Proceedings of
3rd International Workshop on Plan 9
October 30-31, 2008

IWP9 2008

Computer and Communication Engineering Department
University of Thessaly
Volos, Greece

Organization

Organizing Committee

Spyros Lalis, University of Thessaly
Manos Koutsoumpelias, University of Thessaly
Francisco Ballesteros, Universidad Rey Juan Carlos de Madrid
Sape Mullender, Bell Labs, Alcatel-Lucent

Program Committee

Richard Miller (chair), Miller Research Ltd.
Peter Bosch, Bell Labs, Alcatel-Lucent
Geoff Collyer, Bell Labs, Alcatel-Lucent
Latchesar Ionkov, Los Alamos National Laboratory
Paul Lalonde, Intel Corp.
Eric Nichols, Nara Institute of Science and Technology
Brantley Coile, Coraid Inc.
Charles Forsyth, Vita Nuova Ltd.

Table of Contents

Glendix: A Plan9/Linux Distribution
Anant Narayanan, Shantanu Choudhary, Vinay Pamarthi and Manoj Gaur.....1

Upperware: Pushing the Applications Back Into the System
Gorka Guardiola, Francisco J. Ballesteros and Enrique Soriano.....9

Scaling Upas
Erik Quanstrom.....19

Vidi: A Venti To Go
Latchesar Ionkov.....25

Inferno DS : Inferno port to the Nintendo DS
Salva Peiro.....31

9P For Embedded Devices
Bruce Ellis and Tiger Ellis.....39

Mrph: A Morphological Analyzer
Noah Evans.....43

Semaphores in Plan 9
Sape Mullender and Russ Cox.....53

v9fb: A Remote Framebuffer Infrastructure for Linux
Abhishek Kulkarni and Latchesar Ionkov.....63

Glendix: A Plan9/Linux Distribution

Anant Narayanan
Shantanu Choudhary
Vinay K. Pamarthi
Manoj S. Gaur

Malaviya National Institute of Technology, Jaipur, India

ABSTRACT

We describe our approach of bringing the Plan 9 userspace to the Linux kernel in order to spread the use of Plan 9 tools amongst the Linux developer community.

1. Introduction

GNU/Linux is a popular free operating system in use today. GNU/Linux strives to be strictly compliant with POSIX standards, and is thus tied down with several requirements and thereby ceases to be innovative as far as operating system design is concerned. Plan 9 [1], on the other hand, was designed to be a from-scratch successor to UNIX. The Plan 9 operating system offers several new features that are very compelling to a developer in today's era of of personal computing.

The Plan 9 kernel, however, supports only a bare minimum of hardware. That is one of the primary reasons of its unpopularity for day-to-day use. The Linux kernel, on the other hand, has had years of development behind it, and enjoys the support of several hardware components and developers alike.

We propose Glendix, a general purpose operating system that aims to combine the Plan 9 userspace with the Linux kernel, to offer today's developer an exciting environment for application development on personal computers and embedded systems alike.

The primary motivating factor here is to promote the Plan 9 style of application development to the large base of developers that Linux already has. A secondary factor is to eliminate the need for *GNU* [2] based userspace software, by replacing them with their lightweight Plan 9 counterparts, which are just as functional and portable. The resulting distribution would be a lightweight Linux based operating system.

In this paper, we describe the approach taken by us to create Glendix. We begin with a review of the different approaches possible, and then describe the chosen methodology, along with significant challenges and how we overcame them. We conclude with a summary of what has been done so far and a few notes on future work.

2. Review

From a broad perspective, there are two kinds of compatibility we can create between programs on Plan 9 and Linux. In this section, we discuss source and binary compatibility, and what they mean in the context of Glendix.

{anant@kix.in,choudhary.shantanu@gmail.com,pamarthi.vinay@gmail.com,msgaur@mnit.ac.in}

2.1. Source compatibility

"Plan 9 from User Space" (also known as *plan9port)* [3] is an existing software package for POSIX compliant operating systems that consists of ports of several Plan 9 applications. While most of Plan 9's libraries have also been ported, the solution is not completely perfect. For example, taking the source for a Plan 9 program and recompiling it using *plan9port* may not result in correctly working binaries all the time.

One of the approaches we reviewed early on during the project was very similar to *plan9port.* The most significant advantage for this approach is that Plan 9 applications can be run on a variety of UNIX clones (not just Linux) after a recompile.

However, this would require us to write POSIX equivalents of all the Plan 9 libraries, which seemed like a step backward. The additional constraint of having to recompile the program for each target environment was not very appealing (what if the sources were not available?), and thus we chose to reject this approach.

2.2. Binary compatibility

A more appealing solution was to achieve binary-level compatibility of all Plan 9 applications. The mantra here was *compile-once-execute-everywhere.* We wanted to ensure that it wouldn't matter where the program was compiled, it should run as expected on both Plan 9 and Linux.

While this approach seems ideal, the Linux kernel provides the capability to support new binary formats, such as Plan 9's *a.out.* In order for this approach to work, we have to make Linux behave exactly as a Plan 9 kernel would, as far as applications are concerned. There are two primary channels for an application to access functionality provided by the Plan 9 kernel: system calls and file servers. If we were to provide suitable implementations of both in the Linux kernel, userspace applications should be oblivious to the fact that the underlying kernel is Linux and not Plan 9, which is exactly what we want.

We decided to adopt this approach because it was interesting and seemed to achieve our stated goals in a clean manner.

3. Methodology

In this section we discuss the implementation details of an *a.out* binary loader for Linux and Plan 9 style system call handling.

3.1. Loader

We will not describe the structure of a Plan 9 executable, which is already documented [4] in *a.out*(5). Linux already supports a variety of executables - ranging from ELF (the native Linux executable format) to COFF. Hence, the foundation for adding support for a new executable format had already been laid, we simply had to use the tools that the kernel offered us.

One of the roles that kernel modules can accomplish is adding new binary formats to a running system, so we chose to write a kernel module for the Plan 9 executable format. The single biggest advantage of writing a kernel module for this purpose is that we didn't have to recompile the kernel and reboot every time we made a change to the loader - thanks to Linux's dynamic module loading/unloading facilities.

Let's take a look at how the *exec* system call is implemented in Linux, because that is central to our objective. The entry point of *exec* lives in the architecture-dependent tree of the source files, but all the interesting code is part of `fs/exec.c`. The toplevel function, `do_execve()`, performs some basic error checking, fills the "binary parameter" structure `linux_binprm` and looks for a suitable binary handler. The last step is performed by a seperate function `search_binary_handler()`, The function finds

the appropriate binary handler by scanning a list of registered binary formats, and passing the `binprm` structure to all of them until one succeeds. If no handler is able to deal with the executable file, the system call returns the ENOEXEC error code.

Linux is also compatible with the standard Unix behavior of supporting exectuable text files that begin with `#!`. Such files are executed with the help of an intepreter which is specified immediately after the `#!` symbol. For this purpose, a binary format specialized in running interpreter files (`fs/binfmt_script.c`), is included. The function is designed to be reentrant, and `binfmt_script` checks against double invocation. The ability to invoke an intepreter in a binary format handler helps us greatly, as we shall see later.

3.2. Binary format handling

As mentioned before, Linux offers the ability to register new binary formats at runtime. The implementation is quite straightforward, although it involves working with rather elaborate data structures – either the code or the data structures must accommodate the underlying complexities; elaborate data structures offer more flexibility than elaborate code.

The core of a binary format is represented in the kernel by a structure called `linux_binfmt`, which is declared in the `linux/binfmts.h` file:

```
struct linux_binfmt {
  struct linux_binfmt *next;
  long *use_count;
  int (*load_binary)(struct linux_binprm *, struct pt_regs *);
  int (*load_shlib)(int fd);
  int (*core_dump)(long signr, struct pt_regs *);
};
```

The three methods declared by the binary format are used to execute a program file, to load a shared library and generate a core dump, respectively. The `next` pointer is used by `search_binary_handler()`, while the `use_count` pointer keeps track of the usage count of modules. Whenever a process *p* is executing in the realm of a modularized binary format, the kernel keeps track of `use_count` to prevent unexpected removal of the module.

Of the three methods, we only need to implement `load_binary`. `load_shlib` is not required as all Plan 9 binaries as statically linked, and `core_dump` is mainly used to generate core dumps readable by the GNU debugger (which we do not want to use).

The binary format handler receives two important parameters by the kernel. The first contains a description of the binary file and the second is a pointer to the processor registers. The first argument, a `linux_binprm` structure, contains, in addition to other fields, the first 128 bytes of the binary file (which enable us to quickly check the *magic* number, and decide if we want to execute this binary or not). We also get addresses of the data pages used to carry around the environment and argument list for the new program.

3.3. Memory layout and padding

Once we've confirmed that the given executable is indeed an *a.out* file, we begin to load its contents into memory. The layout in memory is described in detail in *a.out*(5) but take note of the fact that the in-memory representation of a binary does *not* match with that of the contents of the file. There is a gap between the TEXT and DATA sections in memory, because of page-alignment. In the executable file, however, all sections are one after the other, so while copying the contents into memory we need to create this extra padding.

This was our first major challenge. We noticed that all of the binary formats Linux

supports, actually do contain the padding in the file itself, and therefore, all their handlers use the (in)famous `mmap()` call to directly map the file to memory. We cannot use that approach because `mmap()` does not work on non page-aligned offsets, and the DATA section is bound to be at such an address in the file.

As a workaround, we use Linux's interpreter capabilities (discussed earlier) to invoke a userspace program whenever an authentic a.out executable is found. This userspace program creates this extra padding in the file itself, which may then be memory-mapped. This padding program also turned out to be extremely useful in later stages of the project, as will be discussed in the next section.

3.4. Top of Stack

The statement that system calls are the only way for Plan 9 userspace applications to interact with the kernel is not entirely true. The Plan 9 kernel initializes and maintains a special structure called `Tos`, which is also used to exchange data between the kernel and userspace:

```
struct Tos {
        struct                  /* Per process profiling */
        {
                Plink *pp;      /* known to be 0(ptr) */
                Plink *next;    /* known to be 4(ptr) */
                Plink *last;
                Plink *first;
                ulong pid;
                ulong what;
        } prof;
        uvlong cyclefreq;
        vlong kcycles;
        vlong pcycles;
        ulong pid;
        ulong clock;
        /* top of stack is here */
};
```

As you can see, there are several fields important for process profiling, which need to made available when a binary in executed. The Plan 9 kernel initializes this area above the userspace stack and stores the address in the accumulator, from which userspace applications retrieve and store it in a global variable `_tos`. This is done by all programs linked with *libc.* Linux, however, resets the accumulator immediately after the loader finishes (to signal the return value of *exec*), so we can't use that register to notify userspace applications of the `Tos` address.

As a workaround, we used the padding program described in the previous section, to mangle the instruction that fetched the address from EAX and changed it to fetch the address from EBX instead (Linux does not modify EBX in any way between the loader's end and the program's beginning). The opcode for the MOV instruction is `0x89`. The first instruction in a typical Plan 9 userspace application, therefore, would usually be:

```
89 05 xx xx xx xx
```

where 'xx xx xx xx' denotes a 32-bit address corresponding to the global variable `_tos` in the DATA section.

We change this instruction to:

```
89 1D xx xx xx xx
```

in accordance with x86 opcode table [6] for MOV:

		r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
Address	Mod	R/M	Value of ModR/M Bytes (In Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[-][-]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F

3.5. System call handler

Once the loader had been written, the next major task was to be able to intercept system calls. In Linux, system calls are invoked using the 0x80 interrupt, which raises the programmed exception with that vector. The calling process passes the system call number to identify the required system call in the EAX register. The kernel saves the contents of most registers in the kernel mode stack, hence other parameters to the system call (if required) are placed on subsequent registers. The handler is exited when the system call finishes, and the registers are restored. The return value of the system call is placed in the accumulator, where it is picked up by the calling process. An example of a 'Hello World' program in pure assembly for Linux is provided for clarity:

```
section .data
        hello: db 'Hello World!', 10
        helloLen: equ $-hello
section .text
        global _start
_start:
        mov eax, 4
        mov ebx, 1
        mov ecx, hello
        mov edx, helloLen
        int 80h
        mov eax, 1
        mov ebx, 0
        int 80h
```

Thankfully, the method of system call invocation in Plan 9 is not very different from what is described above. The only two big changes are: a) Plan 9 uses programmed exception vector 0x40 to notify the kernel, and, b) Plan 9 applications store arguments for the system call on the userspace stack, just like for any other function call. An example program for Plan 9 will make the differences clear:

```
DATA    string<>+0(SB)/8, $-"Hello0
GLOBL   string<>+0(SB), $8
TEXT    _main+0(SB), 1, $0
MOVL    $1, 4(SP)
MOVL    $string<>+0(SB), 8(SP)
MOVL    $7, 12(SP)
MOVL    $-1, 16(SP)
MOVL    $-1, 20(SP)
MOVL    $51, AX
INT     $64
MOVL    $string<>+0(SB), 4(SP)
MOVL    $8, AX
INT     $64
```

Unfortunately, the Linux kernel was not built to support the interception of different interrupt vectors in a kernel module. The initialization is done at boot time, hence, for this part of the project, we had to directly edit the kernel source (as opposed to a module as done for the binary format loader).

`arch/x86/kernel/traps_32.c` is where programmed exception gates are created. The routine `set_system_gate()` is provided by the kernel to set an interrupt service routine (ISR) for a particular exception vector. We used that function to set a gate for interrupt vector 0x40. As for the ISR, we copied the same routine as for interrupt vector 0x80, with the exception of calling a custom system call implementation in the end: `sys_plan9()`, irrespective of the system call number in the accumulator. The ISR copies the register values to the kernel stack as usual, and triggers `sys_plan()` with appropriate arguments. We use the value of the `EBP` register to obtain the stack pointer in userspace and extract system call arguments using the `__get_user()` routine provided by Linux. These arguments are in turn passed to an internal system call implementation. Sometimes this means calling an existing Linux system call, but in many cases, we had to write one from scratch (eg: `sys_fd2path)`. A snippet of the `sys_plan9` function is as follows:

```
asmlinkage long sys_plan9(struct pt_regs regs) {
  .
  /* retrieving arguments from userspace stack */
  unsigned long *addr = (unsigned long *)regs.esp;
  /* check syscall number and invoke */
  switch (regs.eax) {
    .
    case 51: /* pwrite */
      arg1 = *(++addr);
      arg2 = *(++addr);
      arg3 = *(++addr);
      addr = addr + 2;

      offset = (loff_t) *(addr);
      if (offset == 0xffffffff)
        retval = sys_write(arg1, (const char __user*)arg2, arg3);
      else
        retval = sys_pwrite64(arg1, (const char __user*)arg2, arg3, offset);

      break;
  }
}
```

4. Conclusion

By implementing 15 of the 39 system calls, we got a surprising number of applications to run. Examples include *8c, sed, grep, echo, cat, tar, cb, cal* and *dc,* among others. We believe that on completing all the system calls, Glendix will provide an excellent base for developers to start writing applications on Linux in the "Plan 9 way". The ability to run unmodified binaries in both operating systems is not provided by any other existing alternative, with the exception of 9vx (which is discussed in the appendix). The performance of these binaries will be the same as other native linux binaries because all the supporting infrastructure is built directly into the kernel.

Glendix, at this stage, serves as proof of concept that ideas from the Plan 9 system can be integrated into the Linux kernel. However, in order to achieve the goal of providing a complete ''Plan 9 experience'' to application developers, there is a lot more to be done, which is discussed in the following section.

5. Future Work

While most of the system calls from Plan 9 map more or less directly to their Linux counterparts, some features are unique Plan 9. Process and address space management along with per-process namespaces are the two most important aspects that affect the implementation of system calls.

Recently, the Linux kernel added support for per-process namespaces via the `CLONE_NEWNS` flag for its `clone` system call. Hence, Linux already contains primitives for namespace manipulation, even if they are not exposed to userspace applications directly. We believe that system calls such as `mount` and `bind` can be implemented using primitives already provided by the Linux kernel, and indeed, we are already working on them. `rfork`, on the other hand, is a little trickier, especially because of the specific combination of the `RFMEM` and `RFPROC` flags; which results in the creation of a new process sharing everything with its parent, except for the stack. For this particular permutation, it will be neccessary to dig deeper into the memory management primitives provided by the Linux kernel, but is entirely possible. In fact, since we are dealing with kernel code here, anything is technically possible, the only variation amongst the different system calls is the amount of code to be changed and/or written.

The other major feature to be emulated is that of the synthetic file systems provided by the Plan 9 kernel. Since Linux already supports such file systems (atleast partially - examples are `/proc` and `/sys`), we think it will not be hard to extend this to true Plan 9 filesystems such as `/net`. `/dev/draw` can be built on top the native Linux framebuffer device.

Once we implement all the system calls and synthetic file systems correctly, there should be no perceivable difference between the Glendix kernel and a Plan 9 kernel as far as an application is concerned. Source code and other details pertaining to the project are available on `http://glendix.org/`. Developers are encouraged to participate!

Acknowledgements

This project was born from earlier open source projects, so we would like to begin by thanking the Plan 9 and Linux communities for giving us such great software and support to work with. Specifically, we would like to thank Charles Forsyth, Russ Cox, Rene Herman and Al Viro, who contributed significantly to the project by offering their insightful comments, suggestions and help.

Major portions of Glendix were executed as a final term project at the Malaviya National Institute of Technology. We would like to thank Dr. Vijaylaxmi for her timely feedback and suggestions.

References

[1] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''Plan 9 from Bell Labs'', Computing Systems, **8**, 3, Summer 1995, pp. 221-254

[2] ''GNU's Not Unix'', http://www.gnu.org/

[3] Russ Cox, ''Plan 9 from User Space'', http://swtch.com/plan9port/

[4] ''Plan 9 Programmer's Manual'', http://plan9.bell-labs.com/sys/man/

[5] Alessandro Rubini, ''Playing with binary formats'',

http://www.linux.it/~rubini/docs/binfmt/binfmt.html

[6] ''Intel 64 and IA-32 Architectures Software Developer's Manual'', volume 2A

[7] Bryan Ford, Russ Cox, ''Vx32: Lightweight User-level Sandboxing on the x86'', USENIX Annual Technical Conference, Summer 2008.

Appendix: Comparison to 9vx

Vx32 [7] is a user-mode library that was recently developed at CSAIL, MIT. The primary purpose of the library is to provide a safe and portable execution environment for untrusted x86 code. One of the interesting applications of this is the ability to run Plan 9 executables on all platforms that Vx32 supports (currently FreeBSD, Linux and Mac OS X). 9vx is the project that uses Vx32 to run an instance of the Plan 9 system.

On the surface, it may seem like the outcomes of 9vx on Linux and Glendix are similar, but there are many important differences. Vx32 can be compared in a very rough sense to a virtual machine, and thus there is a disjoint between the binaries running inside it, and the operating system it runs on. Glendix, however, aims to provide a more close coupling between Plan 9 applications and the Linux kernel, whether you trust the executables or not. Secondly, Vx32 is restricted to x86 binaries only. While this paper discusses only the x86 implementation of Glendix, we can easily extend it to cover other architectures as well, given the cross-platform nature of both Plan 9 binaries and Linux.

Upperware: Pushing the Applications Back Into the System

Gorka Guardiola, Francisco J. Ballesteros, Enrique Soriano

Rey Juan Carlos University, Spain
{nemo,paurea,esoriano}@lsub.org

ABSTRACT

It is quite difficult and tedious to share devices among different operating systems. If we also want to share other resources, like the state of a web browser or an editor, it becomes next to impossible.

Similar problems are solved inside Plan 9 [14] and Inferno [4] by using the 9P protocol [9]. The normal approach, though, is to write an application or a device driver providing a filesystem interface. Our problem is somewhat different. We already have native applications like Word or Firefox. How can we use these applications, native to several operating systems, and at the same time have the ease of communications provided by 9P?.

