iwp9 2025
9
PARIS

11th IWP9 Program

	Thursday, May 22nd	Friday, May 23rd	Saturday, May 24th	Sunday, May 25th
	Doors Open at 09:00 2 rue conté, room 30-1.20	Doors Open at 09:00 2 rue conté, room 30-1.20	Doors Open at 09:00 2 rue conté, **room 30-1.21** HACKATHON The room must be vacated at 18:00	Doors Open at 09:15 **38, rue Saint Sabin** HACKATHON The room must be vacated at 18:00
10: 00	Introductory words			
	Ron Minnich — VMThreads: virtual machines as threads	**Angelo Papenhoff** — Lola: A new Window System for Plan 9		
11: 00	Break	Break		
	Konstantinn Bonnet — Plan 9 Doomed: a review	**Anthony Sorace** — Nile: A More Transparent Window System		
12: 00	Lunch Break	Lunch Break		
13: 00				
14: 00	**J. Frech** — Toiled	**J. Frech** — Amber		
	A. Kozdra — Uglendix	**J. Amoson** — Bitfields		
	P. Lalonde et al. — NIX	**G. G. Muzquiz** — Socarrat		
15: 00	**E. Klein** — WebAssembly	Lightning Talks		
	O. Bernstein — PKI			
	Break	Break		
16: 00	**Brian Stuart** — Plan 9, the Raspberry Pi, and the ENIAC	**T. Laronde** — TeX		
		Jacob Moody — Modern Unicode Requires Modern Solutions		
17: 00	**Geoff Collyer** — What I Saw at the Evolution of Plan 9	Expected at museum around 17:45		
	The room must be vacated at 18:00			
18: 00		Social Event: CNAM Museum guided tour, ends around 19:30		

11th IWP9 in Paris

22–25 May 2025

The 11th International Workshop on Plan 9 was held at the Conservatoire National des Arts et Métiers (CNAM) in Paris, from the 22nd to the 25th of May, 2025.

We would like to thank those without whom this event would not have happened:

- our friends at the CNAM for letting us host this event within their walls, especially Philippe Baumard, Julia Pieltant, and Alexandra Carl;
- the Fondation Pour l'Homme (FPH) for hosting us on Sunday;
- Tomáš Rodr (`https://triapul.cz`) for the T-shirt and cover artwork;
- the program committee members and the Plan 9 Foundation board members for their time and guidance;
- the authors, for their high-quality contributions;
- and of course all the attendees, without whom this whole endeavor would have been quite pointless.

VMThreads: virtual machines as threads†

Ron Minnich

rminnich@gmail.com

ABSTRACT

We describe a new Virtual Machine model, VMThreads, for Plan 9, which allows Virtual Machines to be integrated into a program as threads. Creating a Virtual Machine is as easy as creating a thread; the function call signature is identical to *threadcreate(2).* Currently, VMThreads requires no kernel changes.

Traditional virtualization systems are designed to run heavyweight guest OS images, usually a Linux or Windows distribution, including a disk image. Their disk activity is visible only as a set of IO operations on a virtual disk, not operations on a file system. Guest VMs are frequently accessed over an emulated network connection, as though they were an entirely independent system.

In contrast, VMThreads allows users to run virtual machines of any size, even a single machine instruction‡; a function; a set of functions comprising a driver; or a kernel. Because the guest VM is created as a thread, it is easy to share data structures with it. Threads outside the VM thread can examine and control the guest behavior, allowing a level of debugging that might otherwise require hardware support. Because the VM is visible to the kernel as a process, standard process management tools can be used to study its behavior. Further, it is possible to implement a driver-less kernel, i.e. one which uses system calls, not low-level IO operations, and monitor the VM's use of network, disk, and other resources directly, rather than trying to reconstruct its activity from its use of emulated devices. We discuss the implementation of this model via the #Z device, which maps driver IO functions to system calls, communicated from guest to host via vmcall instructions. We have added this device support to the 9front vmx command.

While the VMs-as-threads model is very simple, it is as capable as more complex systems: we first implemented VMThreads in the Akaros research kernel, and showed that it could support the Google production kernel stack as a guest.

We discuss the implementation of this model on the 9front release of Plan 9. While the implementation is not quite as complete as the Akaros version, it requires no kernel changes.

†License: CC-BY-ND-SA

‡For example, we used a single-instruction VM guest to debug our vmcall support.

Introduction

Virtualization is a hardware capability that allows privileged code to run in a less-privileged context, i.e. a virtual machine (VM). VMs do not stand alone; they require a Virtual Machine Monitor (VMM), also known as a hypervisor, to start them, provide services such as IO to them and, as needed, shut them down. Virtualization, once limited to the exotic domain of IBM mainframes, is ubiquitous on commodity client and server systems, running on everything from phones to cloud servers. Chip vendors have made significant changes over the last 20 years to make virtualization efficient. A guest VM, as measured on cloud servers, can achieve about 91% of the throughput of running on host; in some cases, it can even run faster.

On almost all modern operating systems, virtual machines (VMs) are supported by hardware, with low level control provided by kernels, and presented to VMMs, as a device. VMMs open a device, and operate on it as they would any other real device, such as a network card.

While virtualization is common, and several VMMs are in wide use, writing a VMM is difficult. The device model provided by most kernels (e.g. Linux) is tricky and easy to get wrong, with difficult to understand failures. This interface is difficult enough that, 17 years after virtualization came to Linux, there are only a handful of commonly used VMMs. The API is also broad: virtualization software for, e.g., Linux, is considered compact if it only requires a few thousands of lines of code to run. Virtual Machine software stacks can be very large, with a memory footprint of 10 MiB to 500 MiB.

In this paper we introduce *VMThreads,* a virtualization model which we have incorporated into the Plan 9 thread library. VMThreads, as the name implies, allows users to create and run VMs as a thread in a process. For example, this code starts an infinite loop as a VM thread, running in guest ring 0 on an x86.

```
u8int brdot[] = {0xeb, 0xfe}; // infinite loop: 1: jmp 1b
// start vmthread at brdot; no argument; 1024 byte stack.
vmthreadcreate((void *)brdot, nil, 1024);
```

The API is identical to the threadcreate API from libthread.

The rest of this paper is organized as follows: we provide an overview of virtualization; describe the Akaros vmthread virtualization model which inspired this work; discuss the implementation of this model in Plan 9 *libthread,* and show examples of its use. We also discuss an extension of the 9front vmx command, and a new kernel device, which maps guest device operations such as walk, open, read, and write, to system calls, communicated to the host via the vmcall instruction.

Virtualization

Virtualization allows privileged software to run in an unprivileged context, under control of a host. The software running in the unprivileged context is also called a guest. Virtualization provides services required by applications, that might not be provided by the host, and a security boundary for the the host, protecting it from the guest, e.g. Linux running in a VM on macOS. Virtualization has been available since the 1960s on IBM mainframes.

Until the release of VMWare several decades ago, virtualization had been almost ignored in non-mainframe operations. Since then, however, virtualization has become very widely used, and even taken for granted, on a wide range of systems, including inexpensive laptops such as Chromebooks, up to and including the largest cloud servers.

Once virtualization became more popular, users wished for an alternative to proprietary virtualization software. Xen was an early open source virtualization system for commodity x86 systems, and was widely deployed on Amazon Web Services.

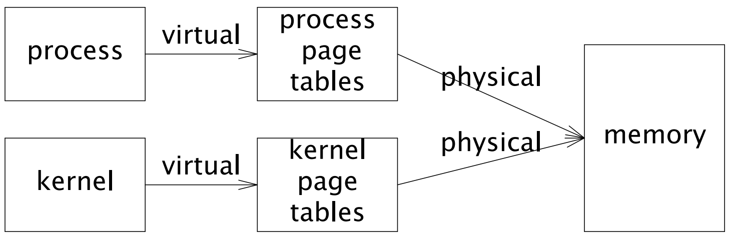
The Xen hypervisor runs kernels as VM guests, calling them "Domains", with one guest kernel in "dom0" being given special privileges to perform IO, since Xen has no drivers of its own. Xen is always the host, and other kernels are always the guest. In Xen, every guest kernel runs at a lower privilege level than the Xen hypervisor.

