bricks. That work was specific to the Lego RCX; it was never meant as a library. Styx-on-a-Brick was only used for one server for the Lego brick that exposed the motors and sensors. NinePea is a more general purpose library in portable C. It is a very small implementation of a 9P server intended for systems without many resources. It does borrow some structs and other header data from Plan9port, but it dispenses with a lot of functionality that would be available in other 9P server libraries.

10. Other Uses

NinePea was originally meant for use on Arduino boards. It could be made to work with a wifi shield to present a slow ethernet device for Plan 9 without writing any drivers. One could add a speaker and have a simple audio device. Arduino is meant for electronics prototyping, and although *NinePea* only builds under the Arduino IDE, once the board is configured and programmed as desired it can be plugged into a Plan 9 system and mounted like any other 9P server. It can be used to gather information from I2C or SPI sensors, to construct or read a signal, or for many other electronics prototyping applications. *NinePea* is also flexible; the Arduino library is labelled a C++ file but it's really just portable C. It's just a header and a C file that can be included with a project for simple 9P support. It does have some drawbacks. It isn't meant to be public-facing. It doesn't do authentication and it barely does any error checking. The V4LFS program shows how it can be included and used on Linux to wrap a webcam as a synthetic JPEG.

11. Performance of 9P

This is an example of the pinfs sketch. One interesting use of *NinePea* was using Plan 9 methodologies to bind the networking stack of the Linux machine the Arduino was plugged into over /net of a computer across the country and mounting the Arduino remotely:

```
linux$ ./tty9p /dev/ttyUSB0 < pipe | nc -l -p 2000 >> pipe

cpu% bind /mnt/term/net /net
cpu% srv tcp!localhost!2000 arduino
cpu% mount /srv/arduino /n/a
```

This command sequence serves the serial port on TCP port 2000 from Linux, switches over to using the Linux machine's networking stack on the remote Plan 9 computer, posts a 9P service for connecting to port 2000, and finally mounts the service. After doing so, writing a 0 or 1 across the country and back takes almost a full second of 9P traffic back and forth:

```
cpu% echo 1 > /n/a/d13
```

9P adds a lot of overhead of messages going back and forth, besides the data inside it. In this case only one byte was being sent, but the overhead of 9P and the Internet across the country and back caused the data to take quite a long time to be sent out across the Internet, return, and finally go out and back over the 115200 baud serial port. The serial port was not the main bottleneck in this case. 9P is still very slow over long distances because of all the overhead going back and forth for each operation. Each operation of writing a 1 to the d13 file involved several bytes of 9P out and back for walks, opens, writes, and closes. Mounting *NinePea* locally on the Linux computer, the main bottleneck as expected was the serial port itself, sending 9P back and forth as quickly as it could. The Arduino has an LED on digital pin 13 and LEDs for serial recieve and transmit. When it was mounted locally the LEDs for the serial port stayed on continuously while blinking the pin 13 LED in a loop, whereas when it was mounted remotely there was a visible delay as each 9P message was received.

12. References

[1] `https://github.com/echoline/NinePea` The source for this project
[2] `https://9p.cat-v.org` A site about 9P
[3] `https://github.com/echoline/V4LFS` NinePea-based V4LFS Linux webcam fileserver
[4] `https://github.com/echoline/etherESP32` ESP32 WiFi dongle
[5] `http://git.9front.org/plan9front/plan9front/HEAD/sys/src/libsec/port/chachablock.c/raw` Chacha20 stream cipher code
[6] `https://9fans.github.io/plan9port/` Plan9port website
[7] `intro(5)` Introduction to manual section 5 of Plan 9
[8] `http://doc.cat-v.org/inferno/4th_edition/styx-on-a-brick/` Inferno "Styx-on-a-Brick" paper

Author Index

Amoson, Jonas, 91

Bernstein, Ori, 35
Boddie, David, 63

Cohen, Eli, 107
Collyer, Geoff, 53

Gette, Guillaume, 1
Gibson, Andrew D., 45

Hancock, Ryan, 27

Klein, Edouard, 1

Mashtizadeh, Ali José, 27
Moody, Jacob, 21

Preil, Noam, 83

Sigrid, 83
Stuart, Brian L., 67, 99

Tsalapatis, Emil, 27