In this paper we propose a simple way to do it: Wrap the applications and drivers with a controlling filesystem running on Inferno, hosted on the relevant machines. Then, export and share the filesystems, exporting them even to the local host system through some protocol it understands. Without much configuration, the user can print and read documents simply by using drag and drop at any of the involved machines. We propose the name *upperware* for this approach, which tries to abstract applications instead of the underlying system.

1. Introduction

The idea of Octopus [3] is to centralize the state of the applications in a computer that we call the PC. Then the user can run the terminal software which exports local resources as filesystems to applications running on the PC. Local resources may be applications and devices running at the terminal. In order to integrate these applications with the rest of the system we had to wrap them with filesystems, aggregating some attributes to them so they could be selected automatically. We have found that this approach is very simple yet powerful and lets users share resources easily without much configuration. We call it *upperware*. By proceeding further and exporting the resulting name space back to the underlying host OS, we reach a portable way to integrate the heterogeneous terminal, in a transparent way. By carefully thinking the filesystem interfaces, so that they can be used by copying files, we can convert this approach into a general solution even for non programmer users, which can do most things by drag-and-drop.

This work supported in part by Spanish TIN-2007-67353-C02-02 and CAM S-505/TIC/0285.

We have applied the *upperware* principle by writing *'device*drivers' for high level applications and services available on various systems we use. This includes printing, document viewers, voice synthesis, user activity monitoring, and a web browser service (still underway). In this article we describe such 'device drivers', their interface and what can be gained by using upperware: seamless communication and easy of use across heterogeneous platforms. We feel that upperware can be useful in general to integrate different platforms, taking the place of typical object-oriented middleware (OOM) approach.

Note that unlike in middleware systems, we wrap the namespace and make it available to the underlying system (at each terminal) in a transparent way. For the host the name space is just another file volume. However, it provides all sort of services for the user. Being natively available, native applications are able to use such name space as the user sees fit.

2. Organization

Experience with Plan 9, Inferno and Plan B [2] taught us that exporting devices and applications as synthetic filesystems makes it easy to integrate them into a distributed operating system. Applications like `acme(1)` [8] and `rio(1)` benefit greatly from programability by exposing a synthetic filesystem.

But our problem and approach is not quite the same, even if we still want to expose software resources from the underlying operating system and integrate them into a name space. Inferno does this to some extent with its devices while running hosted. The main difference is that Inferno places itself *side by side* to the host operating system, that is, it provides its own distinct virtual platform. Instead, we try to place ourselves above all the software running natively, meaning that we try to take advantage of the underlying operating system as much as possible, including also some of the applications.

What we are trying to accomplish is similar in approach to what Inferno does with the underlying filesystem of the host and to what 9vx does with the TCP/IP stack. We are trying to wrap software resources with a filesystem but using all the mechanisms provided by the host to ease programability and to interact with the user when necessary.

When trying to wrap underlying applications, we have to be careful to distinguish between two types of interfaces (meaning the semantics we assign to the filesystem): *passive* and *active*:

1.Passive interfaces wrap devices that are similar to low level devices. They do not require interaction with the user via the underlying system. In a sense they are data sinks and sources. Examples of this are the Octopus voice synthesis and printer devices. The fact that they do not require interaction with the user is very important, because it places restrictions on the way they should behave. For example, if the user imports a printer from a nearby computer and sends something to print, she does not want the remote host OS to ask her what paper size to use in the remote computer. Instead, it is more convenient if the file system interfaces provides an agreed-upon standard configuration and some means to change it.

2.Active interfaces wrap applications which do require interaction with the user. They are normally conduits of data or, at least, control the data flow. Examples of this kind of interface are editors and web browsers. An editor takes some file, changes it and puts

it somewhere else as dictated by the interaction with the user. This interaction happens in the underlaying host system of the computer providing the service.

Orthogonal to this classification, but also important, is that for some of these devices it may be desirable to keep their state across computers and/or sessions, which affects the implementations of the servers to be provided (e.g., editors would copy the files, to remain autonomous).

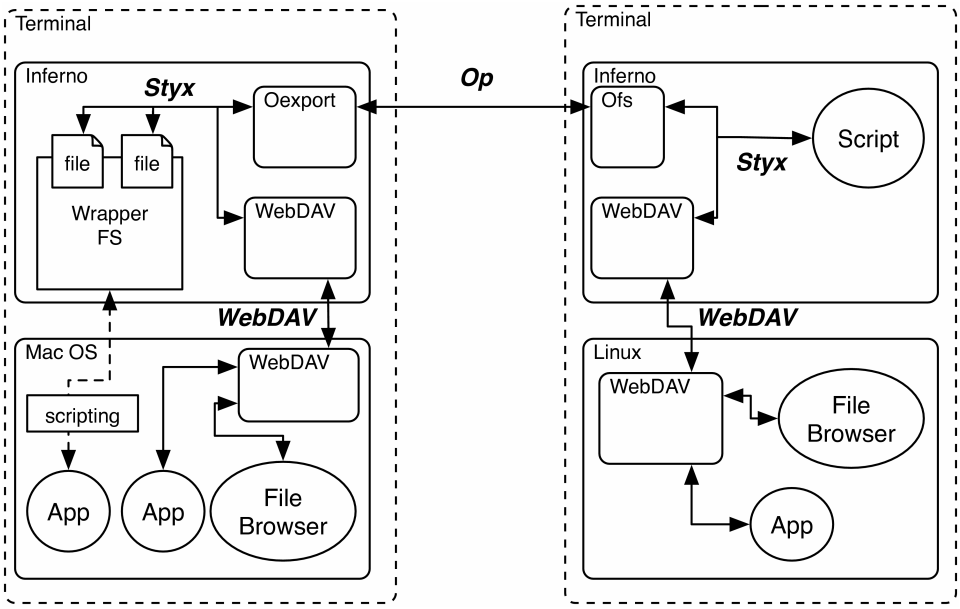


Figure 1: *Upperware organization.*

The filesystems we share in this fashion need some attributes to let the user select them appropriately by their location, operating system, etc. By convention, we add a file named `ndb` to the root of each filesystem. This file contains a list of attribute/value pairs describing the system exported. Such information is also kept at a central registry, for the user to look. In this way, the filesystem carries within itself its own description, should the user need to know.

3. Spooling

One thing we have found while trying to devise ways to expose and interact with the synthetic filesystems, is that *spooling* is a powerful strategy. By *spooling* we do not refer just to the classical spooling interface, because for us the directory doing the spooling is synthetic and some magic may occur in it. Nevertheless, the main idea remains the same: Files are copied to a directory and things happen to them when their turn comes. The resulting mechanism is similar to Apple droplets [1], as far as the user is concerned. Copying a file to a directory triggers an action.

Spooling also provides a way to interact with the filesystem through the host OS. Spooling directories are exported back to the host system using, for example, WebDAV. A simple drag and drop may be used to print a file or to edit it, hiding to the end user the details of the hosted system providing the mechanism.

Besides, spooling provides a way to structure the software. A whole filesystem does not need to be implemented for each different service (device or application); just some functions implementing the action performed to the files inside the spooler.

In the current implementation there is a portable module implementing the spooling filesystem. For each different *spooler*, an architecture dependant implementation of a spooler module provides an interface to the actual service. This interface has three operations, `start` `stop` and `status`. When a file has been copied (detected by the *clunk* on a fid open for writing) `start` is called. In some circumstances, reading a file also triggers a call to `start`. This makes the operation to be performed automatically when a file is copied to the directory. The `stop` operation gets called when a file is removed from the directory. It provides a mean for the user to abort or cancel a spooling on progress.

Using this simple scheme, we have a portable interface which can be used right from the file browsers provided by most (all?) host operating systems as well as a simple way to implement new servers and a straightforward user interface, well known by any user no matter the system employed.

Two example of spooling servers, discussed next, are the *view* and *print* devices.

3.1. View

View is an implementation of the spooler interface, i.e., provides a spooling device for viewing files and documents. It relies on the generic *open* command of the host OS to open the relevant file when its `start` function is called. For example, the device uses `gnome-open` in Linux, `open` in MacOS, and the plumber on Plan 9.

The device is intended for reading documents, like PDF or similar formats, in a passive way, although it is a little more general than that. For example, copying an MP3 file would reproduce it on the default player and, in a similar way, can be used to display images, play video files, open documents and web pages, etc. But note that edition of the browsed files is not supported by the device.

3.2. Print

Print is the interface to the printer system of the host OS. It is also a spooler module. A filed copied onto it is printed like in an old fashion printing spooler system. At the moment it uses the (CUPS) lpr commands in Mac OS and Linux, and lp in Plan 9. It prints just to the default printer with the default options, without any possibility of configuration. There has also been an ad-hoc printer module for Windows.

Even though this module is quite naïve, it has proved to be highly useful. Using it, it becomes trivial to do things like print to the closest printer (selected automatically via the ndb) from different locations and different operating systems.

4. Voice

This is a simple device, not based on the spooler. *Voice* exports mostly a file (apart from the `ndb`) which can be written to. Any text written into it is synthesized as voice. This is used mostly for messages of the system, but it can also be used for messages among users, like on Plan B.

Used with care (of not annoying the user by frequent messages), it is a useful complement for other interfaces to the system. For example, long termed commands, meaning those that do not complete after several seconds, generate a complete voice message to

reassure the user that the command has finished.

In Mac OS we use a dynamically generated AppleScript script. For Linux we use *esspeak*. On Plan 9 we rely on the Plan B device for voice synthesis, which is actually relaying the work to any near-by Linux machine equipped with Festival.

5. Idle

In order to do things automatically, our system needs to know if the user is idle, meaning if he is using the terminal or not. Of course, we also need to know *which* terminal has been used last, to locate the user. This is done by employing some heuristics on data collected from several interactive applications and from the system I/O statistics. The *Idle* device is responsible for collecting such data and updating agreed-upon files describing the activity of the user on all her terminals. The rest of the system relies only on the portable files (and events) and does not need to be concerned about platform-specific details.

6. Web

We are currently developing a device for the web browser, *browserfs.* The first prototype offers three synthetic files to pull data from the web browser (`open`, `history`, and `bookmarks`) and a `ctl` file to push data and perform basic operations on the browser. All pull files are read-only and exclusive-open.

The three pull files provide their data in a canonical format. Different browsers use different formats to store data (mainly XML), but this is hidden by the provided interface. Browserfs offers the information using plain text files, in order to make it easy for humans to read it and for programs to transform it.

When a pull file is opened, the device retrieves the corresponding data from the host's browser by executing certain native programs on the host.

The *open* file offers the list of URLs that are currently opened in browser's windows and tabs, one per line. It corresponds to the pages being viewed presently.

History provides the last 100 entries in the browser's history record. Each line is formed by the date of the entry (seconds since epoch), the URL for the entry, and the title of the HTML page, separated by blanks. This format is easily tokenizable, because the date and the URL cannot include blanks.

The *bookmarks* file provides the list of bookmarked pages, as contained in a particular folder in the browser interface. The user is in charge of creating any new bookmarks in this folder to make them available from all his octopus terminals. This way, the user is able to select which bookmarks are shared among terminals and which ones are not. As it could be expected, the format of the *bookmarks* file is simply one bookmark per line, formed by the URL and its description.

The *ctl* file is used to push data and to perform control operations on the terminal's browser. So far, it implements only two commands: *open* and *close.* The *open* command executes the browser if it is not yet executing, makes a new window, and opens the given URLs in tabs. For example:

```
echo open http://google.com http://lsub.org > /term/browser/ctl
```

The *close* command forces the browser to close its windows and quit.

Together with *browserfs* we provide several scripts to capture and recreate the state of the browser.

The script *bookmarks.sh* reads the bookmarks from the user's terminals (i.e. */pc/terms/*/browser/bookmarks*) and merges them on a single file, that is automatically opened by the browser of the terminal in which it is executed. The same is done by *history.sh* for history entries, offering a HTML file with entries ordered by date.

Although this simple approach works for providing the web browsing state to the user, it is not enough to create a full illusion of using the same browser at all the terminals. Several extra control operations to update the browser's idea of bookmarks and history would be needed (not to talk about cookies).

In any case, the current version as described provides a portable implementation.

7. Mobile devices

For some devices it may be desirable for their state to follow the user. For example, we might want to keep the set of web pages being viewed the same, no matter the terminal. For other devices or terminals we may not want this to happen. Only the user knows the appropiate thing to do.

We try to provide this facility in a simple way, leaving it up to the user the choice of when and for which services it is to be employed. The overall scheme is described next, but we it is to be noted that this facility is still in the early stages and is not yet available.

We will try to keep this description concrete to the example of the browser, though the ideas are easily generalizable. We are currently working on this implementation of the browser. Before the user turns off a terminal, a shell script, *dump.sh,* is executed. This script stores the data of the terminal's browser in real files, in a well-known directory of the PC. When the users turn on a terminal, another script, *restore.sh,* is executed. This script reads the files in the well-known directory and recreates the state in the terminal's web browser. Another script, *followme.sh,* can automatically dump and restore data when the user location changes (i.e. he moves from one office to another). We will experiment with *browserfs* in order to create common policies for other application wrappers.

8. Implementation

The implementation of the machine dependant modules is a mixture of shell scripts, applescripts, C and whatever the host system might provide to do the job at hand.

Some of the operations that have to be done in the underlying system are trivial to implement, while other are annoying. It all depends on the interface offered by the application considered and the set of tools available.

For example, Applescript on Mac OS X provides a reasonable interface to control most applications. For instance, the Safari API provides operations to deal with tabs, URLs, and execute Java Script over a document, so some operations are easy to implement. On the other hand, Safari does not contemplate bookmark and history manipulation by third parties, so we have to deal with Mac OS property list (XML) files by hand.

We try to keep the number of features to a minimum so that the implementation is easy to write for all the operating systems involved as hosts for the Octopus. In the cases in which we can avoid most of the interaction with the native applications by relying on files instead, we do so.

This way, it is easier to keep up with the idiosyncrasies of the different systems, versions and applications involved. We also try to use the default or most popular

applications of the system if there is more than one. For example, in Linux we are sticking whenever possible to applications which come by default with Ubuntu and Gnome.

To give a taste of how simple it is to write upperware for applications in Mac OS we will give the SLOC for the relevant parts in the Octopus. Most of the code is Limbo, with some scripts and applescripts generated at run time for the MacOS dependant part.

Tables 1 and 2 show the lines of code needed to implement the wrappers themselves, including the spooler (portable and Mac OS modules, respectively). Table 3 shows the lines of code needed to implement the Mac OS scripts of the current version of *browserfs.* Table 4 shows the lines needed to implement the infrastructure to be able to reexport the fileservers to the local host and to select the appropiate tree automagically.

lines	module
288	*browserfs.b*
277	*spool.b*
110	*view.b*

Table 1: *Lines of code, portable modules.*

lines	module
94	*idle.b*
159	*mbrowser.b*
82	*print.b*

Table 2: *Lines of code, Mac OS modules.*

lines	program
269	*browserfs.scpt (applescript)*

Table 3: *Lines of code, host commands.*

lines	program
1662	*webdav*
177	*watcher*
751	*mux*

Table 4: *Lines of code, infrastructure to reexport the fileservers to the local host.*

9. Related work

Middleware commonly sits between the application and the operating system and tries to abstract the operating system. We are trying to do the opposite, and abstract the application to make it available to the system (i.e., to other systems).

We are by no means the first ones to wrap local devices of the host OS and to export

them using a lower level API. Npfs [6] and 9vx [5] wrap the local TCP/IP stack, for example with a filesystem. Inferno, `drawterm`, and many virtual machines like VMWare [12] [13] are able to export the local filesystem and some devices by using the host os. This is a common technique used in paravirtualization. The main difference is that we are trying to wrap high level devices and applications.

Filesystem like clients like `ftpfs(4)` or `sshfs(4)` try to wrap servers using filesystems as clients. The idea is similar to what we are trying to achieve, but they are remote servers. Here the application would be the client and it is built into the filesystem. We are wrapping local applications instead, also with a filesystem, in order to export them. In a sense, it is the reverse strategy.

Some early rudimentary attempts have been made before to export applications like a browser using plumber and ssh see [7]. As far as we know no one has tried wrapping them with a full fledged filesystem, which is much more rich in its interface and possibilities.

Regarding web browsing, there are utilities to manage and share bookmarks, such as Del.icio.us [10]. There are also programs to merge and translate bookmark files from different browsers [11]. None of these systems provide mechanisms to automatically recreate the state of the browser in different machines which is finally our goal. Also, they are meant just for a single particular service (web browsing) and not for all other tools needed by a computer user.

10. Conclusions

Our experience with the Octopus has made us realize how easy can it be to use applications once they are wrapped up and integrated back into the system interface. Something as simple as the print device which accounts for less than 500 lines of code including the spooler filesystem enables the user to print from Linux to Mac OS and viceversa (something that, perhaps surprisingly, we could not do due to incompatibilities between different native systems involved). It is amazing that similar issues are still problems in practice due to different configurations, security, and version issues. With upperware this can be accomplished smoothly keeping the user oblivious to the magic glue hiding behind the scene. The same goes for most other devices and services.

11. Future work.

More applications are to be wrapped to have a complete set. The web browser prototype has been written for Safari, but there is not yet supported for the other mainstream operating systems. An editor and a music player filesystem need to be written as well.

As of today, the spooler does not listen to messages from the application to detect when the user is done with a file (e.g., when a view window has been closed). Some infrastructure is already available for this, but it is not being used. The problem is that this would suppose a more intimate relationship with the native application, making it harder to write/port the machine dependent part of the service.

Support for Windows is also needed, but none of the Octopus developers use Windows daily and, although basic services are available thanks to the portability of Inferno, much remains to be done.

References

1. Apple, Shiny Droplets, http://www.apple.com/downloads/macosx/productivity_tools/shinydroplets.html, 2008.
2. F. J. Ballesteros, G. Guardiola, K. L. Algara, E. Soriano, P. H. Quirós, E. M. Castro, A. Leonardo and S. Arévalo, Plan B: Boxes for network resources, *Journal of the Brasilian Computer Society. Special issue on Adaptable Computing Systems. To appear. Also in http://lsub.org/ls/export/box.html*, 2004.
3. F. J. Ballesteros, P. Heras, E. Soriano and S. Lalis, The Octopus: Towards building distributed smart spaces by centralizing everything., *UCAMI*, 2007.
4. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey and P. Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal 2*, 1 (1997), .
5. B. Ford and R. Cox, Vx32: Lightweight, User-level Sandboxing on the x86, *USENIX*, 2008.
6. Npfs, Npfs project, http://sourceforge.net/projects/npfs, 2007.
7. R. pike, Message by rob pike in 9fans: My web browsing technique, http://9fans.net/archive/2002/11/529, 2003.
8. R. Pike, Acme: A User Interface for Programmers, *Proceedings for the Winter USENIX Conference*, 1994, 223-234. San Francisco, CA..
9. D. Presotto and P. Winterbottom, The Organization of Networks in Plan 9, *Plan 9 User's Manual 2*.
10. J. Schachter, Del.icio.us, http://del.icio.us, 2003.
11. E. Software, Bookit, http://everydaysoftware.net/bookit, 2003.
12. J. Sugerman, G. Venkitachalam and B. H. Lim, Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *USENIX Annual Technical Conference*, 2001, 1-14.
13. VMWare, VMWare, *http://www.vmware.com*, 2001.
14. Plan B User's Manual. Second edition., *Laboratorio de Systemas, URJC. GSYC-Tech. Rep.-2004-04.*, 2004.

Scaling Upas

Erik Quanstrom
quanstro@coraid.com

ABSTRACT

The Plan 9 email system, Upas, uses traditional methods of delivery to UNIX® mail boxes while using a user-level file system, Upas/fs, to translate mail boxes of various formats into a single, convenient format for access. Unfortunately, it does not do so efficiently. Upas/fs reads entire folders into core. When deleting email from mail boxes, the entire mail box is rewritten. I describe how Upas/fs has been adapted to use caching, indexing and a new mail box format (mdir) to limit I/O, reduce core size and eliminate the need to rewrite mail boxes.