Following close on the heels of Xen came other systems. Linux KVM (Kernel Virtual Machine) is probably the most widely used†. Unlike Xen, VMMs on Linux run under a driver. In this model, a single general purpose host kernel is more privileged than the virtual machine software, which in turn is more privileged than any VM guest. The driver manages architectural features that enable and control virtualization.

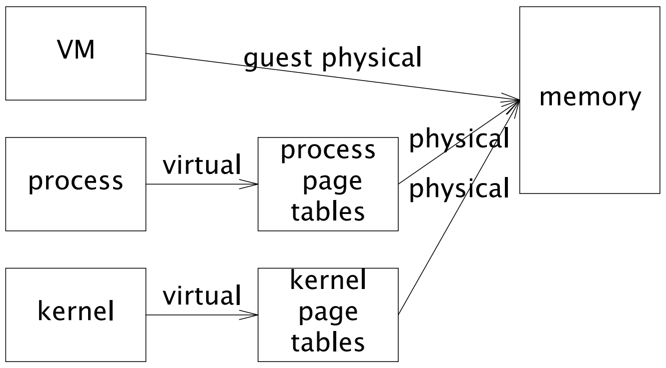
Xen and KVM are structurally very different systems. The arguments over classifying types of VMMs may never end, but, commonly, Xen is called a Type-1 hypervisor, as it is in itself a kernel above all other kernels, whereas KVM, the BSD VMMs, the OSX and Plan 9 VMMs are called Type-2 hypervisors, as they are contained in a general purpose kernel. For the rest of this paper, we restrict the discussion to Type-2 hypervisors. When we use the term VM, we are referring to a guest under a Type-2 hypervisor.

Virtual Machine Memory addressing

In this section we review some basics of VM virtual address spaces. VMs, on x86, can run in 16-, 32-, and 64-bit modes. In 16- and 32-bit modes, virtual addressing is optional. In 64-bit mode, it is mandatory, as required by the architecture. Recall that virtual addressing is enabled by page tables, as shown below.

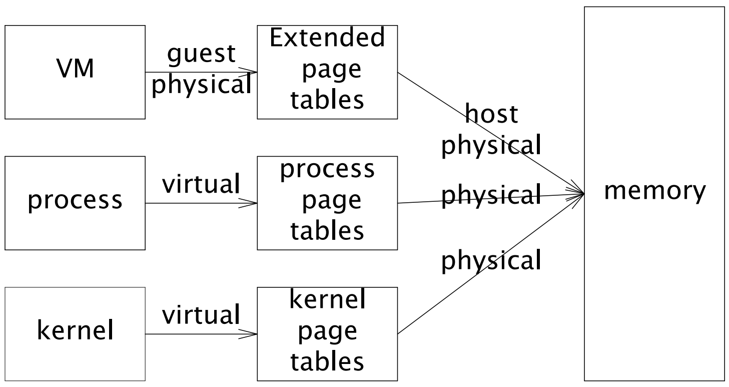


One might wonder: when a VM is in 16- or 32-bit mode, without virtual addressing enabled, can it transparently reference physical memory, as host software in those modes can? Does the picture look like the one below?



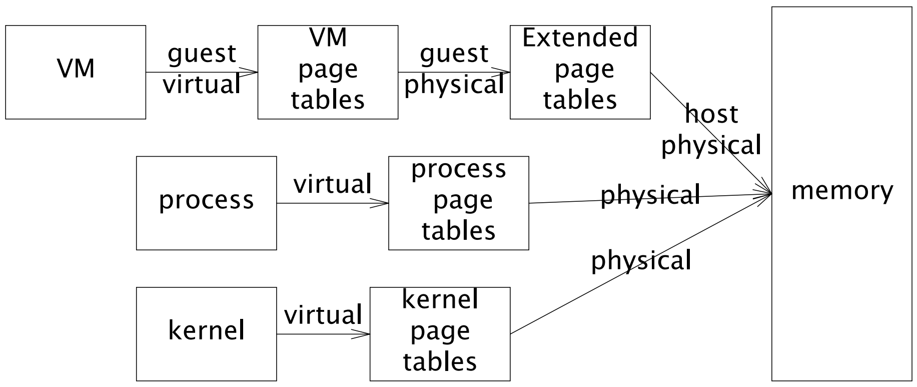
The answer, as you might guess, is "no". Were VMs to work that way, there would be no security at all. Instead, there is a page-table-like structure that maps guest physical addresses to host physical addresses. On Intel, this is called the EPT.

†There are multiple VMM implementations available for Linux, while there are fewer for the BSDs and macOS, and there is only one for Plan 9.

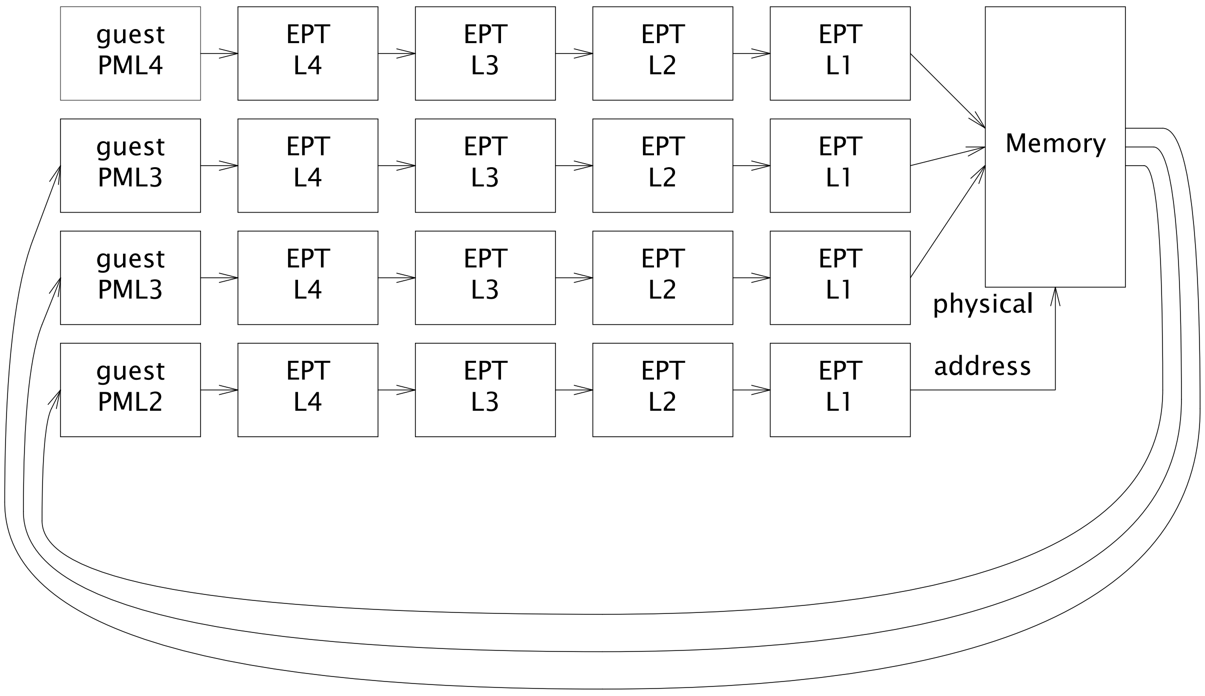


What is shown will work for any 16- or 32-bit code that does not enable virtual addressing. It will not work for 64-bit mode; that requires virtual addressing to be enabled. Further, no 16-bit process can run with paging enabled. The result is that there are at least 5 possible ways to run a guest.

In this paper, we are only concerned with a 64-bit guest, with virtual addressing enabled. We show what that looks like below.



That may look like a lot of table walking. In fact, it is worse than it appears. Every guest page table walk resolves to a guest physical address. That guest physical address must, in turn, be translated to a host physical address. The process is shown below. For the guest page table, we show the page table levels as Page Map Level (PML), ranging from 4 to 1, since PML0 is the guest physical page address. For the guest physical to host physical, we show Extended Page Table (EPT), as levels 4 to 1, where EPT0 is the host physical page address.



Virtual Machine monitors

Running a VM guest requires two software components: the VM code in the kernel; and a so-called Virtual Machine Monitor (VMM). The VMM is a program that directs the kernel to load, run, and manage a guest. As the guest VM runs, it will exit with error codes, interrupts, or when the guest requests services; these VM exits are managed by the VMM. The VMM can be called upon to emulate some instructions; handle timer interrupts; or manage faults. The VMM determines if the VM exit can not be handled, meaning the VM must end; or the VM can be resumed. During the lifetime of a typical VM, millions of VM exits are processed and millions of VM resumes are issued by the VMM.