1. Introduction

> Chained at his root two scion demons dwell
> – Erasmus Darwin, The Botanic Garden

At Coraid, email is the largest resource user in the system by orders of magnitude. As of July, 2007, rewriting mail boxes was using 300MB/day on the WORM and several users required more than 400MB of core. As of July, 2008, rewriting mail boxes was using 800MB/day on the WORM and several users required more than 1.2GB of core to read email. Clearly these are difficult to sustain levels of growth, even without growth of the company. We needed to limit the amount of disk space used and, more urgently, reduce Upas/fs' core size.

The techniques employed are simple. Mail is now stored in a directory with one message per file. This eliminates the need to rewrite mail boxes. Upas/fs now maintains an index which allows it to present complete message summaries without reading indexed messages. Combining the two techniques allows Upas/fs to read only new or referenced messages. Finally, caching limits both the total number of in-core messages and their total size.

2. Mdir Format

In addition to meeting our urgent operational requirements of reducing memory and disk footprint, to meet the expectations of our users we require a solution that is able to handle folders up to ten thousand messages, open folders quickly, list the contents of folders quickly and support the current set of mail readers.

There are several potential styles of mail boxes. The maildir[1] format has some attractive properties. Mail can be delivered to or deleted from a mail box without locking. New mail or deleted mail may be detected with a directory scan. When used with WORM storage, the amount of storage required is no more than the size of new mail received.

Mbox format can require that a new copy of the inbox be stored every day. Even with storage that coalesces duplicate blocks such as Venti, deleting a message will generally require new storage since messages are not disk-block aligned. Maildir does not reduce the cost of the common task of a summary listing of mail such as generated by acme Mail.

The mails[2] format proposes a directory per mail. A copy of the mail as delivered is stored and each mime part is decoded in such a way that a mail reader could display the file directly. Command line tools in the style of MH[3] are used to display and process mail. Upas/fs is not necessary for reading local mail. Mails has the potential to reduce memory footprint below that offered by mdirs for native email reading. However all of the largest mail boxes at our site are served exclusively through IMAP. The preformatting by mails would be unnecessary for such accounts.

Other mail servers such as Lotus Notes[4] store email in a custom database format which allows for fielded and full-text searching of mail folders. Such a format provides very quick mail listings and good search capabilities. Such a solution would not lend itself well to a tool-based environment, nor would it be simple.

Maildir format seemed the best basic format but its particulars are tied to the UNIX environment; mdir is a descendant. A mdir folder is a directory with the name of the folder. Messages in the mdir folder are stored in a file named *utime.seq*. *Utime* is defined as the decimal UNIX seconds when the message was added to the folder. For the inbox, this time will correspond to the UNIX "From " line. *Seq* is a two-digit sequence number starting with `00`. The lowest available sequence number is used to store the message. Thus, the first email possible would be named `0.00`. To prevent accidents, message files are stored with the append-only and exclusive-access bits turned on. The message is stored in the same format it would be in mbox format; each message is a valid mbox folder with a single message.

3. Indexing

When upas/fs finds an unindexed message, it is added to the index. The index is a file named *foldername*.idx and consists a signature and one line per MIME part. Each line contains the SHA1 checksum of the message (or a place holder for subparts), one field per entry in the *messageid/info* file, flags and the number of subparts. The flags are currently a superset of the standard IMAP flags. They provide the similar functionality to maildir's modified file names. Thus the 'S' (answered) flag remains set between invocations of mail readers. Other mutable information about a message may be stored in a similar way.

Since the *info* file is read by all the mail readers to produce mail listings, mail boxes may be listed without opening any mail files when no new mail has arrived. Similarly, opening a new mail box requires reading the index and checking new mail. Index files are typically between 0.5% and 5% the size of the full mail box. Each time the index is generated, it is fully rewritten.

4. Caching

Upas/fs stores each message in a `Message` structure. To enable caching, this structure was split into four parts: The `Idx` (or index), message subparts, information on the cache state of the message and a set of pointers into the processed header and body. Only the pointers to the processed header and body are subject to caching. The available cache states are `Cidx`, `Cheader` and `Cbody`.

When the header and body are not present, the average message with subparts takes roughly 3KB of memory. Thus a 10,000 message mail box would require roughly 30MB of core in addition to any cached messages. Reads of the `info` or `subject` files can be satisfied from the information in the `Idx` structure.

Since there are a fair number of very large messages, requests that can be satisfied by reading the message headers do not result in the full message being read. Reads of the `header` or `rawheader` files of top-level messages are satisfied in this way. Reading the same files for subparts, however, results in the entire message being read. Caching the header results in the `Cheader` cache state.

Similarly, reading the `body` requires the body to be read, processed and results in the `Cbody` cache state. Reading from MIME subparts also results in the `Cbody` cache state.

The cache has a simple LRU replacement policy. Each time a cached member of a message is accessed, it is moved to the head of the list. The cache contains a maximum number of messages and a maximum size. While the maximum number of messages may not be exceeded, the maximum cache size may be exceeded if the sum of all the currently referenced messages is greater than the size of the cache. In this case all unreferenced messages will be freed. When removing a message from the cache all of the cacheable information is freed.

5. Collateral damage

> Each new user of a new system uncovers a new class of bugs.
> — Brian Kernighan

In addition to upas/fs, programs that have assumptions about how mail boxes are structured needed to be modified. Programs which deliver mail to mail boxes (deliver, marshal, ml, smtp) and append messages to folders were given a common (nedmail) function to call. Since this was done by modifying functions in the Upas common library, this presented a problem for programs not traditionally part of Upas such as acme Mail and imap4d. Rather than fold these programs into Upas, a new program, mbappend, was added to Upas.

Imap4d also requires the ability to rename and remove folders. While an external program would work for this as well, that approach has some drawbacks. Most importantly, IMAP folders can't be moved or renamed in the same way without reimplementing functionality that is already in upas/fs. It also emphasises the asymmetry between reading and deleting email and other folder actions. Folder renaming and removal were added to upas/fs. It is intended that mbappend will be removed soon and replaced with equivalent upas/fs functionality — at least for non-delivery programs.

Mdirs also expose an oddity about file permissions. An append-only file that is mode `0622` may be appended to by anyone, but is readable only by the owner. With a directory, such a setup is not directly possible as write permission to a directory implies permission to remove. There are a number of solutions to this problem. Delivery could be made asymmetrical—incoming files could be written to a mbox. Or, following the example of the outbound mail queue, each user could deliver to a directory owned by that user. In many BSD-derived UNIX systems, the "sticky bit" on directories is used to modify the meaning of the `w` bit for users matching only the other bits. For them, the `w` bit gives permission to create but not to remove.

While this is somewhat of a one-off situation, I chose to implement a version of the

"sticky bit" using the existing append-only bit on our file server. This was implemented as an extra permission check when removing files. Fewer than 10 lines of code were required.

6. Performance

A representative local mail box was used to generate some rough performance numbers. The mail box is 110MB and contains 868 messages. These figures are shown in table 1. In the worse case—an unindexed mail box—the new upas/fs uses 18% of the memory of the original while using 13% more cpu. In the best case, it uses only 5% of the memory while using only 13% of the cpu. Clearly, a larger mail box will make these ratios more attractive. In the two months since the snapshot was taken, that same mail box has grown to 220MB and contains 1814 messages.

Table 1 - Performance

action	user s	system s	real s	core size MB
old fs read	1.69	0.84	6.07	135
initial read	1.65	0.90	6.90	25
indexed read	0.64	0.03	0.77	6.5

7. Future Work

While Upas' memory usage has been drastically reduced, it is still a work-in-progress. Caching and indexing are adequate but primitive. Upas/fs is still inconsistently bypassed for appending messages to mail boxes. There are also some features which remain incomplete. Finally, the small increase in scale brings some new questions about the organization of email.

It may be useful for mail boxes with very large numbers of messages to divide the index into fixed-size chunks. Then messages could be read into a fixed-sized pool of structures as needed. However it is currently hard to see how clients could easily interface a mail box large enough for this technique to be useful. Currently, all clients assume that it is reasonable to allocate an in-core data structure for each message in a mail box. To take advantage of a chunked index, clients (or the server) would need a way of limiting the number of messages considered at a time. Also, for such large mail boxes, it would be important to separate the incoming messages from older messages to limit the work required to scan for new messages.

Caching is particularly unsatisfactory. Files should be read in fixed-sized buffers so maximum memory usage does not depend on the size of the largest file in the mail box. Unfortunately, current data structures do not readily support this. In practice, this limitation has not yet been noticeable.

There are also a few features that need to be completed. Tracking of references has been added to marshal and upas/fs. In addition, the index provides a place to store mutable information about a message. These capabilities should be built upon to provide general threading and tagging capabilities.

8. Speculation

Freed from the limitation that all messages in a mail box must be read and stored in memory before a single message may be accessed, it is interesting to speculate on a few further possibilites.

For example, it may be useful to replace separate mail boxes with a single collection of messages assigned to one or more virtual mail boxes. The association between a message and a mail box would be a "tag." A message could be added to or removed from one or more mail boxes without modifying the mdir file. If threads were implemented by tagging each message with its references, it would be possible to follow threads across mail boxes, even to messages removed from all mail boxes, provided the underlying file were not also removed. If a facility for adding arbitrary, automatic tags were enabled, it would be possible to tag messages with the email address in the SMTP From line.

9. References

[1]D. Bernstein, "Using maildir format", published online at http://cr.yp.to/proto/maildir.html

[2]F. Ballesteros published online at http://lsub.org/magic/man2html/1/mails

[3]MH Wikipedia entry, http://en.wikipedia.org/wiki/MH_Message_Handling_System

[4]Lotus Notes Wikipedia entry, http://en.wikipedia.org/wiki/Lotus_Notes

[5]D. Presotto, "Upas—a Simpler Approach to Network Mail", Proceedings of the 10th Usenix conference, 1985.

Vidi: A Venti To Go

Latchesar Ionkov
*Los Alamos National Laboratory**
lionkov@lanl.gov

ABSTRACT

Vidi is a Venti proxy that allows certain clients to work when there is no connection to the Venti server. Vidi can be used on computers, such as laptops, to create archives of the file system even when disconnected, and later to transfer the archives to the Venti server. This paper describes an archival configuration used by the author as well as the design and implementation of a proxy that allows it to work in a disconnected state.

1. Introduction

Venti [7] archival server and the utilities for using it, Vac and Vacfs, allow simple and convenient way of keeping history of computer's files forever. Venti's interface doesn't allow data to be deleted or modified once it is stored. The fact that block's address depends on its content allows Venti to coalesce all blocks with the same content and keep a single copy in its storage. The archival utilities that use Venti don't need to implement complex algorithms to detect which files on the filesystem are modified. Archiving multiple filesystems with similar files leads to even better utilization of disk's space. Initially Venti was designed to replace Plan9's WORM [6] filesystem, but with Plan9 from User Space [2] the server and clients are also available for POSIX compatible operating systems.

A common Venti setup consists of a server with many disks, possibly in RAID configuration, and multiple clients in the same network archiving their filesystems daily on the server. This setup doesn't work well when the clients are mobile and can be used disconnected for long periods of time. If the clients cannot connect to Venti, gaps are introduced in history of the filesystem, data might be lost, and some of the important advantages of using Venti no longer exist.

One of the solutions for mobile computers is to run Venti locally, eventually copying the local Venti content to the central Venti server later. A major drawback for this approach is that Venti requires at least 105 percent as much disk space as the data stored. The disk space of the mobile computers is not as abundant as for the servers and desktops and the restriction is very often unacceptable.

Vidi introduces an alternative solution for disconnected archival that only uses 0.5 to 2 percent of the space required by running a local Venti server. Instead of keeping locally the content of all the blocks when making an archive, it keeps only the addresses of the blocks that were already sent to Venti, and the content of the blocks that were written while the Venti server was unavailable. Once the mobile computer is connected to its home network, Vidi copies the blocks to the central Venti server and deletes them from the local disk. A notable disadvantage of using Vidi is that it doesn't allow access to previous snapshots when disconnected.

2. Plan9 dump for Linux

2.1. Venti

Venti is a network storage system that uses a hash value of a block's content as an address for the block. Once data is stored into the Venti storage, it cannot be deleted. Venti provides simple interface for storing and retrieving data. When a client sends a data block for storage,

*LANL publication: LA-UR-08-05603

the Venti server responds with the SHA-1 [1] hash of the block contents called `score`. If the client needs to retrieve the data, it sends the score to the server and Venti sends back the block's content. The maximum size of a block Venti can store is 56 Kilobytes.

Using the SHA-1 hash of a block as an address allows the Venti server to detect blocks with the same content and ensure that they are stored only once on the disk. This property simplifies considerably the archival clients because they no longer need to figure out which files on the file system were changed. If the files are not modified, their subsequent archival is not going to use any more space in the archival system.

At a higher level, Venti supports storing and retrieving larger files by splitting them into blocks. The scores of the data blocks are combined into indirect blocks, their scores are combined further until a single score is produced that can later be used to retrieve the whole file. Venti files don't have names or any metadata information typically present for any modern operating system files. Venti also supports "directory" files that contain description (scores and some additional information) of Venti files. Each block in Venti has assigned a type value that indicates whether it is a data block, an indirect block (and the level of indirection), or a directory block.

Venti ships with utilities to store, copy or retrieve Venti files and directories.

2.2. Vac

Vac is a utility for storing files and directories in Venti. Venti converts the specified list of files into a list of Venti files and directories, saving the score of the top directory in a special *root* block. The score of the root block is returned to the user and can be used to retrieve the file hierarchy. Vac stores the regular files as a single Venti file. Because the Venti directories don't store files' metadata, each directory is represented with two Venti files – a Venti data file containing the metadata of the files from the directory, and a Venti directory.

Vacfs is a 9P [4] file server that given a score for a Vac root block can serve all the files stored with Vac. Vacfs can be used natively in Plan9, or using the v9fs [3] filesystem in Linux.

2.3. Using vac for archival

The Plan9 [5] dump filesystem provides a convenient view of its previous states. Each night a snapshot of the filesystem is taken and its content is available forever. The content of the file system on January 1st, 2001 can be reviewed by going to /n/dump/2001/0101 directory.

It is possible to achieve similar results on Linux by using Vac and Vacfs. Each night Vac is run to store the Linux filesystem in Venti, and the resulting score is saved in separate directory */YYYY/MMDD*. Then Vac is run again with `-m` option to expand and merge all vac scores in a single tree. The resulting score can be mounted using v9fs to provide the convenient Plan9 dump interface.

3. Vidi: archive when disconnected

Vidi is a server that speaks the Venti protocol. When the Venti server is available, Vidi acts as a proxy, redirecting client's requests to Venti, and Venti's responses back to the client. In addition to the redirection, Vidi builds a locally stored cache of scores for blocks that were sent to Venti. The cache is used in the disconnected state to detect blocks that Venti has and not store them for later transmission. Blocks whose scores are not present in the Vidi's score cache are saved in a block log. Both the score cache and the block log are stored on a local disk.

Figure 1 shows Vidi's operation when it is connected to Venti. Reading a block is always sent to the Venti server. The read operations don't affect Vidi's score cache. Writing a block first checks if its score is present in Vidi's score cache, and if it is present, a "success" response is sent back to the client without contacting the Venti server. Otherwise, the block is sent to the Venti server and on success, the score of the block is saved in Vidi's score cache.

When Vidi is not connected to Venti (Figure 2), read operations check if the score is present in the score cache, and if so whether the block is available from the local block log. In the unlikely case when the block is available locally, its content is sent back to the client, otherwise Vidi responds with an error. On write, if the score of the block is found in the score cache, a "success" response is sent to the client. Otherwise, Vidi appends the block content to its block log and adds the score to the score cache.

```
Get(score)
  if (data = venti.blockget(score))
    cache.putscore(score)
    respond(data)
  else
    responderror("not found")
```

```
Put(data)
  score = sha1(data)
  found = cache.putscore(score)
  if (found)
    respond(score)
    return

  if (venti.blockput(data))
    respond(score)
  else
    responderror(...)
```

Figure 1: Operation when Vidi is connected to the Venti server

```
Get(score)
  entry = cache.getscore(score)
  if (entry && blklog.valid(entry.address)
    data = blklog.read(entry.address)
    respond(data)
  else
    responderror("not found")
```

```
Put(data)
  score = sha1(data)
  found = cache.putscore(score)
  if (!found)
    found.address = blklog.append(data)

  respond(score)
```

Figure 2: Operation when Vidi is disconnected from the Venti server

Vidi keeps two pointers into the block log – of the first block that wasn't sent to Venti yet, and the position where the next block should be written to. When Vidi is reconnected to the Venti server, it starts sending the blocks from the block log to Venti. When all blocks are sent, i.e. the two pointers have the same value, the block log size is reset to the initial size. When a block is appended to the block log, its address doesn't directly reflect the offset where it is written in the log file. The block addresses always grow, even when the block log is shrunk after all blocks are submitted to Venti. This prevents updating the score cache addresses once the blocks are not in the block log anymore. When the block file is shrunk, Vidi updates a third pointer it keeps which keeps the logical address of the first block in the file. To check if a block for a score stored in the score cache is still available in the block log, Vidi checks if the "start" pointer is greater than the address of the block.

Unlike Venti's index, Vidi's score cache can drop scores of existing blocks. That can cause blocks that are already present in Venti (and even in the block log itself) to be added to Vidi's block log. The duplicates don't cause incorrect operation for Venti or Vidi. The only issue is the increased size of the local block log file. Our results show that with a reasonable size of the score cache, the number of duplicate blocks is not outrageous.

In addition to saving the score cache on a local disk, Vidi keeps some of the scores in RAM to improve the performance.

The prototype Vidi server is implemented for Unix operating system in 4000 lines of C code. It doesn't use the standard libventi libraries that are distributed with Plan9 from User Space.

3.1. Score cache disk layout

The disk layout (Figure 3) of Vidi's score cache is similar to Venti's index layout. The available disk space is divided into buckets (64K by default) and each bucket contains a map for a slice of the score space. The entries in the bucket are sorted by score. Unlike Venti, which depends on its index not overflowing, Vidi is designed to handle overflows and keep the most recently used scores in a bucket. Vidi doesn't keep a global LRU list. Instead it keeps per bucket LRU list. If a score needs to be added to a bucket, the least recently used entry in the bucket is

removed. In order to keep a LRU list, in addition to the block score, and its position in the local block log, the entry has pointers to the previous and next entry in the list.

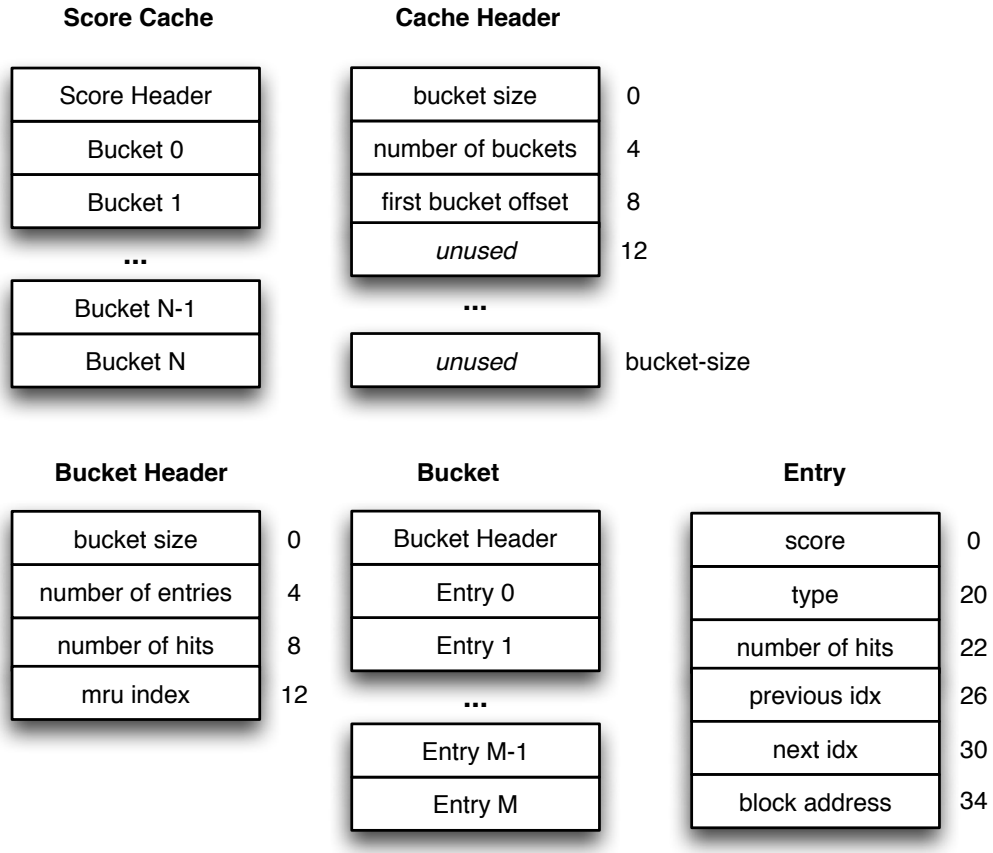


Figure 3: Score cache disk layout

3.2. Block log disk layout

Unlike Venti, Vidi's block log (Figure 4) is stored in a local file that is allowed to grow and shrink. The block log is not divided into arenas. The block log file consists of a header, list of data blocks and a trailer. The header contains a magic number and the "start" pointer. Each block contains a magic number, block's type, size and content. The log's trailer contains the "read" and "write" pointers. Vidi doesn't compress the block contents.

3.3. Using Vidi with Vac

When Vac is not used in an incremental mode, it converts the file system into a stream of "write" operations. Because Vac doesn't try to retrieve data from Venti, it would work well when connecting to Vidi even when disconnected. As Vidi doesn't always contact the Venti server even when connected, Vac's performance is improved even in non-incremental mode.

4. Performance results

The performance of the prototype is evaluated with different score cache and RAM cache sizes. The Venti server is running on a Linux server with 16 CPUs, 32GB RAM and 2.7TB arena space. The Vidi server is running on another Linux server with 2 CPUs and 2GB of RAM. Both servers are connected to the network with a Gigabit Ethernet card, but not to the same Ethernet switch and are 3 hops apart. The tests were performed using the vac program from Plan9 from User Space on a directory containing 117347 files with total size 11.38 GBytes. Before the tests were run, the directory was stored to the Venti server.

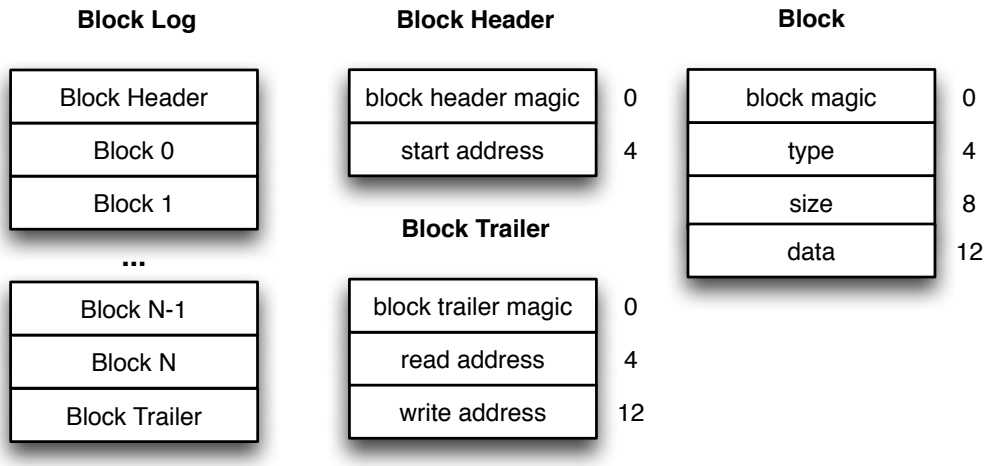
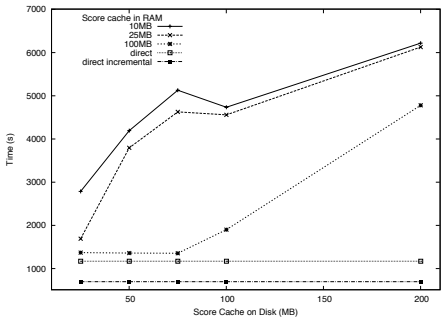
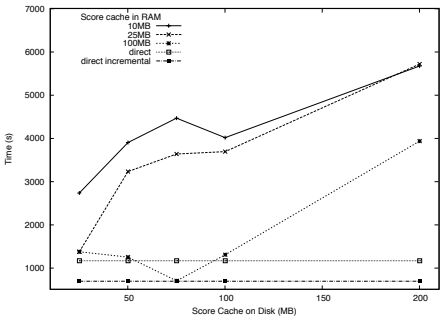


Figure 4: Block log disk layout

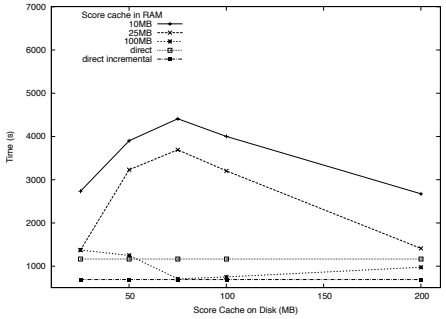
Figure 5 shows results running Vac with 64 Kilobyte buckets. Keeping information for the score recency leads to changes in the score cache even when the score is already present in the cache. This leads to higher number of operations to the local disk compared with the standard Venti which doesn't keep recency information per score. Having too small score cache leads to increased number of missed scores and even though the completion time is lower, Vidi stores an unacceptably high number of blocks (50 percent) even though they already are present in the central Venti server. Using too large score cache increases the I/O operations to the score cache too much decreasing the performance. The best results are achieved when the score



(a) Time to archive with empty score cache



(b) Time to archive with populated score cache



(c) Time to archive when disconnected

Score Cache Size (MB)	Cache Utilization (percent)	Block log size (MB)
25	100	6339
50	100	876
75	73	12.37
100	53	12.37
200	26	12.37

(d) Score cache utilization and block log size

Figure 5: Results using Vidi with 64 Kilobyte buckets.

cache is about 75 percent full. In that case, Vidi uses 0.74 percent of the storage a local Venti would use to archive the file system with performance comparable with the one achieved when using Vac in incremental archive mode.

Tests performed with smaller bucket size show improved performance at the expense of using more space used by the block log. Using smaller buckets reduces the I/O bandwidth, but the smaller number of scores in a LRU list increases the chance of score miss.

5. Conclusion and Future Work

Vidi allows standard Venti tools to be used for archiving when the central Venti server is not available. It caches locally the scores of the most recently written blocks. Vidi provides reasonable performance using a fraction of the disk space that other alternatives would use.

An interesting future work would be to extend Vidi to cache not only scores, but also the content of the blocks, allowing partial access to the archived file system. Experimentation with caching techniques other than LRU (ARC, MQ, etc.) could improve the hit ratio on both score and block cache further improving the performance and the user experience as a whole.

The implementation could be further improved by compressing the blocks in the local block log, and improving the I/O operations to the score cache.

References

[1] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[2] Russ Cox. Plan9 from user space. http://swtch.com/plan9port/.

[3] Eric Van Hensbergen and Latchesar Ionkov. The v9fs project. http://v9fs.sourceforge.net.

[4] AT&T Bell Laboratories. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[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. A cached WORM file system. *Software — Practice and Experience*, 21(12):1289–1299, 1991.

[7] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 7, Berkeley, CA, USA, 2002. USENIX Association.

Inferno DS: Inferno port to the Nintendo DS

Salva Peiró
Valencia, Spain
saoret.one@gmail.com

October 12, 2008

Abstract

The Inferno DS port began in 2007 as a one-man Google Summer of Code project, to make Inferno available on a standard, cheap, networked device with graphics and audio. The GSoC project attracted a small group of developers that is completing the port, to make the device fully usable for application development. This paper describes the current status of the port. It reviews the background and the motivation for the work, provides a DS hardware overview, and discusses the kernel development process, focusing on the setup and development of Dis applications running on the DS. There is plenty of scope for futher work. We hope to encourage others to contribute to the project.

1. Background

The DS [1] native port of Inferno [2] was started by Noah Evans for GSoC 2007 [3]. At the the end of GSoC the port was starting to be usable under the no$gba [4] emulator, enough that it was possible to interact with Inferno's window manager **wm(1)** [1] using the emulated touch screen. Inferno also booted and ran on a real DS, but the touch screen did not work. In spite of its limitations the port provided enough basic functionality to encourage further development. The GSoC project sparked the interest of a small group of enthusiasts to finish the port and begin work on new applications suitable for the platform. It is an Open Source project, hosted on Google Code, and supported by discussions in Google Groups and on IRC.

1.1. Motivation

The current project shares the motivation stated by Noah Evans on his GSoC 2007 application [3]: by using cheap and easily accessible hardware, native Inferno on the DS would show a wide range of users the power and possibilities of the Inferno and Plan 9 approach to building distributed systems. On other platforms, instead of a native port, we might consider hosting Inferno under an existing system, but we found that **emu(1)** on DSLinux [5] was not viable as when running with graphics the program crashed due to out of memory errors. There was thus increased curiosity about the advantages of a native port for DS software development. For instance, a proper operating system would overcome limitations of some homebrew programs for the DS, such as no multi-tasking, and it would give the benefits of having a coherent system with a standard set of tools. Furthermore, it would provide a "real" testbed for Limbo applications, including those developed in the inferno-lab [6]. The DS is particularly interesting as an Inferno target because it provides WiFi networking, allowing us to have fun with multi-user games and applications, including Voice-over-IP and jukebox programs using its audio input and output.

2. DS Overview

The native port had to address unusual aspects of the Nintendo DS hardware, so some knowledge of that is helpful. What follows is a small overview of the DS hardware organized in three subsections: the system processors, its inter-communication mechanisms, and last the built-in devices (and expansions).

[1] the notation **page(section)**, refers to Inferno manual pages [14]

2.1. Processors

The DS has two 32-bit ARM [7] processors: an ARM946E-S running at 66MHz that is in charge of the video and performs the main computations; and an ARM7TDMI at 33MHz that acts as a slave to deal with the remaining devices, including wireless, audio, touch screen, and power management.

The system is shipped with the following internal memory:

- 4096KB Main ARM9 RAM
- 96KB Main ARM7 WRAM (64Kb + 32K mappable to NDS7 or NDS9)
- 60KB TCM/Cache (TCM: 16K Data, 32K Code) (Cache: 4K Data, 8K Code)
- 656KB Video RAM (usable as BG/OBJ/2D/3D/Palette/Texture/WRAM memory)
- 256KB Firmware FLASH (512KB in iQue variant)
- 36KB BIOS ROM (4K NDS9, 16K NDS7, 16K GBA)

For more details see [8][GBATEK, NDS Overview].

2.2. Communication

The two processors in the DS can communicate using combinations of the following methods:

- Shared memory: The 4Mb of ARM9 RAM starting at 0x02000000 can be shared by both processors. It can be configured so that one cpu can be given priority over the other when they access the memory concurrently.
- Hardware FIFOs: The DS FIFO controller allows the processors to exchange 32 bit values. It allows full-duplex communication, where each cpu has a destination queue that stores the values sent by the other cpu, and interrupts notify the appropriate cpu about queue activity.

  This mechanism is crucial as it allows sending messages to request actions. This is used for example to read and write the real-time clock, obtain the touch coordinates, perform WiFi tasks, and request audio samples to be played or recorded to the ARM7 cpu.
- Sync interrupt: The Sync IRQ is a simple mechanism that allows one cpu ('local') to generate an IRQ to the other ('remote') cpu. We can use that to emulate WiFi receiver interrupts: when the ARM7 detects when a packet has been received it informs the ARM9 using Sync.

Given that accessing shared memory generates wait states to the cpu with less priority, it must be used with care. It works well in combination with FIFOs, by passing FIFO messages with pointers to shared memory. This is analogous to passing parameters by value or by reference.

See [8][GBATEK, DS Inter Process Communication (IPC)] for a more detailed description.

2.3. Devices

The Nintendo DS has the following built-in devices:

- Video: There are two 3-inch backlit LCD screens, each 256x192 pixels, with 18bit color depth. Each screen has a dedicated 2D video engine, and there is one 3D video engine that can be assigned to either screen.
- Sound: There are 16 sound channels (16x PCM8/PCM16/IMA-ADPCM, 6x PSG-Wave, 2x PSG-Noise). Output can be directed either to built-in stereo speakers, or to a headphone socket. Input can come either from a built-in microphone, or a microphone socket.

- Controls: A user interacts with the DS through a gamepad and a touch screen. The gamepad provides 4 direction keys plus 8 buttons, and the touch screen on the lower LCD screen can be used as a pointing device.
- Networking: WiFi IEEE802.11b wireless networking is provided by the RF2958 (aka RF9008) chip from RFMD. The main drawback is that there is no documentation from the manufacturer about its interfacing and programming. All that is known was reverse engineered by other projects. That information is gathered in [8][GBATEK, DS Wireless Communications] and also in the **dswifi** project and DSLinux [5]
- Specials: Additional devices include: a built-in real time clock, power managment device, hardware divide and square root functions and the ARM CP15 System Control Coprocessor (controlling cache, tcm, pu, bist, etc.)
- External Memory: There are two available slots: NDS slot (slot-1) and GBA slot (slot-2), which are the preferred way to plug in expansion cards and other devices. The slots are commonly used to provide storage on SD/TF cards. There are, however, other devices such as **Dserial**, **CPLDStarter** or **Xport** [9], which provide UART, MIDI, USB and standard digital I/O interfaces together with CPLDs or FPGAs.

see [8][GBATEK, NDS Hardware Programming].

3. DS Port

This section describes the idiosyncrasies of the DS port, in particular those related to the setup, kernel and application development.

3.1. Environment

The development environment is the default shipped with Inferno. The compiler used is `5{a,c,l}`, which forms part of the *Inferno and Plan 9 compiler suite* [11]. It is used to build the ARM [12] binaries for both the ARM7 and ARM9 cpus, together with the companion tools: `mk, acid, ar, nm, size, etc.` which are used for building, debugging and examining the resulting binaries.

The only special tool required is `ndstool` [10] which generates a bootable image to be launched by the NDS loader running on the DS. The image contains everything required to describe how to boot the code, which includes the ARM7 and ARM9 binaries and their corresponding load addresses and entrypoints.

3.2. DS kernels

The Inferno DS port follows the usual pattern for a port of native Inferno to a new platform for an already-supported processor. Much of the code of the Inferno native kernel is platform-independent, including the IP stack. The Dis interpreter and built-in Limbo modules are also platform-independent. That platform-independent code only needs to be compiled, which is done automatically by a `mkfile`. A relatively small amount of platform-specific code must be written. The DS port shares much of the ARM-specific code with the other ARM ports of Inferno, including the 'on the fly' compiler (JIT) for Dis for the ARM processor. There are existing ports of Inferno to the ARM, which have been used both as a source of ideas and code. Inferno's earlier port to the iPAQ is the closest existing platform to the DS: both have touch screens, storage, audio and wireless networking, The underlying hardware is completely different, however, and the DS often looks like a small brother of the iPAQ: a slower 66 Mhz CPU clock, only 4 Mb of available RAM, small LCD displays and reduced wireless capabilities.

One of the first things to address in the port was how to use the two processors. The ARM9 cpu has 4 Mb of RAM, which permits it to run an Inferno kernel, but the slower ARM7 has only access to 64 Kb or EWRAM (exclusive RAM). Given this memory limitation the ARM7 cannot sensibly run an Inferno kernel. Instead it runs specialised code that manages the hardware devices assigned to the ARM7. The ARM7 kernel is interrupt driven. During its initialisation phase, it sets device interrupts, and configures the buttons, touch screen, FIFOs, and the devices on the SPI. It then switches to a low-power mode, where it endlessly waits for interrupts to wake it. The kernel currently has 2,630 lines of C code, over half of that in its WiFi interface, and 70 lines of assembly code.

The ARM9 runs the full Inferno kernel, and provides devices like **pointer(3)**, **ether(3)**, **rtc(3)**, **audio(3)**, etc. About 6,500 lines of C code and 310 lines of assembly code is specific to either the ARM processor or the DS platform. Most of that code is in device drivers. The implementation of the device drivers is unusual: because of the division of work between the processors, the drivers must access and control many of the physical devices via the ARM7, and we discuss that next.

3.3. Communication: FIFOs IPC

To avoid conflicts that would arise if sharing the hardware devices between cpus, each device is assigned exclusively to one cpu or the other. For example, the Serial Peripheral Interface (SPI) is owned by the ARM7. Many of the peripherals are accessed through SPI, including touch screen, WiFi, rtc, firmware, power management and audio. The LCD hardware by contrast is owned by the ARM9. Consequently, the ARM9 cannot directly drive the audio device, nor can the ARM7 directly display on the screen for debugging.

To overcome this, we use the interprocessor communication mechanisms listed above – FIFOs and shared memory – to implement a simple messaging protocol that allows one cpu to access devices owned by the other. It is a Remote Procedure Call protocol: each message is associated at the receiving cpu with a function that performs the work requested by the message. For simplicity the function and its arguments are encoded into a 32 bit message as follows:

```
msg[32] := type[2] | subtype[4] | data[26], where
field[n] refers to a field of n bits of length

type[2] := 00: System, 01: Wifi, 10: Audio, 11: reserved.
subtype[4] := 2^4 = 16 type specific sub-messages.
data[26] := data/parameters field of the message.
```

The encoding was chosen to have a notation that was easy to read in the calling code, yet accommodate all the data to be exchanged between the cpus:

`type[2]` is used to have messages organised in 4 bit types: System, Wifi, Audio and a Reserved type.

`subtype[4]` is used to further qualify the message type.

For example, given message `type[2] = Wifi` actions to be performed include initialising the WiFi controller, setting the WiFi authentication parameters, and preparing to send or receive a packet. Those and the other operations required can all be encoded using the 16 available message `subtypes`.

`data[26]` the data field is just big enough to allow passing of pointers into the 4Mbyte shared memory. (This will have to be revised when using memory expansions @ 0x08000000, 16 Mb)

The protocol has a simple implementation. For instance, here is the low-level non-blocking FIFO put function:

```
int
nbfifoput(ulong cmd, ulong data)
{
    if(FIFOREG->ctl & FifoTfull)
        return 0;
    FIFOREG->send = (data<<Fcmdlen|cmd);
    return 1;
}
```


Here is an example of its use, extracted from devrtc.c, executed by the ARM9 side to read the ARM7 RTC:

```
ulong secs;
...
nbfifoput(F9TSystem|F9Sysrrtc, (ulong)&secs);
```

Because the hardware interface to the FIFO is the same for each processor, similar code can be used by the the ARM7 in the other direction, for instance to send a string to the ARM9 to *print* on the LCD. (The code is not identical because the ARM9 kernel environment includes scheduling.)