VMMs can be very complex, and, consequently, can be very large. We know of one proprietary VMM which is a 500 MiB binary. At the same time, a basic Linux VMM, written in C, can be as small as a megabyte. Typically, smaller VMMs can not handle widely varying demands of guests. The larger VMMs have support for more complicated operations, such as BIOS interrupts needed for DOS. A common open source VMM is QEMU, but there are a few others, e.g. the KVMTool, CrosVM, Cloud Hypervisor, and GoKVM tools for Linux.

With few exeptions, the VM API is built around a device model, frequently requiring special privilege to access. VMMs open the VM device and use a file descriptor to control it. The separateness of the device is always apparent. The abstraction is complicated and not convenient.

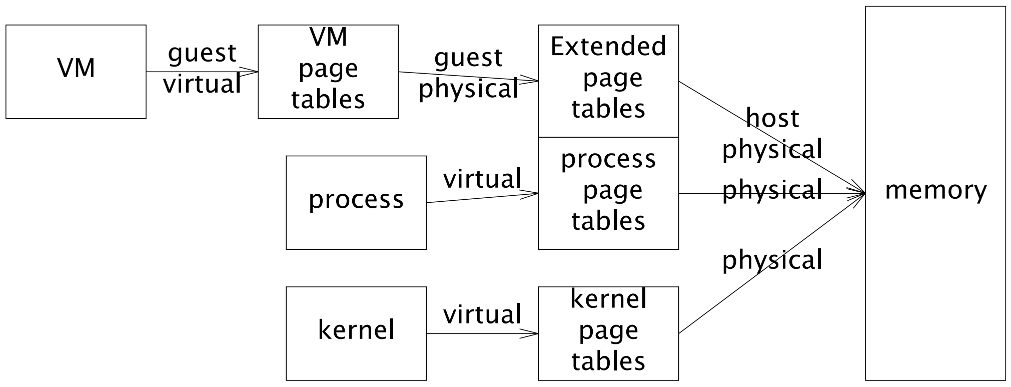
Akaros virtualization: VMs as threads†

As part of our Akaros research at Google, we developed an entirely new VM model, by adding a new program protection level called Ring V. Ring V is almost identical to the x86 Ring 3, a.k.a. user mode, save that it is a VM. When the Akaros kernel exits Ring 0 (kernel) to the program, it checks a flag in the process structure and either does a

†https://github.com/brho/akaros. See the patches starting at ddb9fa78b5d97e4107a9404f4146140ef8bf9a5a from April 1, 2015 and on (yes, April Fool's day!).

traditional exit to Ring 3, using the saved trap context and RETI (return from interrupt) instruction, or to Ring V, using the VM Control Structure (VMCS) context and a VMRESUME instruction. Akaros can decide, each time it resumes a process, whether to resume it in Ring 3 or Ring V. Processes can flip back and forth between the two modes at will.

Akaros takes the model one step further: the guest physical address space is the same as process virtual address space, as shown below:



A VMM can start a kernel by reading the kernel file into memory and starting a vmthread with the kernel entry point as the initial program counter (PC). VMMs can even run user programs in Guest Ring 0, i.e. as kernels. One use of a Guest Ring 0 program would be a device driver, which could safely contain the device I/O in the VM.

How do VMThreads look in practice? Shown below is a function that runs in the guest. It is called thread_entry, and is part of a single file containing both the VMM and the guest code. They can be compiled into one unit because the guest physical addresses and the host virtual addresses are the same. In fact, the guest physical memory is provided directly from the host virtual memory. This code also shows the use of *vmcall,* an instruction which allows a guest to request system-call like operations on the host.

```
static void thread_entry(void *arg)
{
        const char nums[] = "123456789";

        for (int i = 0; i < sizeof(nums); i++)
                vmcall(VTH_VMCALL_PRINTC, nums[i]);
        vmcall(MY_VMCALL_TEST1);
        vmcall(VTH_VMCALL_EXIT, arg, 0, 0, 0);
}
```

The function prints out several numbers, via vmcall, which exits the guest, and resumes in the host program vmcall handler. The parameters to vmcall are placed in registers, by the compiler, following the 64-bit x86 calling convention. There are two standard vmcalls used in this example: VTH_VMCALL_PRINTC and VTH_VMCALL_EXIT; VTH_VMCALL_TEST1 is a custom vmcall implemented in this same program.

Below we show a VMM that runs this function. This example runs 5 VMs simultaneously, i.e. on 5 cores. This is a multi-threaded program running several VMs on several cores at once. It is the technique we use to start a multiprocessor kernel such as Linux: each Linux core (VM guest) is a thread in the host.

```
int main(int argc, char **argv)
{
        #define NR_VTHS 5
        struct vthread *vths[NR_VTHS];
        void *retvals[NR_VTHS];

        /* Tests multiple threads at once */
        for (long i = 0; i < NR_VTHS; i++)
                vths[i] = vthread_create(&vm, thread_entry, (void*)i);
        for (long i = 0; i < NR_VTHS; i++) {
                vthread_join(vths[i], &retvals[i]);
                assert(retvals[i] == (void*)i);
        }

        return 0;
}
```

To reiterate, all the code we are showing is in the same file: the VMM and the guest "kernel" are compiled in one file, one time. In this case, it creates 5 threads with vthread_create, using thread_entry as the function to activate. It then performs a vthread_join. Each of the threads, when it exits with VTH_VMCALL_EXIT, will complete the join.

This API is designed to be familiar to pthreads users. There is one compilation unit; all data and code share an address space; it uses a fork/join model. The only difference is that all the threads run in a VM. Threads can call host functions with the vmcall. Interestingly enough, since the address space is shared, threads can call libc functions to format strings, and then print them with the vmcall.

Moving this model to 9front

There are two aspects of Akaros' virtualization model that require impactful, albeit small (2500 lines total), changes.

The first is the implementation of Ring V. Processes need to record which ring they are returning to; they need to store extra state for Ring V (about 8k); and they need the additional assembly support. These changes are not large, but they are hard to get right. With VM programming on x86 processors, one symptom of incorrect setup is a very fast reset/reboot. Further, they add complexity to the kernel.

Second is the code to implement host process virtual and guest physical identity mapping. In Akaros, this is achieved by having every host page table page paired with a guest page table page -- i.e., every page table page is now 8k, not 4†k. On 9front, we have implemented this "double page table page" change, and whether it is used is controlled with a boot time command line argument. I.e., "double page page tables" do not require recompiling the kernel‡.

Although we have part of the Akaros model working, before making more far reaching changes, we decided to take a less impactful approach, leveraging the way that Plan 9 threads and the 9front vmx device work. Before describing our implementation, we will review Plan 9 threading and the 9front VMX device.

†People tend to worry when they hear this, but the actual impact is quite minor: on an 8 GiB machine the memory for page table pages increases by 16 MiB.
‡Code avaliable at shithub.us.

A quick review of Plan 9 libthread

Plan 9 threads are lightweight, non-preemptive coroutines, running in one or more processes which share memory. Lightweight threads can be created in the current process, via threadcreate; or in an external process, via procrfork. Whether in the same process or a process created by procrfork, all memory is shared, including stacks. The main purpose of procrfork is to allow calls to operations, such as file I/O, that can block the process and hence all the threads. A common programming idiom is to use procrfork for each file descriptor a program creates. Libthread also allows creation of processes which do not share data memory, via proccreate: these share the process code image and also run their own threads, however, they are entirely independent of the other processes. We will not consider that additional external process model here, as it has no bearing on VMThreads.

As mentioned above, in libthread, a single thread in the same process is created by `threadcreate`. We follow this naming convention: VMThreads are created with *vmthreadcreate.*

Threads communicate via channels. A channel is an in-memory pipe which transmits basic scalar types such as bytes, ints, and pointers. When a thread sends on a channel, it will usually be descheduled and another thread can run. Threads can also voluntarily yield the processor with a yield() function.

Threads do not normally yield; sends and receives on chans cause scheduling events, which is how concurrency is supported. For parallelism, additional processes can be added via procrfork. For more information on Plan 9 threading, please see the documentation.

9front vmx device

From the 9front man page:

```
The vmx device supports "virtual CPUs" using the Intel VT-x
extension (a.k.a. VMX instruction set).  This is used by
vmx(1) to implement virtual machines.  Access to the vmx
device is restricted to the hostowner.

The top level directory contains a clone file and numbered
subdirectories representing the allocated virtual CPUs.
```

The device is available in #X. Initially, only a file named clone appears. As VMs are created, numbered directories appear with control files, as in this example, where the first VM has been created. Unlike PIDs, the VM numbers are not reused; as new VMs are created, the number continues to increase.

```
'#X/0'
'#X/clone'
'#X/0/ctl'
'#X/0/fpregs'
'#X/0/map'
'#X/0/regs'
'#X/0/status'
'#X/0/wait'
```

Control of VMs is via commands written to the ctl file. Registers are available from the fpregs and regs files. The map file shows how guest physical memory is mapped to the segment device. Status shows the VM status. Wait can be opened and read; the read call returns when the VM guest status changes (usually via a VMEXIT). For further information, see the man page on 9front.

The wait file information is provided as text records, delimited by newlines. For example, a VM that exits from a vmcall would return:

```
    .vmcall 0x0 pc 0x20091a sp 0x4ffefac ilen 0x3 iinfo 0x0
```

The wait record for vmcall provides the decoded instrucion (vmcall); the PC and SP, and the instruction length. Nothing more is needed to know how to handle the vmcall; the handler need only extract the first parameter from RBP; and succeeding parameters from the stack†.

The VM exit suspends the VM, and returns information about the exit via the wait file, to a process that is reading from it. In current implementations, following libthread convention, that function bundles up the information and writes it to a channel, where it comes under the domain of the lightweight threads. The process is used so that it can block on the wait file read without blocking the lightweight threads‡.

Handling of the vmexit is simple. The line is broken into fields, and the first field is checked against a lookup table:

```
struct ExitType {
        char *name;
        void (*f)(ExitInfo *);
};
static ExitType etypes[] = {
        {"io", iohandler},
        {".cpuid", cpuid},
};
```

Each entry in the lookup table names a class (e.g. "io") or an instruction (.e.g ".cpuid") and a pointer to a handler. The handler is provided with the ExitInfo. The handler decodes the information, implements the operation, and updates the return PC. Here is the code for the hlt instruction:

```
static void
hlt(ExitInfo *ei)
{
        if(irqactive < 0)
                state = VMHALT;
        skipinstr(ei);
}
```

This instruction will set the VM state to VMHALT if there are no instructions active; advance the PC by calling skipinstr; and return.

In the next section, we describe how we implement VMThreads with libthread and devvmx.

VMThreads on 9front

As mentioned, on Akaros, the guest physical address space is mapped 1:1 with the process virtual address space. On Akaros, we call the guest-host mapping EPT=KTP, meaning that guest mappings (EPT) and the host mappings (KPT) all point to the same memory. To keep it shorter, we will call this ID, meaning Identity mapped host virtual and guest physical memory. All that it means is that a guest physical address and a host virtual address point to the same memory; hosts and guests can communicate pointers to each other without having to translate them. In the examples we present below, process virtual memory addresses a 64MiB region, 0x1000000-0x5000000, corresponding to guest physical addresses 0x1000000-0x5000000. A physical guest address in this

†By contrast, on Akaros, a VM exit causes the thread to resume in Ring 3.

‡This same technique is used in the Go runtime, but the process creation is managed by the Go runtime, not by the user.

range points to the same memory as the process virtual address.

Unlike Akaros, we have not implemented complete ID mapping for guest to host. Most of the effect can be obtained by using segattach, to create shared memory regions. The VMM creates a global segment, then directs the VM instance, via the ctl file, to use that segment for part of its memory. The result is that the VMM and the VM are using a common memory segment.

The mapping is created by vmthreadinit. VMthreadinit maps a data area via segattach, and maps two segments from it as specified by two parameters: the size of the low, unshared memory; and the size of a high range of shared memory.

In the examples presented, the user calls vmthreadinit as follows:

```
    vmthreadinit(16*1024*1024, 64*1024*1024);
```

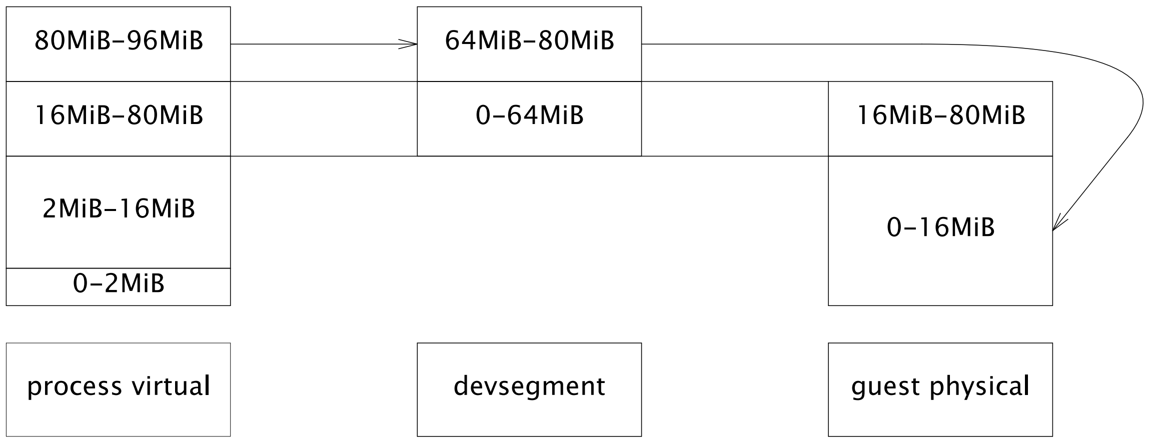
This is enough to allow 1 MiB for low guest physical memory; 1 MiB for the shared frame buffer, and a large shared ID area for the guest and host. Host software can not use the memory range from 0 to 2 MiB, but the guest can use physical memory from 0 to 2 MiB. The memory from 2 MiB to 16 MiB is copied, not shared, between host and guest. Any new guest code, such as a kernel, must be loaded at 16 MiB, and be able to function with a 15 MiB hole starting at 1 MiB. Fortunately, modern kernels, including Plan 9, work fine with this hole. The days of having to load at the 1 MiB boundary are long gone†.

The result is that a large fraction of the memory is shared between guest and host: starting at the 16 MiB boundary, and up to the size of the VM, all memory is shared and all pointers are interchangeable. This is particularly useful for virtio, since it uses ring buffers to pass pointers. Process stack is not accessible from the guest. The guest has a copy of the host text, data, and BSS at the time of vmthread creation, but that is not shared.

Limiting the programs to low 16 MiB might seem to be a problem. For non-Plan 9 systems, fitting a full VMM in 14 MiB would be very difficult. But Plan 9 programs tend to be compact, and the ABI of the #X device helps: the 9front VMM, called vmx, is only 512 KiB, which is anywhere from 10 to 100 times smaller than many Linux VMMs. 14 MiB is more than enough for the foreseeable future. However, should the day come when more is needed, it is easy enough to load the kernel at a much higher address, by adjusting the parameters to vmthreadinit. The decision on where to split the shared and unshared memory is easy to change. In this example, the addresses from 16MiB to 80MiB are shared in in the process virtual and guest physical.

†To have 9front boot at 16 MiB requires a 3-line change to the kernel mkfile:

```
    -KTZERO=0xffffffff81110000
    -APBOOTSTRAP=0xffffffff81007000
    -REBOOTADDR=0x111000
    +KTZERO=0xffffffff80110000
    +APBOOTSTRAP=0xffffffff80007000
    +REBOOTADDR=0x11000
```



Code

Below we present some examples of the threading in action.

The first example shows an infinite loop.

```
            vmthreadinit(16*1024*1024, 64*1024*1024);
            static u8int brdot[] = {0xeb, 0xfe};

            if (vmthreadcreate(brdot, nil, 1024) < 0) {
                    exits("vmthreadcreate failed");
            }
```

For this first example, we show the vmthreadinit call; Since it is required setup code, we will not show it again. VMThreads functions test whether vmthreadinit has been called and will return with an error if it has not.