The interrupt-driven part of the FIFO driver is also straightforward. An extract is shown below to give the flavour:

```
static void
fifotxintr(Ureg*, void*)
{
    if(FIFOREG->ctl & FifoTfull)
        return;
    wakeup(&putr);
    intrclear(FSENDbit, 0);
}

static void
fiforxintr(Ureg*, void*)
{
    ulong v;
    while(!(FIFOREG->ctl & FifoRempty)) {
        v = FIFOREG->recv;
        fiforecv(v);
    }
    intrclear(FRECVbit, 0);
}

static void
fifoinit(void)
{
    FIFOREG->ctl = (FifoTirq|FifoRirq|Fifoenable|FifoTflush);
    intrenable(0, FSENDbit, fifotxintr, nil, "txintr");
    intrenable(0, FRECVbit, fiforxintr, nil, "rxintr");
}
```

Here `fiforxintr` is executed when an message receive IRQ is triggered, then the FIFO is examined to read the message, which is passed to `fiforecv` which knows the encoding of the messages, and invokes the corresponding function associated with each message.

3.4. Graphics

The DS has two LCD screens, but the **draw(3)** device currently provides access only to the lower screen, because it is the only touch screen in the DS, and mapping touch screen coordinates to screen coordinates (pixels) makes obvious sense to a user: touching the screen refers to that point on the screen.

The DS port could also draw on the upper screen, but it will take some experimentation to determine how to best use both screens so the result still makes sense to both user and programmer. For example, although Limbo's **draw(2)** does not require that everything drawn be accessible through **/dev/pointer**, existing interactive applications effectively assume that.

One interesting alternative is to use the touch screen coordinates as relative instead of absolute: this would provide access to both screens, and visual feedback can be provided by a software cursor.

3.5. Memory

Having 4 Mb of RAM limits the programs that can be run. To overcome this memory limitation, it is possible to use slot-2 memory expansions; the expansions can add between 8 Mb and 32

Mb of RAM, Unfortunately, owing to the slot-2 bus width it can only perform 32-bit and 16-bit writes; when an 8-bit write is performed it results in garbage written to memory.

This problem is circumvented in DSLinux [5] by modifying the compiler to replace `strb` instructions with `swpb`, with appropriate changes to surrounding code. We might be able to do the same in the Inferno loader `5l` (since that generates the final ARM code), but failing that, make a similar change to the compiler `5c`.

3.6. DLDI

The Dynamically Linked Disc Interface (DLDI)[16], is a widespread way of accessing storage SD/TF cards. It provides the IO functions required to access storage independently of which boot card is being used. When a `file.nds` file is booted, the boot loader auto-patches the DLDI header contained inside the `file.file` with the specific IO functions for this card.

This has been partially implemented in the DS port, where the `devdldi.c` file provides a suitable DLDIhdr, which is properly recognised and patched by the boot loader.

The problem with this approach is that the DLDI patched code (*arm-elf*) contains instructions which modify a critical register without restoring it afterwards, which would panic the kernel.

For that reason, at this moment the DLDIhdr is only used to detect the card type and then select one of a set of compiled-in drivers, one for each type of card.

3.7. Application

As usual for Inferno ports, the existing Dis files for applications run unchanged (subject to available resources). At the application level the DS has some features that make it interesting.

User input comes from various buttons, and the touch screen. Graphical output is on two small LCD displays. As noted above, having two displays but only one with a touch screen presents a different graphical interface from the one that applications (and users) expect. This is currently the object of experimentation in the inferno-lab [6].

Whichever approach is chosen, being able to run Limbo applications in the full Inferno environment on the DS already opens the field for interesting applications, which combine graphics, touch, networking and audio. This can include games, VoIP, music, MIDI synths, and other more common uses, such as connecting to remote systems with **cpu(1)**, and managing them from the DS, or accessing remote resources using the **styx(5)** protocol.

3.8. Setting up the development environment

It is easy to set up Inferno to run on the Nintendo DS. An Inferno kernel that can be distributed as an `.nds` image is available for download from the Inferno DS project site [1]. A standard Inferno distribution is placed on an SD/TF card, and the .nds kernel image can be copied to an SD/TF card, to be booted by the NDS loader.

This kernel provides access to the underlying hardware through Inferno's normal device interface, namely through a file system interface that is used by applications to access most kernel services. The kernel includes the normal Inferno interfaces for **draw(3)**, **pointer(3)**, **ether(3)** and **audio(3)**, and a DS-specific **devdldi** that provides storage access to SD/TF cards.

With all this, the development of applications consists of the following steps:

1. setup Inferno emu on a development host: where the applications can be coded, compiled and tested, see [13] for more details.

2. test applications on a DS emulator *(optional)*: like no$gba [4] or *desmume*.

3. transfer applications (.dis files) to SD/TF card: to be launched after booting the Inferno DS kernel.

4. Conclusions

The main conclusion extracted during the development of the port has been how the careful design and implementation of the whole Inferno system have made the task of developing this port easier. Most of the kernel code is portable, including the whole of the Dis virtual machine, and just needs to be compiled. The platform-specific kernel code for any native port is fairly

small (on the order of a few thousand lines). There was already existing support for the ARM processor, and a few sample ports to ARM platforms to act as models. The device driver interface is simple and modular.

This has had also an effect on the tasks of locating and fixing errors, and introducing new functionality like input, storage, networking and audio which have become easier. Emulators are still of great help to save test time.

The benefits of the Inferno design [2] will be also noticed when developing Limbo applications for the DS, as this area has been less used and tested during the development of the port.

5. Future work

This project is *work in progress*, and significant things remain to do. There are undoubtedly places where a simple-minded implementation just to get things going needs to be redone. For example, the graphics implementation is being extended to allow Inferno to take advantage of both LCD screens, and the audio driver is being reworked to improve playing and recording quality.

One big task is to finish and test the wireless networking code. The DS will be much more interesting once it can communicate with other devices, because Inferno comes into its own in a networked environment. That will allow it to access file systems and devices provided by an **emu(1)** instance running hosted elsewhere. We can also speed development by booting remote kernels. The wireless provides only WEP and open modes at 2.0 Mbps. Once the WiFi code is fully working, it will be interesting to see how the relatively low data rate (in current terms) affects the use of the **styx(5)** protocol to access remote filesystems.

As low-level device support is completed, effort will shift from the kernel side to the applications side. Indeed, that is already happening with the inferno-lab [6] experiments with the Mux interface and with the QUONG/HexInput [15] keyboard to ease interaction with the system through the touch screen.

Please join in! [1]

References

[1] Noah Evans, Salva Peiró, Mechiel Lukkien "Inferno DS: Native Inferno Kernel for the Nintendo DS". http://code.google.com/p/inferno-ds/.

[2] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, Phil Winterbottom "The Inferno Operating System". Computing Science Research Center, Lucent Technologies, Bell Labs, Murray Hill, New Jersey USA http://www.vitanuova.com/inferno. http://code.google.com/p/inferno-os/.

[3] Noah Evans, mentored by Charles Forsyth, "Inferno Port to the Nintendo DS". Google Summer of Code 2007, http://code.google.com/soc/2007/p9/about.html.

[4] Martin Korth, "no$gba emulator debugger version". http://nocash.emubase.de/gba-dev.htm.

[5] Pepsiman, Amadeus and others, "DSLinux: port of uCLinux to the Nintendo DS". http://www.dslinux.org.

[6] Caerwyn Jones & co, "Inferno Programmers Notebook". http://caerwyn.com/ipn, http://code.google.com/p/inferno-lab

[7] ARM (Advanced Risc Machines), "ARM7TDMI (rev r4p3) Technical Reference Manual". ARM Limited, http://www.arm.com/documentation/ARMProcessorCores.

[8] Martin Korth, "GBATEK: Gameboy Advance / Nintendo DS Technical Info". http://nocash.emubase.de/gbatek.txt. http://nocash.emubase.de/gbatek.htm.

[9] Charmed Labs, "Xport". http://www.drunkencoders.org/reviews.php.

[10] DarkFader, natrium42, WinterMute, "ndstool Devkitpro: toolchains for homebrew game development". http://www.devkitpro.org/

[11] Ken Thompson, "Plan 9 C Compilers". Bell Laboratories, Murray Hill, New Jersey 07974, USA. http://plan9.bell-labs.com/sys/doc/compiler.html.

[12] David Seal, "The ARM Architecture Reference Manual", 2nd edition. Addison-Wesley Longman Publishing Co. http://www.arm.com/documentation/books.html.

[13] Phillip Stanley-Marbell, "Inferno Programming with Limbo". John Wiley & Sons 2003, http://www.gemusehaken.org/ipwl/.

[14] "The Inferno Manual". http://www.vitanuova.com/inferno/man/.

[15] http://www.strout.net/info/ideas/hexinput.html.

[16] Michael "Chishm" Chisholm, Dynamically Linked Disc Interface. http://dldi.drunkencoders.com/index.php.

9P For Embedded Devices

Bruce Ellis
Tiger Ellis

Club Birriga
Bellevue Hill, NSW, Australia
brucee@chunder.com

ABSTRACT

9P has proved over the years to be a valuable and malleable file system protocol. Furthermore, as is it embraced by Plan9, it is more than a convenient protocol for interaction between disparate devices. Indeed Plan9 relies on it.

The protocol can be used to encapsulate control of an embedded device, which simply serves a 9P file system. However, even though 9P is very lightweight, it can be adapted to be more frugal on device resources. This is important on very small devices (FPGAs) where a full 9P implementation can consume most of the available gates.

We address this issue as a filesystem (embedfs) on the embedded machine's gateway Plan9 machine. We provide implementation and configuration details targeted at the *Casella* Digital Audio device.

1. Introduction

9P filesystems are used for diverse and often unexpected purposes. You need only look at upas [ref], fossil [ref], and ftpfs(1). Most are served by user-level processes, the kernel providing the necessary multiplexing and presenting physical devices as 9p servers. Remote devices are accessed seamlessly via whatever connection protocol is appropriate to the target. Typically this a common service, like 9fs, using a TCP connection. It can easily be a specialized server on an embedded device connecting via USB, serial, raw ether, etc.

A small embedded device may not have enough resources to provide a full 9P service. The resources that may be lacking include buffer space, outstanding request queue space; and of major concern sufficient silicon for handling the full protocol. Our intention is to provide a a file system which acts as an interface to a device implementing a (configurable) subset of 9P, seamlessly – respecting the integrity of the model.

Arguably a filesystem tailored to a specific device with a custom protocol is a more efficient use of cycles. We instead embrace a reuseable, respectable, configurable model and existing code – a more efficient use of brain cycles.

2. An Embedded File System Interface

The interface is implemented using lib9p [ref], which provides some clear optimizations. (Familiarity with the 9P protocol is assumed in this paper for brevity.) It is well structured and malleable.

Given the disclaimer we will state a result for a small embedded device, which has a very fixed structure and limited resources. This could easily be the conclusion – except there

is more to tell.

This is what Casella looks like:

```
% cd /n/casella; ls -l
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 audioctl
---w--w--w- M 324 casella casella 0 Aug 26 22:02 audioin
--r--r--r-- M 324 casella casella 0 Aug 26 22:02 audioout
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 ctl
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 irom
--rw-rw-rw- M 324 casella casella 0 Aug 26 22:02 midictl
---w--w--w- M 324 casella casella 0 Aug 26 22:02 midiin
--r--r--r-- M 324 casella casella 0 Aug 26 22:02 midiout
```

The directory served is flat with a constant map between name and stat info (including Qids). This information is loaded by embedfs from a configuration file.

Enumerating the 9P Tmesgs served by embedfs:

Tversion
: lib9p handles this message.

Tauth
: lib9p user auth() function handles this. Usually no authentication is required, access is managed by permissions on the srv file. It seems unnecessary to replicate the natural plan9 access mechanism.

Tflush
: Passed onto the device, held by the server, or even discarded.

Tattach
: Returns the root Qid.

Twalk
: Returns the appropriate Qid.

Topen
: Returns the appropriate Qid, and a suitable iounit. Informs the device if appropriate.

Tcreate
: Eperm.

Tread, **Twrite**
: Passed onto the device.

Tclunk
: lib9p handles this message. User function destroyfid() informs the device if appropriate.

Tremove
: Eperm.

Tstat
: lib9p user stat() function handles this (based on configuration data).

Twstat
: Eperm.

Note that the communication with the device can (and does) use a subset of 9p (specifically: open, clunk, read, and write). In fact the device need only support read and write.

3. A Closer Look

The result presented above is readily implemented using `9pfile(2)` – the `Tree` and the collection of `Files` are fixed once the configuration is loaded, the communication with the device uses `fcall(2)`. The device requirements are small – storage and logic fall into "a small chunk of the device" category. So what's up? First we'll look at improvements to this implementation for a small, simple, device (`casella`) and then examine enhancements for more capable devices.

3.1. `iounit` Bottleneck

The high bandwidth files, `audioin`, `audioout`, and `irom`, have small on-chip buffers, so the obvious thing is to reflect this in the returned `iounit`. This has a very adverse effect upstream as a read of 8K will generate an enormous amount of host to host traffic. If these files are configured as "buffered" we can advertise a large iounit and handle the large transaction in the server with multiple (local speed) transactions with the device.

Example: The server receives a `Tread` request with size of 4K. The device has a 32 byte buffer. The server sends multiple 32 byte `Tread` requests to the device until one of a) the 4K buffer is full, b) a short read, or c) an `Rerror`. Similarly for `Twrite`.

3.2. Outstanding Requests

The chip has limited resources for storing outstanding requests. The device architecture is such that a restriction of a single request per file is natural and adequate. The server could simply queue requests per file. It may also wish to gate file opens to effectively make each file "exclusive-open with wait rather than error", allowing reads/writes of an open file to overtake waiting opens. This is particularly handy for control files. `Fids` and `Tags` are handled in the server, translated to device file number for communication with the device.

3.3. The Result

With these modifications the silicon footprint on the device is bounded (always good) and small in both storage and logic.

4. Enhancements

`Casella` has strict real-time constraints. Audio input and output are both 176KB/sec. Midi is much slower but still must not overflow/underflow. A program using `embedfs` to control a `casella` must use multiple outstanding reads and writes to meet these constraints. A library is provided to encapsulate this. The server uses `edf` [ref] to guarantee the device data rates specified in the configuration file.

5. Example Configuration

The configuration file for `casella` is listed below.

```
#
# casella.conf
#
downlink   2M
uplink     2M
iounit  32
buffer  8K
file   audioctl   666
file   audioin    222    buffered 176K
file   audioout   444    buffered 176K
file   ctl        666
file   irom       666    buffered
file   midictl    666
file   midiin     222    buffered 3125
file   midiout    444    buffered 3125
```

Mrph: a Morphological Analyzer

Noah Evans
noah-e@is.naist.jp

ABSTRACT

Developing tools for Natural Language Processing is hard, requiring careful tuning of statistical models and data processing optimization. It's even harder given the many competing and incompatible tools, encodings and data sets in use in modern NLP research.

To implement new tools researchers have to reimplement or port the previous tools, using time for development that could be better spent doing productive research. There have been attempts to make portable, flexible low level analysis systems, notably Freeling, that incorporate a flexible NLP tool chain in a language independent way that can be easily incorporated into other tools and workflows.

We present a new morphological analyzer, mrph, which attempts to implement a language independent morphological analyzer both a viable vehicle for research and the day to day use. The system is divided into modules, written with native support for utf8 , and uses a shell and pipeline syntax that is semantically identical across systems. It is also written in a statically typed language with module support, allowing it to dynamically load and discard language resources at will. This allows mrph to change the processed language on dynamically, giving it potential for irregular data sets like the web.

Introduction

Mrph is a morphological analyzer written in Limbo for inferno.

A morphological analyzer is a tools for taking a sentence and breaking it up into its component morphology, a set of terms describing the implicit structure of the sentence.

For instance the sentence:

日本語を教える人と日本語を学習する人がともに楽しめるポータルサイトです。

when tokenized and classified by a morphological analyzer becomes:

日本語	ニホンゴ	日本語	名詞-一般
を	ヲ	を	助詞-格助詞-一般
教える	オシエル	教える	動詞-自立　一段　基本形
人	ヒト	人	名詞-一般
と	ト	と	助詞-並立助詞
日本語	ニホンゴ	日本語	名詞-一般
を	ヲ	を	助詞-格助詞-一般
学習	ガクシュウ	学習	名詞-サ変接続
する	スル	する	動詞-自立　サ変・スル　基本形
人	ヒト	人	名詞-一般
が	ガ	が	助詞-格助詞-一般
ともに	トモニ	ともに	副詞-一般
楽しめる	タノシメル	楽しめる	動詞-自立　一段　基本形
ポー	ポー	ポー	名詞-固有名詞-地域-一般
タル	タル	タル	名詞-接尾-助数詞
サイト	サイト	サイト	名詞-一般
です	デス	です	助動詞　特殊・デス　基本形
。	。	。	記号-句点

This provides an annotation that allows the sentence to be dealt with by the user and do what the user wants when they want to do greater amounts of research. In the case of this Japanese sentence is provides the pronounciation, the uninflected form of the verb, and its inflection type in each column respectively.

This information and annotation provided by the analysis forms the basis for creating solutions to larger problems in natural language problems, including syntax tree parsing and anaphora resolution.

Given their importance to other tasks, good morphological analyzers are a foundational tool for NLP researcher, much other research depends on it. This means that a lot of effort goes into optimizing the performance and accuracy of different analyzers.

The streaming nature of the morphological analyzer's task(i.e. a stream of input sentences each transformed into a set of moprhological tokens annotated with linguistic information) coincides nicely with the unix piped workflow, which connects small tools using pipelines provided by the operating system.

However, despite this natural affinity morphological analyzers are rarely inplemented as software tools. This happens for a few reasons, primarily one of portability. Given the importance of morphological analyzers to linguistic analysis and the popularity developers and researchers make special effort to implement the system as a library that can be used by a larger application or by providing bindings to other scripting languages like perl and python. This also encourages a style of programming where many types of functionality independent of the analyzer, like formatting systems and character set handling functions are implemented into the analyzer directly.

By trying to shoehorn analyzers into a variety of different operating systems with functionality trying to be all things to all users, morphological analyzers typically become arcane and verbose, making it difficult to add new functionality and change the system without major changes to the underlying analyzer itself. This makes it very difficult to support new languages or implement improved analysis systems in preexisting systems, typically they are reimplemented from scratch.

There have been attempts to deal with this problem, Freeling[cite] but they fall back on the method of using libraries and overly complicated interprocess communication protocols like corba to implement.

Goal: A software tool that can be used for research

With these problems we designed mrph with the goal of providing a tool that can reliably used for both day to day use as a morphological analyzer for higher level tasks and be easy to use to advance the state of the art in morphological analyzer research. With these motivations in mind we set the following goals:

1. develop a well engineered modular analyzer suitable to generalizing its methods. especially one with a set interface. Morphological analyzers typically use "one-shot" methods[cite], so the ideal way to deal with the system is to generalize one shots and allow *any* method to deal with it.

   Make it possible so that any developer as well as user can add the various parts of it. A tool for research.

2. engineer a system that would work as part of the inferno/plan 9 "software tools" ecosystem. giving data in a form that could easily be reparsed using stream transform tools similar to awk or sed.

3. choose an interface that allows the user to use unix style goodies, but, at the same provides sane defaults without configuarability. keep the interface the same across systems.

Mrph: a software tool for morphological analysis

Mrph was implemented with these goals in mind.

It is structured as a set of modules that works to compose. Mrph takes a different approach. it is a set of modules the goal is to be able to swap languages and data on the fly. it uses a system based on tokenization of asian languages. sacrifices efficincy for that ability to handle words as prefixes. it implements caching manually to allow itself to handle ranges that are much larger.

unlike many morphological analyzers only analyzes unicode.

it also attempts to be multilingual by ignoring traditional language tokenization, using the approach of asian language analyzers of deciding on possible morphs by doing prefix searches. this does stemming and lemmitization and multi word expression validation essentially for free. by ignoring

Interface

Unlike many morphological analyzers mrph is implemented as a "software tool" in the unix tradition.

linguistic researchers can be traditional researchers, but they typically have a variety of systems to work on. Given the idiosyncrises of these systems, it is impossible to assume support for things. to support all possible users people use approaches like Chasen[cite] or Freeling[cite] developing systems as tools and libraries, allowing the system to be used as part of a greater monolithic system.