Vmthreadcreate is called with pointer to brdot, which is the 2-byte infinite loop for x86. This shows the ease with which both a VMM and a guest VM can be set up in the same program. The VM is run by a VMM function, runloop, which starts the VM and calls yield(), allowing other threads to run while the VM runs.

Using vmcall

VMcall allows guests to call functions in the host. Because pointers in the VM and in the host are compatible, any pointers returned via the VM exit, from the VM itself, and passed back via the wait file, can be transparently used by the thread managing that exit.

```
   // This code runs in the host to support vmcall exits.
   static void
   dovmcall(ExitInfo *ei)
   {
           void (*f)(void);
           uvlong *sp;
           f = (void *)rget(RBP);
           sp = (void *)rget(RSP);
           f(sp[1], sp[2], sp[3], sp[4]);
           skipinstr(ei);
   }
```

And add that function to the set of instructions handled in vmexit:

```
static ExitType etypes[] = {
        {"io", iohandler},
        {".cpuid", cpuid},
        {".xsetbv", xsetbv},
+       {".vmcall", dovmcall},
};
```

Then it is just a matter of calling the vmcall instruction in the guest. We show an example below.

First, the library provides a vmcall assembly stub, which is called in the guest:

```
// This code runs in the guest to implement a vmcall.
TEXT vmcall(SB), $0
        BYTE $0xf; BYTE $0x1; BYTE $0xc1
        RET
```

The function itself is declared in C as taking varargs:

```
void vmcall(uvlong, ...);
```

The stub need do no argument setup; the compiler does that as part of normal function calling.

We can define a simple function to be called from the vmcall handler, i.e. in the host:

```
// Host code, callable from the host vmcall handler.
void vhello(void){
        print("hello");
}
```

The guest code vmthread then becomes:

```
// Guest code, set up by a vmthreadcreate call:
void
setter(void *arg)
{
        vmcall((uvlong)vhello);
}
```

The threadcreate is:

vmthreadcreate(setter, nil, 1024);

The VMThread is able to call, via a vmcall, a function in the host.

Extending the use of vmcall

Valid virtual user mode addresses in Plan 9 start at 2MiB and go up. This allows us to use the addresses in the 0-2MiB range for other operations. We currently divide this range as shown in the table below.

vmcall opcodes and arguments

opcode	arguments	function
0-127	None	Print the argument to stdout
0x1000	None	Print registers and the first 2MiB of page table mappings
0x2000-0x2fff	[arg0, arg1, ...]	Call syscall number (opcode-0x2000), with the args
2MiB and up	[arg0, arg1, ...]	Call function pointer in arg0 with the args
0x80000000 | opcode	[arg0, arg1, ...]	Call syscall number ((uint32)opcode), with the args. Intended for guest kernel system calls

Running code in shared memory

Next we show the code which runs in the shared space. First, support code copies the running program to the shared space,to provide a default image:

```
memmove(vmbase+0x200000, (void*)0x200000, (uvlong)sbrk(0) - 0x200000);
```

Note that the code must copy from the start of the program, at 0x200000, to that location offset from vmbase. In any real PC, the low 1MiB of memory has many legacy addresses that are best avoided; further, it is best to align the destination on a 2 MiB boundary.

To call the program, for now, we must offset the addess with vmbase.

```
vmthreadcreate((void *)((u8int*)vmbase+(uvlong)setter), nil, 10240;
```

This is what the guest code looks like in this example:

```
        vmcall((uvlong)vhello, "hi %p %d", arg, 0xaa55);
        vmcall((uvlong)vhello, "hi");
        vmcall((uvlong)vhello, "hi");
```

This code is running in the VM, calling a function in the host. The function is being run by the thread that created the VM.

More complex guest code

Once we are able to call arbitrary code, we can implement more complex guest functions. The function shown below is the first part of ping.

```
void directcall(void *arg)
{
        extern uvlong vmcall(uvlong,uvlong,uvlong,uvlong,uvlong);
        int fd;
        USED(arg);
        vmcall((uvlong)print, (uvlong)"direct call", 0, 0, 0);
        fd = (int) vmcall((uvlong)open,
                          (uvlong)"/net/icmp/clone",
                          (uvlong)ORDWR, 0, 0);
        vmcall((uvlong)print, (uvlong)"fd is %d", fd, 0, 0);
        while(1);
}
```

Using vmcall to implement kernel drivers in the guest

When VMThreads are being used to run a kernel, the common shared memory can provide a new way to implement access to the platform. Since all resources in Plan 9 are accessed via the file interface, it suffices to make the file interface conveniently accessible to the guest -- which, as shown above, we have done. The entire bottom half of the kernel can consist of calls to functions in the low memory. The code is readonly, so corrupting it is not an issue. The data is writable, but it is a copy -- any corruption of the low data memory will result in the untimely exit of the guest.

Using host virtual addresses from the kernel requires that the kernel be linked with vmx, which is inflexible. Instead, we created a device, #Z, which maps device functions to system calls. In this case, all pointers must be passed as kernel physical, not kernel virtual; further, data from user programs must be copied in to kernel space, to ensure it is physically contiguous and to avoid Time Of Check/Time Of use (TOCTOU) problems.

The bounce buffer adds overhead, but a non-bounce-buffer implementation would be far more complex, requiring breaking the read into groups of physically contiguous pages, with the required page table walks, locking, and bugs. For now, the performance of the bounce buffer implementation is more than adequate.

The result is a very different model than has been used in recent virtual machine models: the guest kernel is accessing system resources in the same way as any host process, via file operations; its operations are visible in the same way host process operations are visible. This would rememedy a common problem that comes with VMs: when malware hides behind a virtualization wall, using the opacity of emulated device IO to hide its network operations. We can easily see what is going on behind the virtualization veil.

Comparative directory performance with du

The performance of #Z was always far superior in the earliest version. Using

```
time du -s
```

on different paths, it was anywhere from 2 to 10 times as fast as the virtual disk. However, there was a problem: the IO was blocking, and programs (e.g. telnet) that forked and had several FDs active would fail, because the entire VM was blocked on one read; the other read never had a chance to run. We moved to using ioprocs for IO, and that caused a dramatic drop in performance for disk walking: that same du is ten times slower. Ioprocs have never been a great performance win; we may bring virtio support over from Harvey-OS and try that. Another option is to use blocking reads for directories.

Comparative IO performance

IO to /dev/console can be much faster, as the IO is direct, rather than by device emulation. In this example, 4 MiB of console IO is 700 times faster.

```
term% time dd -if /dev/zero -of /dev/cons -bs 4096 -count 1024
0.00u 0.02s 315.89r      dd -if /dev/zero -of '#Z'/dev/cons ...
term% time dd -if /dev/zero -of '#Z'/dev/cons -bs 4096 -count 1024
0.00u 0.53s 0.53r        dd -if /dev/zero -of #Z/dev/cons ...
term%
```

An example offloading of kernel components to the host

All Plan 9 network access is effected by file I/O. By convention, the files are located in /net or /net.alt. One could build a kernel with no IP stack, and no network drivers, and provide network access as follows:

```
bind '#Z'/net /net
```

At that point, all programs which use the network will transparently use the network of the VMM. The kernel running in the guest can hence be very small.

Debugging guest kernel allocation with host kernel acid

Going beyond using external name spaces for kernel use, we can even externalize core kernel functions to the vmm. Consider the kernel allocator. It is compatible with the libc calling conventions. We could establish a memory pool in the shared area, and then outsource memory allocation to the VMM:

```
void *kernelmalloc(int sz) {
        return vmcall(malloc, sz);
}

void free(void *f) {
        vmcall(free, f);
}
```

Once the guest kernel is using the host (VMM) allocator, we can, further, use the acid storage leak detector to look for kernel misbehavior. The VMM could also look for memory corruption in the arena and notify the user when it occurs. This could be a useful technique for debugging very difficult memory corruption.

When starting a VM becomes equivalent to starting a thread, many possibilities present themselves. The community has gotten into something of a herd mentality, where starting a VM is a Big Deal, and naturally requires large, complex libraries and system management. We argue the exact opposite: starting a VM should be no harder than starting a thread.

Conclusions and future work

We have described vmthreads, an extension to the 9front libthread, which allows guest virtual machines to run as a thread. VMs share memory, and can call host functions directly via vmcall. The current implementation provides most of the capability we had on Akaros: a large memory area is shared, but the code, data and bss segmeents are copied, not shared. Stack is never shared or copied.