Implementing the analyzer in limbo obviates many of the problems. both in terms of interface and implementation,

Input

Mrph expects plain utf8 text data as input, now, currently limited to the format of one sentence per line. It takes Japanese text input and gives you the value of their analysis.

Given that its expected language is utf8 by having native support for the system. Since inferno supports utf natively both in the programming language and the system interface level it makes it possible for mrph to handle any language naturally(except right to left languages like Arabic and Hebrew which still confound the construction of a simple

interface).

This allows mrph to handle any language automatically(potentially, right now it only supports Japanese). interspersed english and Japanese are handled in the same way provided that the input is utf8.

Output

The system outputs data in tree paths[cite]. This may seem unnecessary but in the future the system will support the input of values already in treepath format, allowing the system to potentially take advantage of higher levels of morphological data when doing analysis, allowing potential positive feedback loops where annotation is fed back to the analyzer allowing each level of the linguistic analysis process to have positive feedback with each other.

```
/N/日本語  ニホンゴ    日本語
/PG/を      ヲ          を
/VBT/教える             オシエル    教える                  一段        基本形
/NG/人      ヒト        人
/APC/と     ト          と
/N/日本語  ニホンゴ    日本語
を          ヲ          を          助詞-格助詞-一般
学習        ガクシュウ  学習        名詞-サ変接続
する        スル        する        動詞-自立   サ変・スル  基本形
人          ヒト        人          名詞-一般
が          ガ          が          助詞-格助詞-一般
ともに      トモニ      ともに      副詞-一般
楽しめる    タノシメル  楽しめる    動詞-自立   一段        基本形
ポー        ポー        ポー        名詞-固有名詞-地域-一般
タル        タル        タル        名詞-接尾-助数詞
サイト      サイト      サイト      名詞-一般
です        デス        です        助動詞      特殊・デス  基本形
。          。          。          記号-句点
```

This sacrifices. some of the readability of the original format, the abbrevation of the more descriptive version. Maintaining the original format added to much visual clutter with unicode fonts,

This is not especially pleasing with other fonts.

Pipelines

utf8 also allows for pipeline streaming covered in Thompos et al[cite]. which makes it possible to use the system with data that is potentially cut up, making it a better software tool candidate.

because the text that mrph supports can be broken up easily it can support pipelining in a very natural way. This allows it to be used as the part of a toolchain which builds up to the solution of a more general problem. In fact the system is meant to be used in Cocytus[cite] which discuss pipelines and the viability of Inferno as a NLP environment in greater detail.

However, as mentioned early pipelines are impossible to implement using an operating systems inherent primitives, which makes it impossible to move analyzers like mrph between systems because the operating system cannot reasonably be expected to handle everything natively.

A great advantage of programming a tool in limbo is that it comes with emu, which allows the system to be semantically identical across platforms. While other bytecompiled languages like Java allow programs to behave similarly across platforms they don't ensure the same *platform* between different systems, breaking one of the primary advantages of a portable language. The quirks of each system forces the user to

abandon.

Because mrph is a simple software tool running in inferno, it interacts with other tools(in inferno) using pipelines, allowing it to be without extra features or extraneous interface. Any text after processing, any character set conversions can be do as pre and post processing over pipelines, allowing mrph to concentrate on its purpose Morphological analysis.

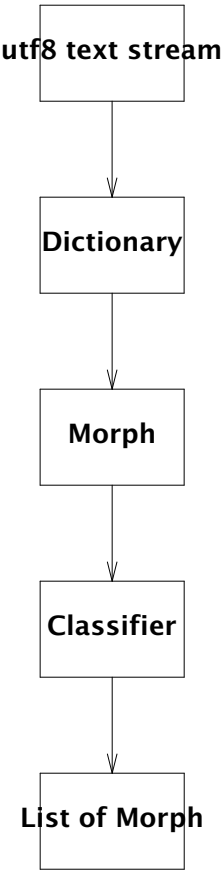
For example if you wanted to analyze a webpage and wanted all examples of consecutive noun phrases.

```
hget http://www.asahi.com | htmfmt | mrph |  readline
```

Implementation

This section describes the implementation of mrph. It begins by describing the modular structure of the implementation and the behavior of the main analyzer. It then goes on to describe the implementation of the original modules for the system and the practical considerations that went into their construction.

Modular construction



The system is implemented as a set of modules which implement the various structures of mrph:

takes a text stream and views the input as a set of prefixes. These prefixes are then fed to a dictionary which returns the valid morphs. which is then done with that.

Fundamentally the system is a mapping from text stream -> a set of morphs.

the system does so by taking the prefixes and giving them to a dictionary. it then gives a constant weight to the undefined and uses that to establish the valid path.

Once the valid path is established then the system is given the proper value. and the sequence of morphs is printed to standard output.

Separate system into modules

Since the system is taking streams of utf8 text and converting them into morphs.

that is why different morph modules need to have a way of dealing with this.

This allows the system to be easily modified. Your dictionary data structure as long as it supports prefix searches

the path module just takes a list of possible morphs and works that out itself.

the goal is the keep the modules as separate as possible, so that when any one of the system is altered it doesn't change the system itself.

Role of Modules

This separation of the system into modules is not novel, information hiding is part of any modular system, but this is especially important regarding parts of language.

programs like freeling use a set of files to determine the structure of the system. This works to a point, but at the same time it is limited to the scope of initial programmer and adds another level of indirection and complexity which precludes the understanding of the system itself.

when the user want to change language it's as simple as compiling a dictionary, set of paths and morphs for the system.

You want the user to be able to take advantage of the system and put everything together.

Tokenization

In order to deal with the largest amount of languages with no change in the processing of the underlying system the system is dealt with in a fashion that maximizes the way that language is dealt with. Language is viewed as a series o characters rather than words. word boundaries are determined entirely by the dictionary which determines all the possible prefixes of the sentence stream that can provide valid words.

This may seem like a waste in whitespace separated languages, where hash based methods on individual words may be more efficient like english but has the advantage of catching simple multiword expressions like "hard drive" automatically. This does not catch all the possible multiword expressions possible see Bond et al. [cite] for a list of the problems of multiword expressions and their detrimental effect for NLP.

This has the effect of punting the tokenization from the hardcoded analyzer itself to the dictionary and its contents, which are replacable.

Using the modules for research

By separating out the modules the system can be optimized in various ways.

The linguistic structure of the values being read is entirely up to the Morph module. The dictionary and the analyzer itself have no idea of the internal structure of a mrph. The morph module also includes a string function which allows it to print itself as well. obviating any need for the morph to know the state.

The dictionary

Actual implementation of the modules

All of this work to modularize the system is means nothing if the system is not efficient. However practical concerns are important as well. The data dealt with the system is large enough that the system is kept in a large enough size by putting the stuff together.

the goal should be to solve a large enough subsection of the modules and so on.

The dictionary module

The central problem of implementing a dictionary module is that there can be a huge amount of words that a system needs to understand. Unkown words are disasterous for the accuracy of morphological analysis, so any effective morphological analyzer will require a large dictionary(ours, IPAdic[cite] is 3910000 words) to effectively deal with language parsed and implemented.

So to do this we need a dictionary structure that is large enough to work with but at the same time.

There is a huge number of words that work well together.

these are well understood problems of morphological analysis, the traditional method of solving this is for the system is to simply write the dictionary structure to an mmap'ed file and let the operating system page in.

however this is inefficient[cite] and non portable, saving processor and memory specific information to the data file. current morphological analyzers get around this problem by compiling the structure for an original dictionary file when they are first set up on a system, but this a suboptimal solution and one that is traditional solved in other ways in the bell labs style.

We solved the problem by using a layered dictionary.

the main problem is that by not using a hash based data structure we are stuck with a trie based structure which is not space efficient to begin with. A hash based structure such as dbm(1) would be better but that would force the system to infer the tokens. by adopting a prefix based approach the dictionary can assign possible tokens as it goes, allowing the natural processing of asian languages as well as English.

Does this by implementing a patricia trie with nodes that are kept in various levels.

the central problem is that since everything is prefix based you eventually end up have to search the entire data structure. if you decompose the dictionary. you can potentially miss morphs. so the system just uses a hash table where the function is computed for each word.

this means that the system potentially does lookups in much slower time than a trie. however because it does this by going through the system. this has potentially poor behavior and may cause many seeks.

you still fail because if you get to the end you never know where the real end is.

So we go about this by going through the system by

Morphs

Likewise by having to store 390,000 keys in a database the system needs to hold the values stored in those keys. Each morph consists of a part of speech id and a various values. this means that a morph includes on average about 32 bytes of data(x words + x values + y somethings).

Also various morphs are much more popular than others. This means that the system needs to cache them in order to get a very good benefit. The caching for the morph module is modeled on the subfonts in Plan 9.

The problem is that the dictionary doesn't know the location of either. The system finds the Morphs by checking the cache value first. The cache is implemented as a hash table. anything that exists in the hash are then available. the system keeps an age for each of the morphs and little used morphs are purged from the system during periodic garbage collections.

need to keep a list of morphs. that is the problem.

The Paths module

The paths module is relatively simple. it uses a hardcoded limbo array, compiled from an external matrix file. this establishes the possible transitions from one combination to another.

The path module is similar to regular morphological analyzers, it contains a conjuction table which figures out how the paths are connected, it also contains a connection matrix which keeps track of how the values are connected in the system.

Finally it contains a Lattice which preserves the state of the analyzer, where the value is in the string, which paths exist and how the paths should be classified.

An easily modifiable tool for research

Data formats

Traditionally morphological analyzers, like Chasen[cite] and Mecab[cite], take the modern unix approach, mmap'ing its data structures to external files and then treating those files as part of the executable itself.

Data Organization

Have three different types.

```
Morphs, the data.
Dictionary
the matrix.
```

Morphs

Have to manually figure out the cache types. Morphs typically have a great deal of locality. Sam, acme

Morph handling

Plan 9 offers a rich model for caching.

```
mrphs sets of files
        sucks in files based on their values.
        files ordered according to their commonality.
        works in a way similar to
```

the goal is to avoid mmap

a caching system similar to fonts

Disadvantages

Inferno doesn't really have the set of tools that it needs to be productive for more tools. Tools like awk are very useful for language processing because they take textual input divided into fields by white space and allow their easily accumulation and editing. Inferno's shell while power is still too verbose for quick and dirty shell pipeline construction.

It can be pretty ugly. Many of the unix conventions came from strictly ascii text which makes it hard for typographical conventions which are human readable like

```
/1.NP/2.Word
```

Which become much harder when they are put into practice using other languages.

Traditional path formats don't look that bad in greek.

```
/Mή/
/μόλις εβδομή δα
```

but characters with large widths and values are much more difficult to visualize easily using an editor like acme. which prompted moving the data formats to ascii.

Conclusion

Eliminates many of the problems and pitfalls that come from trying to implement a tool for a software tools system.

Limbo is a great language for doing multilingual programming. By allow the language itself to use utf8 and integrating it with the system.

Also by working the same way across architectures you don't need to go through the same issues that people normally go through to integrate with other tools, especially languages like Java.

It avoids the problems with C(i.e. people being able to randomly type things) but it lets the user get the way of doing things right.

No mmaping.

Forces many of the system's dark corners into the light.

The act of doing this, and making the various formerly implicit or recondite aspects of the system more accessible to programmers makes the system much more amenable to experimentation.

The module boundaries are clear.

None of this is specific to limbo per se.

but limbo does provide a way of doing things that encourages well engineered programs with less complexity.

Future work

Inferno's shell tools are still insufficient. for instance many tricks that work well in unix i.e. sort | uniq -c don't work in Inferno.

A morphological analyzer is very nice. Want to experiment with a variety of dictionary types.

the inferno shell, while general and powerful doesn't provide a nice environment for dealing with utf8 tabular output. a utf8 aware "little language" similar to or based on awk would be ideal.

Inferno really needs a font with complete coverage of the unicode set.

Make the system fully concurrent.

Work on incorporating polymorphism correctly. the amount of private data, breaks the interface. especially in terms of polymorphism. can include another internal module, but that adds complexity.

Come up with a way of making tree paths look better when used with a tree path.

[Asahara00] Asahara, M. and Matsumoto, Y., ''Extended models and tools for high-performance part-of-speech tagger'', Proc. of COLING Saarbrücken, Germany 2000.

[Carreras04fos] Carreras, X. and Chao, I. and Padro, L. and Padro, M., ''Freeling: An open-source suite of language analyzers'' Proc. of the 4th LREC 2004.

[Sag02] Sag, I.A. and Baldwin, T. and Bond, F. and Copestake, A.A. and Flickinger, D., ''Multiword Expressions: A Pain in the Neck for NLP'' Proc. of the Third International Conference on Computational Linguistics and Intelligent Text Processing},
pages={1--15},
year={2002},
publisher={Springer-Verlag London, UK}

Semaphores in Plan 9

Sape Mullender

Bell Laboratories
2018 Antwerp, Belgium

*Russ Cox**

MIT CSAIL
Cambridge, Massachusetts 02139

1. Introduction

Semaphores are now more than 40 years old. Edsger W. Dijkstra described them in EWD 74 [Dijkstra, 1965 (in Dutch)]. A semaphore is a non-negative integer with two operations on it, *P* and *V*. The origin of the names *P* and *V* is unclear. In EWD 74, Dijkstra calls semaphores *seinpalen* (Dutch for signalling posts) and associates *V* with *verhoog* (increment/increase) and *P* with *prolaag*, a non-word resembling *verlaag* (decrement/decrease). He continues, *''Opm. 2. Vele seinpalen nemen slechts de waarden 0 en 1 aan. In dat geval fungeert de V-operatie als 'baanvak vrijgeven'; de P-operatie, de tentatieve passering, kan slechts voltooid worden, als de betrokken seinpaal (of seinpalen) op veilig staat en passering impliceert dan een op onveilig zetten.''* (''Remark 2. Many signals assume only the values 0 and 1. In that case the V-operation functions as 'release block'; the P-operation, the tentative passing, can only be completed, if the signal (or signals) involved indicates clear, and passing then implies setting it to stop.'') Thus, it may be that *P* and *V* were inspired by the railway terms *passeer* (pass) and *verlaat* (leave).

We discard the railway terminology and use the language of locks: *P* is *semacquire* and *V* is *semrelease*. The C declarations are:

```
int  semacquire(long *addr, int block);
long semrelease(long *addr, long count);
```

Semacquire waits for the semaphore value **addr* to become positive and then decrements it, returning 1; if the *block* flag is zero, *semacquire* returns 0 rather than wait. If *semacquire* is interrupted, it returns -1. *Semrelease* increments the semaphore value by the specified count.

Plan 9 [Pike et al., 1995] has traditionally used a different synchronization mechanism, called rendezvous. Rendezvous is a symmetric mechanism; that is, it does not assign different roles to the two processes involved. The first process to call rendezvous will block until the second does. In contrast, semaphores are an asymmetric mechanism: the process executing *semacquire* can block but the process executing

* Now at Google, Mountain View, California 94043

semrelease is guaranteed not to. We added semaphores to Plan 9 to provide a way for a real-time process to wake up another process without running the risk of blocking. Since then, we have also used semaphores for efficient process wakeup and locking.

2. Hardware primitives

The implementations in this paper assume hardware support for atomic read-modify-write operations on a single memory location. The fundamental operation is ''compare and swap,'' which behaves like this C function *cas*, but executes atomically:

```
int
cas(long *addr, long old, long new)
{
    /* Executes atomically. */
    if(*addr != old)
        return 0;
    *addr = new;
    return 1;
}
```

In one atomic operation, *cas* checks whether the value **addr* is equal to *old* and, if so, changes it to *new*. It returns a flag telling whether it changed **addr*.

Of course, *cas* is not implemented in C. Instead, we must implement it using special hardware instructions. All modern processors provide a way to implement compare and swap. The x86 architecture (since the 486) provides a direct compare and swap instruction, CMPXCHG. Other processors—including the Alpha, ARM, MIPS, and PowerPC—provide a pair of instructions called load linked (LL) and store conditional (SC). The LL instruction reads from a memory location, and SC writes to a memory location only if (1) it was the memory location used in the last LL instruction, and (2) that location has not been changed since the LL. On those systems, compare and swap can be implemented in terms of LL and SC.

The implementations also use an atomic addition operation *xadd* that atomically adds to a value in memory, returning the new value. We don't need additional hardware support for *xadd*, since it can be implemented using *cas*:

```
long
xadd(long *addr, long delta)
{
    long v;

    for(;;){
        v = *addr;
        if(cas(addr, v, v+delta))
            return v+delta;
    }
}
```

3. User-space semaphores

We implemented *semacquire* and *semrelease* as kernel-provided system calls. For efficiency, it is useful to have a semaphore implementation that, if there is no contention, can run entirely in user space, only falling back on the kernel to handle contention. Figure 1 gives the implementation. The user space semaphore, a *Usem*, consists of a user-level semaphore value *u* and a kernel value *k*:

```
typedef struct Usem Usem;
struct Usem {
    long    u;
    long    k;
};
```

When *u* is non-negative, it represents the actual semaphore value. When *u* is negative, the semaphore has value zero: acquirers must wait on the kernel semaphore *k* and releasers must wake them up.

```
void
usemacquire(Usem *s)
{
    if(xadd(&s->u, -1) < 0)
        while(semacquire(&s->k, 1) < 0){
            /* Interrupted, retry */
        }
}

void
usemrelease(Usem *s)
{
    if(xadd(&s->u, 1) <= 0)
        semrelease(&s->k, 1);
}
```

If the semaphore is uncontended, the *xadd* in *usemacquire* will return a non-negative value, avoiding the kernel call. Similarly, the *xadd* in *usemrelease* will return a positive value, also avoiding the kernel call.

4. Thread Scheduling

In the Plan 9 thread library, a program is made up of a collection of processes sharing memory. A thread is a coroutine assigned to a particular process. Within a process, threads schedule cooperatively. Each process manages the threads assigned to it, and the process schedulers run almost independently. The one exception is that a thread in one process might go to sleep (for example, waiting on a channel operation) and be woken up by a thread in a different process. The two processes need a way to coordinate, so that if the first has no runnable threads, it can go to sleep in the kernel, and then the second process can wake it up.

The standard Plan 9 thread library uses rendezvous to coordinate between processes. The processes share access to each other's scheduling queues: one process is manipulating another's run queue. The processes must also share a flag protected by a spin lock to coordinate, so that either both processes decide to call rendezvous or neither does.

For the real-time thread library, we wanted to remove as many sources of blocking as possible, including these locks. We replaced the locked run queue with a non-blocking array-based implementation of a producer/consumer queue. That implementation is beyond the scope of this paper. After making that change, the only lock remaining in the scheduler was the one protecting the ''whether to rendezvous'' flag. To eliminate that one, we replaced the rendezvous with a user-space semaphore counting the number of threads on the queue.

To wait for a thread to run, the process's scheduler decrements the semaphore. If the run queue is empty, the *usemacquire* will block until it is not. Having done so, it is guaranteed that there is a thread on the run queue:

```
// Get next thread to run
static Thread*
runthread(void)
{
    Proc *p;

    p = thisproc();
    usemacquire(&p->nready);
    return qget(&p->ready);
}
```

Similarly, to wake up a thread (even one in another process), it suffices to add the thread to its process's run queue and then increment the semaphore:

```
// Wake up thread t to run in its process.
static void
wakeup(Thread *t)
{
    Proc *p;

    p = t->p;
    qput(&p->ready, t);
    usemrelease(&p->nready);
}
```

This implementation removes the need for the flag and the lock; more importantly, the process executing *threadwakeup* is guaranteed never to block, because it executes *usemrelease*, not *usemacquire*.