Future work includes exploring the model in its current state, to see how it might be used; and, going one step further and enabling full sharing of the data and bss segments. The idea of a driverless guest needs to be explored; replacing emulated devices with system calls should allow higher perfomance. Finally, since the Go threading model derives directly from Plan 9, we can create a vmthreads package for Go. So much of the Plan 9 model already exists in the Go runtime, it will mostly be a matter of adding the vmx device IO to a Go program.

Plan 9 Doomed: a review

Konstantinn Bonnet
qwx@nopenopenope.net

ABSTRACT

Doom's release in 1993 was a pivotal moment in gaming and software history, arguably creating an entirely new genre of video games, spearheading online gaming and pushing the technological boundaries of its time. Today, its community is still very much alive and there are dozens of active software projects surrounding it. Since the public release of the Linux version of its source code, porting it to every platform, however unlikely or over the top that may seem, has practically become a discipline of its own. It should then come as no surprise that Plan 9 has its own port, which 9front has maintained since its inception in 2011. The port has seen many improvements and additions over time, including multiple software utilities written from scratch. This paper aims to showcase the Plan 9-specific aspects of the game port, the unique features and tooling which they have enabled, contributions to other game ports, and future developments.

1. Porting Doom

Doom was initially ported to Plan 9 as `p9doom` by James Tomaschke in 2009 [1], based on the 1997 Linux Doom 1.10 source code release [2]. It is one of -- if not the first -- such example of a real time, interactive and graphically intensive application to be successfully ported to the system. Unfortunately, the original porter's site and repositories have been defunct since at least 2011 and no snapshots are available [3], so we can only speculate what the port looked like. Reportedly, rendering, sound effects and demo playback worked, but key input did not; this initial port may have only taken a few days of work.

At the time, the port must have used *thread*(2) together with *keyboard*(2) and *mouse*(2), or perhaps just *event*(2) for input handling. However, neither interface communicates raw key presses and releases nor modifier keys, instead only producing composed Runes as they are typed. This would make keyboard input impractical for actual gameplay, specifically for player movement or firing a weapon, where actions are maintained active for as long as their key is held. One should note however that Doom, like its predecessors, can use the mouse for turning, firing, as well as movement, acting similarly to a joystick.

Shortly after its inception, 9front saw the inclusion of the newly implemented *kbdfs*(8), which translates raw keyboard scancodes and provides a new `/dev/kbd` file, muxed by *rio*(1), from which a reader receives a stream of key events. It includes an indicator regarding whether a key was pressed or released, the full set of all currently pressed keyboard buttons including modifiers, but also, while pressed, the composed character. This not only allows applications to detect key presses and releases, but to also receive the state of modifier keys, and the last composed character at keyboard repeat rate, while simultaneously altogether avoiding the possibility of stuck keys. A few weeks later, a modified version of `p9doom` using the `/dev/kbd` interface was imported into 9front, solving the input issue. The game at this point was finally functional enough to be playable. Still, because Linux Doom did not include any music playback code, neither did the Plan 9 port. Networking was also yet to be implemented.

2. Engine improvements and extensions

The original Doom engine suffers from a number of bugs, which in addition vary depending on engine version and game version. These range from cosmetic bugs such as graphical or sound glitches, to crashes due to exceeded hard-coded limits, or simply non-portable code and undefined behavior.

In 2014, 9front added an experimental amd64 kernel, and over time it gained support and new kernels for other architectures such as arm64 and mips. Its Doom port has been verified to work on real hardware on all supported platforms provided that they have graphics support. This implied fixing a number of typical issues including pointer to integer conversions, unaligned memory accesses, endianness issues, and other misc. undefined behavior. Doom exclusively uses fixed-point arithmetic, which has made it useful for early testing when developing a new platform.

One important consideration is demo playback. Demos record one or more players' input in a file, and assume a specific game and engine version. To play back a demo, Doom runs just as it would normally, but reading input events at a certain rate from the demo file instead of from the mouse and keyboard. This provides an excellent test to verify that most of the code works correctly, as any deviation will skew computations and accumulate errors, eventually throwing the pseudo-random number generator (PRNG) off and diverging completely away from the original gameplay. Such desynchronization issues are evident in the few surviving video recordings of the original `p9doom`'s demo loop, where the player seemingly ends up butting against a wall and gets stuck in a corner, or dies prematurely. That specific case was due to an undefined evaluation order bug in calls to the PRNG itself [11].

A problem arises when the bugs are wrong behavior of the engine that have nonetheless made their way into one of the commercial releases of the game, and for which thousands of demos (for example, for speed-running or deathmatch competitions) have already been recorded. For example, Doom 2's Lost Souls, flaming skulls which propel themselves toward the player at speed, were meant to bounce off of walls and floors, but a bug in their momentum calculation disabled it [6]. That bug was fixed in later versions of the engine, Ultimate Doom and Final Doom, making Doom 2 demos incompatible. Demo repositories typically identify the game and engine versions in their help files, since the demo file itself doesn't contain enough information. Thus, in 9front this and another similar case were handled by adding bug compatibility switches to re-enable the behavior for proper playback.

In addition to some cosmetic fixes not affecting demo playback, such as truncated sound effects or ones played at the wrong time, a number of static limits have been raised in order to allow loading and playing larger maps in terms of surface area and complexity than originally intended, as is common in today's map making community. Potential extensions to extend the engine to accommodate newer map formats and features that are frequently in use today have been considered as well, but for now 9front's Doom port can be classified as a limit-removing vanilla source port.

3. The unified Doom namespace

Game data in Doom is stored in WAD files (short for "Where's All the Data"). WADs are either *internal* WADs (IWADs), comprising all resources required to run Doom, or *patch* WADs (PWADs), which override or add to the content of an IWAD. Every version of the game has its own IWAD, identified almost solely by filename. For example, the shareware version of the first Doom is named `doom1.wad`, the full version is `doom.wad`, and the later Ultimate Doom release is `doomu.wad`. The game must handle special cases for each, for monster and weapons behavior, animated or toggled textures, level or episode structure, and so on. The game version is essentially determined by checking for a list of known IWADs in the current directory. The only way to set a specific game version is to ensure that the right IWAD is found first. In addition, user files such as the configuration file, savegames and screenshots are all saved in the current working directory as well, but are not identified by version and can be overwritten.

Instead of adding more code to override the game version, IWAD as well as user and game directories, a launcher script *doom*(1) could instead set up the namespace for Doom. Using `/sys/games/lib/doom/$version` as a system game directory containing the right IWAD, and `$home/lib/doom/$version` as a user game directory only containing user data, the script only needs to do the following:

```
mntpt=$home/lib/doom
mkdir -p $mntpt/$version
bind -ac $mntpt/$version $mntpt
if(test -d /sys/games/lib/doom/$version)
    bind -a /sys/games/lib/doom/$version $mntpt
bind -b '{pwd} $mntpt
cd $mntpt
```

In this way, user data such as savegames is stored in version-specific user directories, while each game's data is isolated in system-wide ones, avoiding duplications and all without any code changes. Other files such as PWADs from community map or music packs and other mods, which are inherently tied to a specific game version, can also safely remain in their respective game directories. For example, here is how a system-wide installation and user files for a user *foo* may be organized:

```
/usr/foo/lib/doom      /sys/games/lib/doom
 ├cfg                   ├d2
 ├d2                    | ├doom2.wad
 | ├DOOM00.pcx          | ├hrmus.wad
 | └doomsav0.dsg        | ├hr.wad
 ├plt                   | └scythe.wad
 | └doomsav0.dsg        ├plt
 └ud                    | └plutonia.wad
  ├DOOM00.pcx           └ud
  ├doomsav0.dsg          ├doomu.wad
  └doomsav1.dsg          └sigil2.wad
```

Additional PWADs or demo files can be still be loaded from the current working directory on top of the rest.