5. Replacing spin locks

The Plan 9 user-level *Lock* implementation is an adapted version of the one used in the kernel. A lock is represented by an integer value: 0 is unlocked, non-zero is locked. A process tries to grab the lock by using a test-and-set instruction to check whether the value is 0 and, if so, set it to a non-zero value. If the lock is unavailable, the process loops, trying repeatedly. In a multiprocessor kernel, this is a fine lock implementation: the lock is held by another processor, which will unlock it soon. In user space, this implementation has bad interactions with the scheduler: if the lock is held by another process that has been preempted, spinning for the lock will not accomplish anything. The user-level lock implementation addresses this by rescheduling itself (with *sleep(0)*) between attempts after the first thousand unsuccessful attempts. Eventually it backs off more, sleeping for milliseconds at a time between lock attempts.

We replaced these spin locks with a semaphore-based implementation. Using semaphores allows the process to tell the kernel exactly what it is waiting for, avoiding bad interactions with the scheduler like the one above. The semaphore-based implementation represents a lock as two values, a user-level key and a kernel semaphore:

```
struct Lock
{
    long key;
    long sem;
};
```

The *key* counts the number of processes interested in holding the lock, including the

one that does hold it. Thus if *key* is 0, the lock is unlocked. If *key* is 1, the lock is held. If *key* is larger than 1, the lock is held by one process and there are *key*–1 processes waiting to acquire it. Those processes wait on the semaphore *sem*.

```
void
lock(Lock *l)
{
    if(xadd(&l->key, 1) == 1)
        return; // changed from 0 -> 1: we hold lock
    // otherwise wait in kernel
    while(semacquire(&l->sem, 1) < 0){
        /* interrupted; try again */
    }
}

void
unlock(Lock *l)
{
    if(xadd(&l->key, -1) == 0)
        return; // changed from 1 -> 0: no contention
    semrelease(&l->sem, 1);
}
```

Like the user-level semaphore implementation described above, the lock implementation handles the uncontended case without needing to enter the kernel.

The one significant difference between the user-level semaphores above and the semaphore-based locks described here is the interpretation of the user-space value. Plan 9 convention requires that a zeroed *Lock* structure be an unlocked lock. In contrast, a zeroed *Usem* structure is analogous to a locked lock: a *usemacquire* on a zeroed *Usem* will block.

6. Kernel Implementation of Semaphores

Inside the Plan 9 kernel, there are two kinds of locks: the spin lock *Lock* spins until the lock is available, and the queuing lock *QLock* reschedules the current process until the lock is available. Because accessing user memory might cause a lengthy page fault, the kernel does not allow a process to hold a *Lock* while accessing user memory. Since the semaphore is stored in user memory, then, the obvious implementation is to acquire a *QLock*, perform the semaphore operations, and then release it. Unfortunately, this implementation could cause *semrelease* to reschedule while acquiring the *QLock*, negating the main benefit of semaphores for real-time processes. A more complex implementation is needed. This section documents the implementation. It is not necessary to understand the rest of the paper and can be skipped on first reading.

Each *semacquire* call records its parameters in a *Sema* data structure and adds it to a list of active calls associated with a particular *Segment* (a shared memory region). The *Sema* structure contains a kernel *Rendez* for use by sleep and wakeup (see [Pike et al., 1991]), the address, and a *waiting* flag:

```
struct Sema
{
    Rendez;
    long *addr;
    int   waiting;
    Sema *next;
    Sema *prev;
};
```

The list is protected by a *Lock*, which cannot cause the process to reschedule. The semaphore value **addr* is stored in user memory. Thus, we can access the list only when holding the lock and we can access the semaphore value only when not holding the lock. The helper functions

```
void    semqueue(Segment *s, long *addr, Sema *p);
void    semdequeue(Segment *s, long *addr, Sema *p);
void    semwakeup(Segment *s, long *addr, int n);
```

all manipulate the segment's list of *Sema* structures. They acquire the associated *Lock*, perform their operations, and release the lock before returning. *Semqueue* and *semdequeue* add *p* to or remove *p* from the list. *Semwakeup* walks the list looking for *n Sema* structures with *p.waiting* set. It clears *p.waiting* and then wakes up the corresponding process.

Using those helper functions, the basic implementation of *semacquire* and *semrelease* is:

```
int
semacquire(Segment *s, long *addr)
{
    Sema phore;

    semqueue(s, addr, &phore);
    for(;;){
        phore.waiting = 1;
        if(canacquire(addr))
            break;
        sleep(&phore, semawoke);
    }
    semdequeue(s, &phore);
    semwakeup(s, addr, 1);
    return 1;
}

long
semrelease(Segment *s, long *addr, long n)
{
    long v;

    v = xadd(addr, n);
    semwakeup(s, addr, n);
    return v;
}
```

(This version omits the details associated with returning -1 when interrupted and also with non-blocking calls.)

Semacquire adds a *Sema* to the segment's list and sets *phore.waiting*. Then it attempts to acquire the semaphore. If it is unsuccessful, it goes to sleep. To avoid missed wakeups, *sleep* calls *semawoke* before committing to sleeping; *semawoke* simply

checks *phore.waiting*. Eventually, *canacquire* returns true, breaking out of the loop. Then *semacquire* removes its *Sema* from the list and returns.

The call to *semwakeup* at the end of *semacquire* corrects a subtle race that we found using Spin. Suppose process A calls *semacquire* and the semaphore has value 1. *Semacquire* queues its *Sema* and sets *phore.waiting*, *canacquire* succeeds (the semaphore value is now 0), and *semacquire* breaks out of the loop. Then process B calls *semacquire*: it adds itself to the list, fails to acquire the semaphore (the value is 0), and goes to sleep. Now process C calls *semrelease*: it increments the semaphore (the value is now 1) and looks for a single *Sema* in the list to wake up. It finds A's, checks that *phore.waiting* is set, and then calls the kernel *wakeup* to wake A. Unfortunately, A never went to sleep. The wakeup is lost on A, which had already acquired the semaphore. If A simply removed its *Sema* from the list and returned, the semaphore value would be 1 with B still asleep. To account for the possibly lost wakeup, A must trigger one extra *semwakeup* as it returns. This avoids the race, at the cost of an unnecessary (but harmless) wakeup when the race has not happened.

7. Performance

To measure the cost of semaphore synchronization, we wrote a program in which two processes ping-pong between two semaphores:

> Process 1 blocks on the acquisition of Semaphore 1,
> Process 2 releases Semaphore 1 and blocks on Semaphore 2,
> Process 1 releases Semaphore 2 and blocks on Semaphore 1,

This loop executes a million times. We also timed a program that does two million acquires and two million releases on a semaphore initialized to two million, so that none of the calls would block. In both cases, there were a total of four million system calls; the ping-pong case adds two million context switches. Table 1 gives the results.

		time per system call (microseconds)		
processor	*cpus*	*ping-pong*	*semacquire*	*semrelease*
PentiumIII/Xeon, 598 MHz	1	2.18	1.35	1.91
PentiumIII/Xeon, 797 MHz	2	0.887	0.949	1.38
PentiumIV/Xeon, 2196 MHz	4	0.970	1.38	1.84
AMD64, 2201 MHz	2	1.08	0.266	0.326

Table 1 Semaphore system call performance.

		time per lock operation (microseconds)	
processor	*cpus*	*spin locks*	*semaphore locks*
PentiumIII/Xeon, 598 MHz	1	5.4	5.4
PentiumIII/Xeon, 797 MHz	2	18.2	5.6
AMD64, 2201 MHz	2	22.6	2.5
PentiumIV/Xeon, 2196 MHz	4	43.8	4.9

Table 2 Performance of spin locks versus semaphore locks.

Next, we looked at lock performance, comparing the conventional Plan 9 locks from *libc* to the new ones using semaphores for sleep and wakeup. We ran Doug McIlroy's power series program [McIlroy, 1990], which spends almost all its time in channel communication. The Plan 9 thread library's channel implementation uses a

single global lock to coordinate all channel activity, inducing a large amount of lock contention. The application creates a thousand processes and makes 207,631 lock calls. The number of locks (in the semaphore version) that require waiting (i.e., a semacquire is done) varies wildly. In 20 runs, the smallest number we saw was 127, the largest was 490, and the average was 288.

Table 2 shows the performance results. Surprisingly, the performance difference was most pronounced on multiprocessors. Naively, one would expect that spinning would have some benefit on multiprocessors whereas it could have no benefit on uniprocessors, but it turns out that spinning without rescheduling (the first 1000 tries) has no effect on performance. Contention only occurs some 500 or so times, and the time it takes to spin 500,000 times is in the noise. The difference between uniprocessors and multiprocessors here is that on uniprocessors, the first *sleep(0)* will put the process waiting for the lock at the back of the ready queue so that, by the time it is scheduled again, the lock will likely be available. On multiprocesssors, contention from other processes running simultaneously makes yielding less effective. It is also likely that the repeated atomic read-modify-write instructions, as in the tight loop of the spin lock, can slow the entire multiprocessor.

The performance of the semaphore-based lock implementation is sometimes much better, and never noticeably worse, than the spin locks. We will replace the spin lock implementation in the Plan 9 distribution soon.

8. Comparison with other approaches

Any operating system with cooperating processes must provide an interprocess synchronization mechanism. It is instructive to contrast the semaphores described here with mechanisms in other systems.

Many systems—for example, BSD, Mach, OS X, and even System V UNIX—provide semaphores [Bach, 1986]. In all those systems, semaphores must be explicitly allocated and deallocated, making them more cumbersome to use than *semacquire* and *semrelease*. Worse, semaphores in those systems occupy a global id space, so that it is possible to run the system out of semaphores just by running programs that allocate semaphores but neglect to deallocate them (or crash). The Plan 9 semaphores identify semaphores by a shared memory location: two processes are talking about the same semaphore if **addr* is the same word of physical memory in both. Further, there is no kernel-resident semaphore state except when *semacquire* is blocking. This makes the semaphore leaks of System V impossible.

Linux provides a lower-level system call named futex [Franke and Russell, 2002]. Futex is essentially ‘‘compare and sleep,’’ making it a good match for compare and swap-based algorithms. Futex also matches processes based on shared physical memory, avoiding the System V leak problem. Because futex only provides ‘‘compare and sleep’’ and ‘‘wakeup,’’ futex-based algorithms are required to handle the uncontended cases in user space, like our user-level semaphore and new lock implementations do. This makes futex-based implementations efficient; unfortunately, they are also quite subtle. The original example code distributed with futexes was wrong; a correct version was only published a year later [Drepper, 2003]. In contrast, semaphores are less general but easier to understand and to use correctly.

References

[Bach, 1986]
M.J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1986

[Dijkstra, 1965]
E.W. Dijkstra, ''Over Seinpalen'', *EWD74*, 1965.
(http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF,
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD74.html)

[Drepper, 2003]
U. Drepper, ''Futexes are Tricky,'' published online at
http://people.redhat.com/drepper/futex.pdf.

[Franke and Russell, 2002]
''Fuss, Futexes, and Furwocks: Fast Userlevel Locking in Linux,'' *Proceedings of the 2002 Ottawa Linux Symposium*, Ottawa, Canada, 2002, pp. 479–495.

[Holzmann, 1991]
G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991

[Pike et al., 1991]
R. Pike, D. Presotto, K. Thompson, and G. Holzmann, ''Process sleep and wakeup on a shared memory multiprocessor,'' *Proceedings of the Spring 1991 EurOpen Conference*, Tromsø, Norway, 1991, pp. 161–166.

[Pike et al., 1995]
R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, ''Plan 9 from Bell Labs'', *Computing Systems*, **8**(3), Summer 1995, pp. 221–254

[Plan 9, 2000]
Plan 9 Manual, 3rd edition published online at
http://plan9.bell-labs.com/sys/man

v9fb: A remote framebuffer infrastructure for Linux

Abhishek Kulkarni, Latchesar Ionkov
Los Alamos National Laboratory
{kulkarni,lionkov}@lanl.gov

ABSTRACT

v9fb is a software infrastructure that allows extending framebuffer devices in Linux over the network by providing an abstraction to them in the form of a filesystem hierarchy. Framebuffer based graphic devices export a synthetic filesystem which offers a simple and easy-to-use interface for performing common framebuffer operations. Remote framebuffer devices could be accessed over the network using the 9P protocol support in Linux. We describe the infrastructure in detail and review some of the benefits it offers similar to Plan 9 distributed systems. We discuss the applications of this infrastructure to remotely display and run interactive applications on a terminal while offloading the computation to remote servers, and more importantly the flexibility it offers in driving tiled-display walls by aggregating graphic devices in the network.

1. Motivation

The framebuffer device in Linux offers an abstraction for the graphics hardware so that the applications using them do not have to bother about the low-level hardware interface to the device. Since the framebuffer is represented as a character device, a userspace application can open, read and write to it as a regular file. However, performing several routine graphic device operations like setting the resolution, fetching the color palette involves making use of a device-specific *ioctl* system call. This makes it difficult to export these devices as a network filesystem hierarchy.

Several remote display protocols for exchanging graphics over the network already exist. The widely used X window system in Linux is inherently based on a client-server model and implements the X display protocol to exchange bitmap display content between the client and the server. It, however, has been a target of much criticism since the early days[2] because of its overly complex architecture, lack of authentication in the protocol and the limited configurability in its client-server setup. Exporting raw pixel data of the framebuffer device makes it possible to run a window system on the CPU server. With the recent ongoing work on per-container device namespaces in the Linux kernel, this infrastructure provides the foundation for implementing a multiplexing window system similar to Rio [7] for Linux.

Remote display provides a way to interact with geographically distributed resources which are not within the physical proximity of the user. In addition to being used for remote display, v9fb can also be used in a few other interesting scenarios where it is not possible to use these other protocols. For instance, v9fb provides an alternative to monitoring the boot process of a remote machine in a network. This helps in cluster environments where the nodes are not equipped with a serial console to check the boot activity remotely. The booting node mounts the remote framebuffer device exported by the control node and the console of the node is mapped onto the remote framebuffer.

The main motivation for this infrastructure is to ease the setup of tiled-display walls for modeling and simulation of scientific data. High-resolution displays are increasingly being used for visualization of large datasets stored at a central storage facility. Display walls made out of commodity clusters are closely tied to the display nodes and do not allow for dynamic configurations. Developing simulation and modeling applications for these high-resolution tiled display walls is typically done using message passing libraries, new programming models or software that use proxies to stream graphic commands over the network [11]. v9fb transparently aggregates the graphic devices in a network and exports a network attached framebuffer thus allowing greater flexibility in setting up a visualization cluster. Network-centric visualization is invariably favored since it ensures integrity and security of the data being maintained at

a central location [6]. The application program is provided with a single logical view of the framebuffer device and thus requires no modifications to its code.

2. Introduction

Everything in Plan 9, including the graphics infrastructure, is implemented as a file server [8]. The file metaphor describes a well-defined interface to interact with all the resources in a distributed system. This makes it easy to work with the system, keeping it simple yet powerful. Raster graphics capability in Plan 9 is provided by devices like /dev/draw, /dev/screen and /dev/window. Along with the input and console devices, Plan 9 offers a highly configurable and customizable window system that works equally well over the network [7].

Despite considerable efforts, graphics in Linux remains poorly integrated with the rest of the system. The limitations of running the X server as a super user (root) further allows security loopholes which could be used to compromise the system. The framebuffer device abstraction was introduced in Linux starting with kernel version 2.1.107 [12]. The framebuffer device is an abstraction for the graphics hardware and is responsible for initializing the hardware, determining the hardware configuration and capabilities, allocating memory for the graphics hardware and providing common routines to interact with the graphics hardware. The Linux kernel contains drivers that support several different video hardware devices. The v9fb infrastructure exports the raw framebuffer memory and its operations as files. This model could be further extended to support specialized graphics hardware like Graphics Processing Units (GPUs).

The Linux kernel 2.6 offers support for the 9P protocol in the form of loadable kernel modules [1]. This allows the kernel to communicate with synthetic fileservers using the 9P distributed resource sharing protocol. v9fb leverages this support to implement a pseudo-framebuffer device which acts as an in-kernel 9P client that communicates with a framebuffer fileserver. The framebuffer appears as a regular character device to the applications using it. Every operation on this device is transparently translated into a 9P message that is sent across to the remote framebuffer fileserver. v9fb can work on any of the transport mechanisms like TCP or virtio offered by the 9P2000 implementation in the Linux kernel.

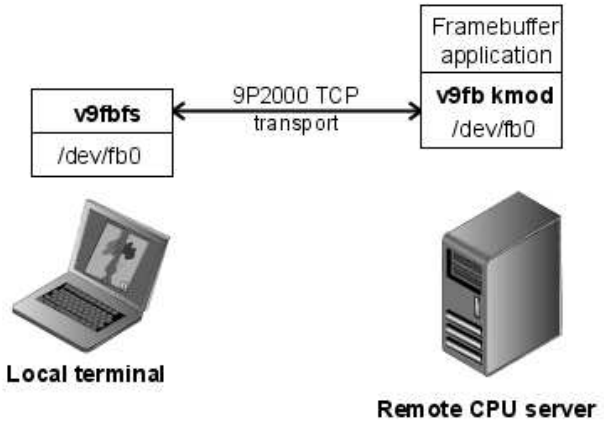


Figure 1: The local framebuffer device is exported by v9fbfs and mounted in the namespace of a remote CPU server which can draw to the remote device

The synthetic framebuffer filesystem *v9fbfs* exports a hierarchy that corresponds to various framebuffer operations which can be executed just by reading off or writing to these files. This also allows the framebuffer devices to be mounted locally and to interact with them as if they were local devices as shown in Figure 1. *v9fbfs* runs on all the display nodes in a visualization cluster and permits a highly-configurable and dynamic setup in which remote display devices can be attached or detached to rendering nodes based on their processing load. v9fb is scalable and can be optimized to support many display devices driving a tiled display wall with an effective resolution of over million pixels.

Coupled with the XCPU cluster management framework [4], this provides a holistic high-

performance visualization environment that is easy to monitor and maintain. It allows a clear segregation of the display nodes from the render nodes and supports heterogeneous display hardware setup as a result of the framebuffer abstraction.

In many cases, simple pixel-based remote display can deliver superior performance than the more complex designs [14] based on other thin-client platform designs. The framebuffer synthetic filesystems allow adding multiple layers above the framebuffer much easier. Compression, encryption or the support for high-level drawing primitives on top of the framebuffer can be easily added without affecting the whole model.

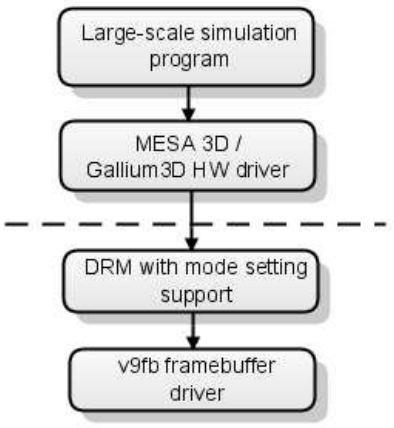


Figure 2: Running simulation and modeling programs directly on a hardware-accelerated framebuffer in absence of the X11 window system

Hardware-accelerated framebuffer makes use of the GPU operations to render graphics on the framebuffer device. Several libraries can use the framebuffer as a target to display high-resolution 2D and 3D graphics. With some of the upcoming changes in the Linux graphics stack like the changes in DRM (Direct Rendering Manager) and Gallium3D, the new proposed architecture for 3D graphics drivers, it would be much easier to display 3D hardware-accelerated graphics on the framebuffer without needing the X server at all as shown in Figure 2. As The framebuffer can be utilized as a drawing surface by the OpenGL applications, the X server and many other graphic drawing libraries like Simple DirectMedia Layer (SDL) or General Graphics Interface (GGI).

The remainder of this paper is organized as follows. In Section 3, we look at some of the related work on remote visualization systems and network-attached framebuffers. Section 4 offers a detailed design overview of the v9fb infrastructure describing how each component in the system interacts with the others. The actual implementation details are discussed in Section 5. We conclude by mentioning some of the future work in the last section.