4. *wadfs*(4): solving WAD file management once and for all

WAD files are indexed flat arrays of uncompressed files referred to as *lumps*. They store all game assets, including graphics, sounds, music, and so on. In addition, 0-length *marker* lumps are used to delimit *sections* (also referred to as *namespaces*), grouping lumps of a given type together and simplifying lookups. Sections begin and end with a marker (suffixed with _START and _END by convention), except for map sections where it was deemed unnecessary[1]. Some sections may also be divided into numbered subsections. The following is an excerpt of the contents of a Doom 2 IWAD with lump name, start and end offsets and arrows indicating marker lumps:

```
PLAYPAL    00000c:002a0c
COLORMAP   002a0c:004c0c
...
MAP01      00d3d0:00d3d0 ←
THINGS     00d3d0:00d682
...
GENMIDI    3090a4:30bf28
DMXGUSC    30bf28:30d234
DPPISTOL   30d234:30d253
...
SLIME16    de12b8:de22b8
F3_END     000000:000000 ←
F_END      000000:000000 ←
```

WADs were designed specifically to promote the modification of assets and maps, as

[1] Map section markers are only used to jump to and load all map lumps, which are assumed to be at fixed offsets from their section marker.

well as their distribution. While a gamble at the time, this helped create a relatively large and prolific modding community still very much active over 30 years later. Along with the tens of thousands of custom WADs that have been released, many tools have been developed to manage them and to view or modify the specific file formats within. These tools have a complex genealogy spanning several decades and multiple software and hardware platforms, but much of their design has largely remained the same. Most have targeted MS-DOS, then later the MS Windows family of operating systems. For both level and WAD editors, the predominant and preferred approach has been the monolithic all-in-one solution with some sort of graphical interface. WAD editors are typically capable of reading and writing WAD files, recognizing most types of lumps and previewing their contents in format-specific ways. Thus they often re-implement a form of picture viewer, a sound player, and sometimes even rudimentary level editing. On the other hand, level editors often also have to handle the specifics of the WAD file format, but also preview level graphics such as wall textures or object sprites. As a result, while they usually achieve a high degree of user friendliness, such software projects are relatively complex from the very outset, re-implement many functionalities again and again, and often take years to reach maturity, if at all. One recently released tool is WadGadget [10], a terminal-based Linux application for WAD editing with a two-pane interface, able to preview most common lumps and even implements a texture editor. This shows that the same trend still remains to this day.

This is where the Plan 9 approach truly shines. While a monolithic one might make sense in a Windows or Unix-based world, splitting everything into small programs with little to no functional redundancy has proven particularly effective here. WADs are little more than file archives. Indeed, section markers merely delimit the start and end of what could be seen as a subdirectory in an otherwise flat list of files. Given the relatively direct correspondence with a directory tree structure, a WAD filesystem seemed like an obvious choice in this environment, hence the addition of *wadfs*(4) which serves the contents of a single WAD as a file tree. Here's an excerpt of the contents of a Doom 2 IWAD, with lump name and start and end file offsets on the left, and the corresponding *wadfs*(4) directory tree on the right:

```
         doom2.wad             /mnt/wad
PLAYPAL     00000c:002a0c        ├playpal
COLORMAP    002a0c:004c0c        ├colormap
…                                …
MAP01       00d3d0:00d3d0        ├map01
THINGS      00d3d0:00d682        | ├things
LINEDEFS    00d684:00eac0        | ├linedefs
SIDEDEFS    00eac0:0128be        | ├sidedefs
VERTEXES    0128c0:012ebc        | ├vertexes
SEGS        012ebc:014ae8        | ├segs
SSECTORS    014ae8:014df0        | ├ssectors
NODES       014df0:01630c        | ├nodes
SECTORS     01630c:01690a        | ├sectors
REJECT      01690c:016ac0        | ├reject
BLOCKMAP    016ac0:0183d2        | └blockmap
MAP02       0183d4:0183d4        ├map02
…                                …
P_START     000000:000000        └p
P1_START    000000:000000          ├p1
WALL00_1    94d268:94f9b0          | ├wall00_1
WALL00_2    94f9b0:950388          | ├wall00_2
…                                  …
P1_END      000000:000000          | └p1_end
…                                  …
P_END       000000:000000          └p_end
```

Note that section ends are also represented as files. This is important because lump lookups are performed back to front, from the end of the list of WADs to its start, and the location of some section markers is saved on start up. For PWADs to be able to register new lumps or override old ones in a particular section, they must be able to override the end (and only the end) of that section. In the example below, a PWAD overrides a sprite graphic by placing it inside a new section, which is terminated by an marker (instead of thus extending the section boundary[2]:

```
            doom2.wad
S_START    000000:000000
...
PUNGA0     5d9778:5da3a0
PUNGB0     5da3a0:5dadc4
PUNGC0     5dadc4:5dbdfc
PUNGD0     5dbdfc:5dd838
...
S_END      000000:000000  ←
            punch.wad
SS_START   00000c:00000c
PUNGA0     00000c:002256
S_END      002256:002256  ←
```

In *wadfs*(4), this is accomplished as follows:

```
; games/wadfs /sys/games/lib/doom/d2/doom2.wad
createfile SW18_7: file already exists
; games/wadfs -m /mnt/wad2
; mkdir /mnt/wad2/ss
adding end marker SS_END
; cp /mnt/wad/s/punga0 /mnt/wad2/ss/
; mv /mnt/wad2/ss/ss_end /mnt/wad2/ss/s_end
; cp WAD /sys/games/lib/doom/d2/punch.wad
```

Wadfs(4) also serves two special files, SIG, which is used to set the type of WAD (PWAD or IWAD), and WAD, which is a new WAD file compiled from the current lump hierarchy. Both them and the individual lumps can be used directly by other programs. For example, to play a MIDI file:

```
; games/wadfs /sys/games/lib/doom/d2/doom2.wad
; games/mus /mnt/wad/d_romero | games/midi
```

As described further on, *wadfs*(4) is used in the same manner to expose lumps used in music playback from within Doom. Several *wadfs*(4) instances can be bound on top of each other to override lumps with custom ones, which is fully transparent to Doom itself.

What is important to note is that no other program than *wadfs*(4) itself has to handle any of this. It is as strict as possible about the conventions used and attempts some error recovery for typical mistakes, such as non-matching or missing start or end marker lumps. Its full implementation including write support stands at around 850 lines of code, and currently serves as the centerpiece for all editing, modding, map making and even music playback in Doom itself.

5. Doom graphics and maps

Having established a way to open a WAD and manage the lumps inside of it, a number of utilities have been written and included in the distribution to decode and possibly re-encode some of the assets. These include *games/dpic*(1) and *games/todpic*(1), which decode doom pictures to and from Plan 9 *image*(6) format. Transparency is handled by specifying a color to use as a mask. Wall texture lumps are mirrored, but flipping is delegated to *rotate*(1) instead of re-implementing it. 9front includes a sprite editor, *spred*(1), which can be used in conjunction with a palette in plain-text format to edit the images once extracted. Simple *awk*(1) and *rc*(1) scripts to convert to and from its own and Doom's formats exist as well.

[2] While such conventions have become common practice, the namespacing implementation in the vanilla engine is incomplete and does not work for sprites. This is fixed in most source ports.

Once changes have been made, it is trivial to create a patch WAD and load it into Doom. For example, to change the first sky texture in Doom 2, we first need to resize, crop, rotate and convert to CMAP8, which ends up being a fairly straightforward pipeline. The image needs to be tiled using a tool such as *tweak*(1). Then, we need to create the new WAD, convert the image and insert it. Finally, we can compile and save the PWAD, then launch Doom with it.

```
; png -9t tuttleglenda.png \
    | resample -x 128 -y 128 \
    | crop -r 0 0 256 128 \
    | rotate -l \
    | iconv -c m8 \
    >tuttlesky
# omitted: tile tuttlesky using tweak(1)
; games/wadfs -m /mnt/new
; games/todpic tuttlesky > /mnt/new/rsky1
; cp /mnt/new/WAD tuttle.wad
; doom d2 -file tuttle.wad
```