3. Related Work

A number of existing proprietary solutions for remote visualization are available. Along with parallel graphics rendering toolkits and cluster management tools, these solutions provide a complete software environment for large-scale modeling and simulations. HP's Remote Graphics software, Sun's Visualization System and SGI's Remote Visualization are among many other proprietary solutions that offer remote access to 2D and 3D graphics. Most of these remote display solutions primarily rely on VNC which uses the Remote Framebuffer Protocol (RFB) to exchange display updates over the network.

Tiled display walls usually use pixel-based streaming software to stream the rendered data to the display nodes or a network attached framebuffer. The Scalable Adaptive Graphics Environment (SAGE), developed at the University of Illinois Chicago, is a distributed visualization architecture specifically designed for decoupling graphics rendering from the graphics display [5]. SAGE dispatches visualization jobs for rendering to the appropriate resource in a cluster and streams the resultant pixel data to the remote display. Others, like TeraVision, JuxtaView

also provide an infrastructure for remotely displaying imagery in a cluster.

OpenGL toolkits for cluster-based rendering like Chromium [3] or VirtualGL use techniques like function call interposing to "snoop" the OpenGL protocol and transfer it over the wire to the remote proxies in a cluster. This techniques make it difficult to keep up with the evolving standards and specifications described by OpenGL and add to the overhead in terms of complexity of the architecture.

IBM's Scalable Graphics Engine (SGE-3) offers a hardware-based approach to a network-attached framebuffer[9, 13]. It aggregates the pixel data generated by a rendering cluster to drive a high-resolution tiled display wall. Several other sort-first rendering systems like WireGL allow unmodified graphics application to be scaled to work on a high-resolution tiled-display.

4. Design Overview

The v9fb infrastructure consists of the following entities interacting with each other to make the process of accessing remote framebuffer devices as transparent as possible.

- v9fbfs
- v9fb kernel module
- v9fbaggr
- v9fbmuxfs

v9fbfs is a userspace 9P fileserver that exports a filesystem hierarchy of the framebuffer. The *v9fb kernel module* creates a virtual framebuffer device that acts a 9P client translating all the framebuffer operations into POSIX-like file I/O operations. These calls are forwarded to either to *v9fbfs* or *v9fbaggr* over the 9P protocol. *v9fbaggr* is another userspace 9P fileserver which aggregates the framebuffer resources provided by multiple *v9fbfs* fileservers to export a logical view of a single large framebuffer. *v9fbaggr* offers an exactly similar interface as *v9fbfs* thus making it seamless to communicate with the *v9fb kernel module*.

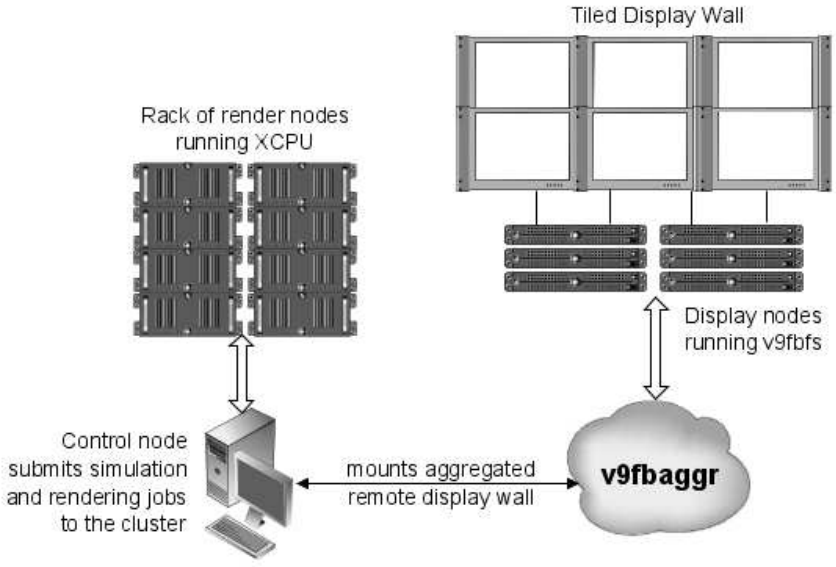


Figure 3: High-performance computing environment for large-scale modeling and simulations using XCPU and V9FB

Figure 3 shows a typical setup of a rendering cluster environment using XCPU and V9FB. At the first glance, the control node appears as a potential bottleneck in this environment. However, the control node only acts as a front-end for submitting jobs. With support for

dynamic namespaces offered by XCPU, the aggregated framebuffer device could be mounted in the namespace of each rendering node which directly writes on to a specific framebuffer of the display wall.

v9fbmuxfs is a userspace 9P fileserver which is almost similar to *v9fbfs*. *v9fbmuxfs* divides a framebuffer into multiple regions exporting each as a logical framebuffer device. It multiplexes the access to each of these regions to simultaneously display the framebuffer output from several clients. Since most modern graphic cards support tiled framebuffers, each tile could be rendered by different machine to achieve a much faster performance.

v9fb offers secure delivery of the display data since it uses the authentication support in 9P2000 protocol. The 9P auth information negotiates authentication between the client and the fileserver before exchange of raw pixel data takes place. The ordered delivery of messages in 9P protocol ensures there is no corruption of the frame pixels. Synchronization has not been taken into account but could easily be added into v9fb.

Synthetic fileservers allow easy addition and removal of functional layers to the architecture. These can further be in the form of fileservers or simple libraries acting on the exported files. For instance, to make efficient use of the network bandwidth the raw pixel data transferred over the network can be compressed before sending. Several performance optimization techniques have been taken into account to achieve a good performance.

4.1. Performance Optimization

v9fb has been designed with low-latency high-bandwidth links in mind where the remote display nodes are connected to the control nodes using a suitably high-speed network interconnect like Gigabit Ethernet. Transmitting raw pixel data over the wire consumes considerable bandwidth for real-time visual applications like video streams and interactive simulations.

4.2. Framebuffer compression

The raw framebuffer data can be compressed using various compression algorithms before transmitting it across the network. This reduces the load on the network, however adds to the overhead of post-processing the data before displaying it on the framebuffer. Compression helps in low-latency links where the network gets overloaded by large bursts of raw pixel data. Video hardware has already started supporting compression at the device level to reduce power consumption [10]. Compression is done on a per-line basis by using a simple compression algorithm like run-length encoding (RLE) or the LZ77 algorithm.

4.3. Framebuffer caching

Caching the framebuffer data at the client can improve the performance in case of non-interactive applications where most accesses involve reading from a static framebuffer. A write to the remotely mounted framebuffer invalidates the cache, and the changes have to be propagated back to the framebuffer fileserver. Introducing caching, however, adds to unmanaged complexity and the performance increases are seldom guaranteed[14].

4.4. Double Buffering

Double buffering at the client and server side can improve performance in most cases. The network-attached framebuffer acts as a back buffer used by the framebuffer fileserver. The scanout buffer acts as a front buffer which represents the memory of the video device. Flipping between the two buffers compensates the network delay to a certain extent and can allow a continuous stream of frames on the video display.

4.5. Multiplexed operations

Multiple clients writing to a single framebuffer pose a potential bottleneck in performance. Multiple reads and writes can be multiplexed at the server with separate threads performing the operations at once. This would significantly add to the performance of *v9fbaggr* which essentially communicates to multiple framebuffer fileservers *v9fbfs* simultaneously. When multiple Treads or Twrites are to be done in parallel, multiple threads are spawned by the server handling these request in parallel.

5. Implementation

5.1. v9fbfs

v9fbfs is a userspace 9P fileserver which scans the local machine for existing framebuffer devices and exports an interface in the form of a file hierarchy given below.

```
/ctl
/data
/mmio
/fscreeninfo
/vscreeninfo
/cmap
/con2fbmap
/state
```

5.2. `ctl` file

The `ctl` file is used to control the framebuffer server and perform some several framebuffer display operations. It supports the following commands :

pandisplay The pandisplay command is used to pan or wrap the display when the X or Y offset of the display have changed.

blank *blankmode* Blank the framebuffer based on the supplied blank mode. This could be used to suspend or power down remote idle displays to save power.

reload Reload the framebuffer filesystem interface. This looks for newly added framebuffer devices and exports them.

5.3. `data` file

The `data` file represents the actual raw framebuffer memory buffer usually represented by the /dev/fb[0-7] device in Linux. Writing to this file writes directly to the framebuffer memory. Similarly, this file is read to fetch the current framebuffer contents.

5.4. `mmio` file

This file represents the memory-mapped IO memory of the framebuffer device. Userspace applications can program the MMIO registers by reading or writing to this file. This can be used to provide hardware acceleration to the framebuffer from the userspace.

5.5. `fscreeninfo` file

Reading from this file retrieves the fixed screen information of the framebuffer graphic device. The device-specific framebuffer information like device type, visual properties, acceleration support, the framebuffer memory length and addresses, the length of the scanline in bytes and the memory-mapped I/O addresses of the device is exported by this file. Fixed information cannot be changed, thus this file cannot be written to.

5.6. `vscreeninfo` file

Reading from this file fetches the virtual screen information of the framebuffer. This can be used to determine the display capabilities of the framebuffer, supported resolutions and color palettes, acceleration flags, bits-per-pixel and the margin and sync lengths among other information. Any of the virtual screen information can be changed by writing to this file.

5.7. `cmap` file

Get/Put the color palette information.

5.8. `con2fbmap` file

Used to map the console onto the framebuffer device and vice versa.

5.9. `state` file

State of the framebuffer device which is used by *v9fbaggr* to maintain synchronization between multiple displays.

Reading and/or writing to a particular file invokes a corresponding framebuffer device-specific operation which talks to the underlying framebuffer device. This provides an alternative to using the ioctl system call for device communication and consequently allows the device to be accessed over the network. This filesystem interface exported by *v9fbfs* can also be mounted as a filesystem using V9FS.

```
$ ./v9fbfs -d
Found framebuffer device /dev/fb0 ...
/dev/fb0 : VESA VGA
Framebuffer device memory from 0xfb000000 to 0xfb600000
Length: 6291456 bytes
Framebuffer MMIO from (nil) to (nil)
Length: 0 bytes
listening on port 8883
```

By mounting *v9fbfs* as a filesystem, framebuffer applications can use this interface to draw to the framebuffer device. With recent support for per-process namespaces in Linux, it allows each process to have an exclusive view of the framebuffer device.

```
$ mount -t 9p 192.168.10.1 /mnt/fb -o port=8883, uname=abhishek, debug=511
$ ls /mnt/fb/fb0/
cmap  con2fbmap  ctl  data  fscreeninfo  state  vscreeninfo
$ cat /mnt/fb/fb0/fscreeninfo
VESA VGA
4211081216 6291456
0 0
2
0 0 0
4096
0 0
```

v9fbfs can handle multiple framebuffer devices (upto 8). It has been implemented using *libspfs*, a library for writing 9P2000 compliant userspace fileservers in Linux. Applications drawing on the top of the framebuffer usually accept a command-line parameter to draw to a different framebuffer device. Alternatively, the global FRAMEBUFFER environment variable can be set to use a different framebuffer device.

5.10. v9fbaggr

v9fbaggr is a userspace 9P server and client typically running on a control node. On startup, v9fbaggr reads a configuration file *v9fbaggr.conf* which specifies the remote framebuffer devices that it needs to aggregate and their relative geometry to export a single logical framebuffer device.

A typical configuration file for a 3x3 tiled display wall is shown below.

```
tile1=192.168.10.40!8883, tile2=192.168.10.64!8883, tile3=192.168.10.67
tile4=192.168.10.41!8883, tile5=192.168.10.65!8883, tile6=192.168.10.68
tile7=192.168.10.42!8883, tile8=192.168.10.66!8883, tile9=192.168.10.69
```

Currently, each newline in the configuration file represents a new row in the geometry of the tiled display wall. Each entry is represented by a nodename followed by its network address and the port on which the server is listening. Use of a rigid data representation format like s-expressions might be considered in the future.

v9fbaggr communicates to the framebuffer fileserver *v9fbfs* running on these machines, fetches their fixed and variable display information and aggregates the remote display resources to provide a logical view of the 3x3 tiled display wall as a single unit of display. Since, v9fbaggr exports an exactly similar interface as that of v9fbfs, application remain transparent of the underlying multiple display devices spread across the network. Framebuffer operations like panning the display, turning the display blank, reloading the fileservers are translated such that

they apply to all the remote framebuffer devices aggregated by *v9fbaggr*. In addition to this, the commands accepted by the `ctl` file also takes an additional parameter, the node name, to which the operation is to be applied.

v9fbaggr implements a memory management unit to translate the virtual address of the aggregated framebuffer to an address of a specific framebuffer device based on the geometry and layout of the tiled display wall. The virtual aggregated framebuffer provides a contiguous linear memory to the application using it. Each memory access to this framebuffer is translated to a 9P read or write to the appropriate framebuffer fileserver. The framebuffer memory of remote framebuffer devices are represented as segments and mapped onto the virtual aggregated framebuffer exported by *v9fbaggr*. Memory accesses to this framebuffer pass through a segment selector which points to the various segment pointers depending on the actual layout of the framebuffer devices. *v9fbaggr* allows unmodified applications and programs to be run on a tiled display wall.

5.11. v9fb kernel module

The v9fb kernel module typically runs on the control node or the head node and creates a pseudo-framebuffer device which translates framebuffer device operations into corresponding 9P calls. The intended use of this kernel module is to mount the filesystem exported by *v9fbaggr* so that it can act as a passthrough framebuffer device to draw transparently to the tiled display wall. It could also be used to mount a single remote framebuffer device for remote workstation display applications.

```
$ modprobe v9fb address=192.168.1.40
$ dmesg | tail -n 2
[118398.958865] v9fb: Enabling remote framebuffer support
[118398.960945] fb1: Remote frame buffer device

$ rmmod v9fb
$ dmesg | tail -n 1
[118401.461253] v9fb: Unmounting remote framebuffer device
```

The kernel module has been written so that v9fb supports existing framebuffer applications without changing them. It translates the device specific ioctl calls into a corresponding 9P call. For instance, to get the virtual screen information of a framebuffer device, the ioctl call to be used is as follows -

```
ioctl(fd, FBIOGET_VSCREENINFO, vscr);
/* vscr is a structure to hold the variable screen
information */
```

The v9fb kernel module translates this into an appropriate 9P operation to read from the `vscreeninfo` file as shown below.

```
<<< (0x8059660) Twalk tag 0 fid 3 newfid 4 nwname 1 'vscreeninfo'
>>> (0x8059660) Rwalk tag 0 nwqid 1 (0000000000000005 0 '')

<<< (0x8059660) Twalk tag 0 fid 4 newfid 5 nwname 0
>>> (0x8059660) Rwalk tag 0 nwqid 0
<<< (0x8059660) Topen tag 0 fid 5 mode 0
>>> (0x8059660) Ropen tag 0 (0000000000000005 0 '') iounit 0

<<< (0x8059660) Tread tag 0 fid 5 offset 0 count 8168
>>> (0x8059660) Rread tag 0 count 110 data 31303234 20373638 20313032
34203736 38203020 300a3332 20300a31 36203820 30203820 38203020 30203820
30203234 20382030 0a300a30 0a343239 34393637

<<< (0x8059660) Tclunk tag 0 fid 5
>>> (0x8059660) Rclunk tag 0
```

```
<<< (0x8059660) Tclunk tag 0 fid 4
>>> (0x8059660) Rclunk tag 0
```

This provides a way to serialize and deserialize device-specific framebuffer calls and obtain the equivalent functionality by marshalling these calls using 9P. Most of the framebuffer `ioctl()` calls are only done at the initialization time and once the display has b een setup properly, majority of the traffic involves reading from and writing to the framebuffer memory. Thus, multiplexing the reads and writes promises considerable performance gains.

5.12. v9fbmuxfs

v9fbmuxfs is similar to *v9fbfs* in a way that it exports the framebuffer device interface as a filesystem. It however creates divides a single framebuffer device into separate regions exporting each as a virtual framebuffer device which a client can write to. Simultaneous rendering and display of a single frame by multiple clients or multiple graphic processing units on a single client can be done with the help of *v9fbmuxfs*. The implementation of *v9fbmuxfs* has not been done and thus qualifies as a future work for this infrastructure.

6. Future Work

Several issues still remain to be dealt with to use v9fb in a production visualization environment. Due to constraints in time, actual performance metrics for driving tiled display walls using v9fb could not be obtained by the time of this writing. Overall performance can be tuned using several ways discussed in Section 4. Apart from this, we are working to support the following features for the v9fb infrastructure.

6.1. Support for input events

Sending keyboard and mouse events over the network forms an integral part of remote display technologies. Currently, v9fb does not address the forwarding of input events over the network. Extending v9fb to support input events is trivial and we have started working on it.

6.2. Hardware-accelerated framebuffer

Due to the proprietary binary-only drivers distributed by major graphic card manufacturing firms like NVIDIA, it has become difficult to use hardware acceleration for the framebuffer. With several initiatives to revamp the state of graphics in Linux, it would soon be possible to use the framebuffer or the in-kernel Direct Rendering Manager (DRM) to draw to the video memory. DirectFB is a thin library which provides hardware graphics acceleration to the framebuffer. A DirectFB extension to v9fb would allow using hardware acceleration to draw high-resolution 3D graphics on the framebuffer device.

6.3. Communication between v9fbfs

One of the most common uses of the tiled display wall is to display high-resolution imagery. Moving and panning of images on the tiled display wall results in resending the pixel data from the control nodes to all the display nodes. This forms a potential bottleneck at the control node. Enabling communication between the individual framebuffer fileservers would increase the performance of interactive applications on the display wall.

7. Conclusion

v9fb provides a novel approach of accessing remote devices over the network in Linux using concepts and ideas employed by Plan 9 since its inception. Withstanding the several difficulties posed by the rigid device subsystem in Linux, this scheme could be easily extended to allow exporting various other devices as a filesystem over the network. v9fb finds various applications in high performance computing and remote visualization technologies. It offers flexibility and configurability leading to dynamic architectures in a large-scale modeling and simulation environment. We are working on several optimizations to this infrastructure to make it capable enough for use in production environments.

References

[1] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space: Using 9p2000 under linux. In *In Proceedings of Freenix Annual Conference*, pages 83–94, 2005.

[2] Don Hopkins. The X-Windows Disaster. *UNIX-HATERS Handbook*.

[3] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002.

[4] Ronald Minnich and Andrey Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *CLUSTER*. IEEE, 2006.

[5] Krishnaprasad Naveen, Vishwanath Venkatram, Chandrasekhar Vaidya, Schwarz Nicholas, Spale Allan, Zhang Charles, Goldman Gideon, Leigh Jason, and Johnson Andrew. Sage: the scalable adaptive graphics environment.

[6] Brian Paul, Sean Ahern, Wes Bethel, Eric Brugger, Rich Cook, Jamison Daniel, Ken Lewis, Jens Owen, and Dale Southard. Chromium renderserver: Scalable and open remote rendering infrastructure. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):627–639, 2008.

[7] Rob Pike. Rio: Design of a concurrent window system. February 2000.

[8] 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.

[9] Prabhat and Samuel G. Fulcomer. Experiences in driving a cave with ibm scalable graphics engine-3 (sge-3) prototypes. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 231–234, New York, NY, USA, 2005. ACM.

[10] Hojun Shim, Naehyuck Chang, and Massoud Pedram. A compressed frame buffer to reduce display power consumption in mobile systems. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 818–823, Piscataway, NJ, USA, 2004. IEEE Press.

[11] Munjae Song. A survey on projector-based pc cluster distributed large screen displays and shader technologies.

[12] Geert Uytterhoeven. The Linux Frame Buffer Device Subsystem. *Linux Expo '99*, 1999.

[13] Bin Wei, Douglas W. Clark, Edward W. Felten, Kai Li, and Gordon Stoll. Performance issues of a distributed frame buffer on a multicomputer. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–96, New York, NY, USA, 1998. ACM.

[14] S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari. The performance of remote display mechanisms for thin-client computing. In *In Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.