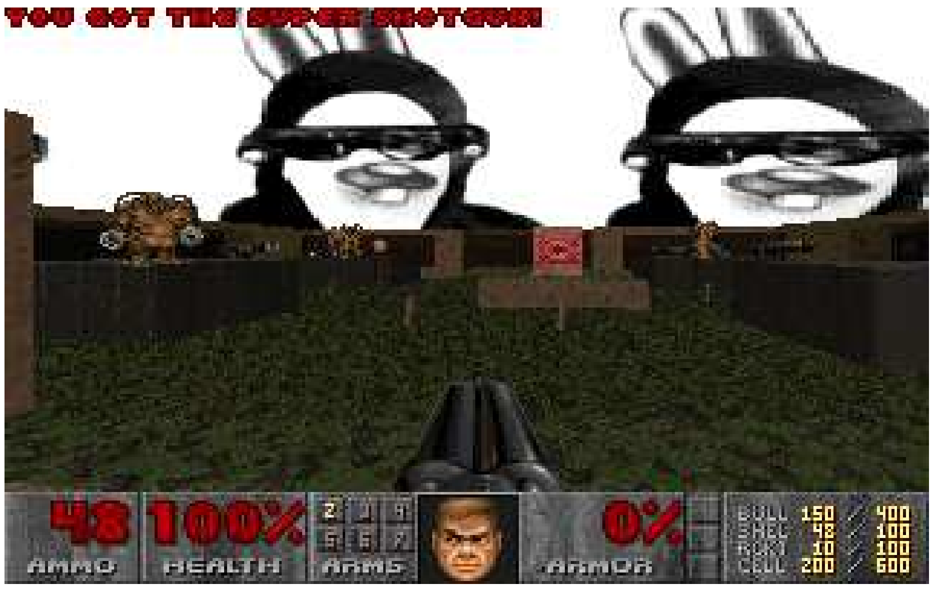


Figure 1. Converting a picture into a sky texture.

Maps in Doom are slightly more complicated to handle. Level data is split into 10 lumps, 5 of which are mandatory and describe the level's geometry and contents: VERTEXES, LINEDEFS, SIDEDEFS, SECTORS and THINGS. A BSP node builder is then invoked to create the remaining ones. Many have been implemented over time, with the earlier ones suffering from limitations and breaking for instance with larger maps. Newer ones address these issues and boast better performance. However, many are written in C++ or other languages, and almost all are licensed under GPL2 or later, though one MIT-licensed nodebuilder built from scratch has appeared recently [4]. A work-in-progress map editor exists, named *dmap*, and will likely be combined with this last nodebuilder in order to finally allow map editing from within Plan 9.

6. Music playback

In Doom, music is played from standard MIDI files or more commonly from MUS files, an in-house format [7]. MUS can be likened to a cut-down version of the MIDI standard. WAD files contain backend-specific lumps such as DMXGUS (for Gravis Ultrasound) and GENMIDI (for OPL3-compatibles), which are meant for configuring a sound card's instrument sounds banks. The DOS games supported multiple types of sound cards, but the most well known music recordings are probably the ones played through a Roland SC-55 with General MIDI. Sound synthesis from MIDI files can be accomplished in software as well, and in many different ways: emulating the sound cards, using generic soundfonts and libraries, or even by simply sampling waveforms at given frequencies.

Because the commercial releases used a proprietary sound library for music playback (DMX) [5], the official source code release has all music code stubbed out. No Doom port would truly be complete without music playback and thus it was felt that it was important to implement it. The code has to read and interpret lumps in MUS or MIDI format and synthesize 16-bit stereo samples at a 44.1 kHz sampling rate, mix it together with 8 or more digital sound effect channels, then write to `/dev/audio`. Audio processing and the write must fit in one frame at Doom's 35 Hz refresh rate or else risk either lowering the game's overall frame rate or causing underruns and choppy audio.

Given that MIDI file players would be useful outside of Doom as well, a natural solution seemed to be to implement MUS and MIDI playback as standalone programs that are executed in their own process, streaming audio samples back via a pipe. *games/midi*(1), a small and simple MIDI player using only square waves, was already added in 2012 and only needed to also be able to write to a pipe. In addition, since the MUS format is a strict subset of MIDI, rather than implementing its own player or supporting multiple formats, a full MUS to MIDI converter seemed like the best option. Doing so proved trivial and resulted in the inclusion of *games/mus*(1). Doom itself would then only need to fork and execute a new process, read samples in from a pipe, and mix them with any sound effects currently playing. With forking processes being fast, and closing the pipe being enough to interrupt and terminate the child process, the code added would barely exceed 100 lines.

Using *games/midi*(1) worked well enough, but its musical quality left much to be desired, particularly since all instruments use the same waveform. It also only implements the bare minimum of the specification and ignores most types of MIDI events such as note bends. This led to the implementation of *games/opl3*(1), an OPL3 sound chip emulator which reads instructions on stdin, synthesizes audio and writes it to stdout, and *games/dmid*(1), a more capable MIDI player using the GENMIDI lump to configure an OPL3 instrument bank, streaming instructions for the chip on stdout. 9front now ships with the shareware version of Doom, to ensure that *games/dmid*(1) can also be used as a general purpose MIDI player without additional downloads. Furthermore, it was reasoned that instead of hard-coding the MUS/MIDI players that the engine uses, it could just fork and execute an *rc*(1) script. If it exists, this script now dubbed *dmus*(1) provides an easy way for anyone to hook in any MIDI player they wish. Here's what it may look like:

```
#!/bin/rc
if(test -f /lib/midi/sf2/patch93.sc-55.sf2)
    c=(games/sf2mid /lib/midi/sf2/patch93.sc-55.sf2)
if not if(test -f /mnt/wad/genmidi)
    c=(games/dmid '|' games/opl3)
if not
    c=(games/midi -c)
if(~ `{file -m $1} audio/mus)
    c=(games/mus '<' $1 '|' $c)
if not
    c=('<' $1 $c)
eval $c
```

This user script checks if a particular SF2 sound font file emulating a Roland SC-55 exists and starts *games/sf2mid*, another (external) soundfont-based MIDI player. If not, depending on whether or not a GENMIDI lump was found by *games/wadfs*(4), either *games/dmid*(1) or *games/midi*(1) is started. This script has also successfully been hooked to a hardware MIDI synth with MT-32 emulation, the Serdaco TM32, by streaming MIDI to a USB interface(3). As a bonus, because it receives the name of the music

(3) *games/mid2s*(1) can stream MIDI events in real time into the endpoint of a USB MIDI dongle.

lump being played as an argument, the script could just for example select and play MP3 files instead. To our knowledge, no other source port offers such flexibility at virtually no cost.

7. Networking

The Linux Doom source code implements a simple peer-to-peer network game protocol via UDP/IP. It is not difficult to fill in the Plan 9-specific blanks to connect several instances together. In the vanilla code, player 0 assumes the role of master and sets the global server settings such as the starting map and game mode. Every additional player joining the game is connected to player 0 and to all others. At the end of every game frame, each player receives and sends frame update messages to synchronize its state with every other. This network topology together with the lack of any recovery or prediction mechanisms severely limits its practicality. It assumes that all machines run at adequate speeds on a low latency connection and is thus unsuitable for games over the internet. Despite some buffering, if any of the machines are slow to respond (for instance because they take too long to draw to their screen), the game is slowed down for everyone. This implementation might be replaced in future with a more typical client-server model capable of dealing with packet loss and large latencies.

The only remaining question is that of interoperability. Possibly because of the aforementioned limitations and missing features, most source ports supporting network play end up being incompatible with one another. While it would be desirable to implement a network protocol compatible with at least some of the current source ports, specifications are typically not available with a permissive license, or at all. Thus any new implementation might likely be 9front's own.

8. Performance notes

Doom renders to a 320x200 framebuffer using a 256-color palette at a constant 35 Hz frame rate. It needs to mix and write to `/dev/audio` one frame's worth of music and sound effects from up to 32 sound effect channels. It also needs to transmit network packets, update the game's state and other things. The performance bottlenecks resided first and foremost in the drawing code, where a buffer the size and pixel channel format of the screen must be filled from the internal framebuffer and current color palette. Because the game's internal resolution is so small, it was desirable to add dynamic scaling based on window size. It had become apparent then that it would make most sense to split up *draw*(3) requests by scaling scanlines horizontally, loading them one at a time into an image with the `repl` flag set, and letting *draw*(3) take care of vertical scaling by tiling scanlines the desired amount of times:

```
r = rectsubpt(rectaddpt(Rect(0, 0, scale*SCRWIDTH, scale),
    center), Pt(scale*SCRWIDTH/2, scale*SCRHEIGHT/2));
if(scale != oldscale){
    if(rowimg != nil)
        freeimage(rowimg);
    rowimg = allocimage(display, Rect(0,0,scale*SCRWIDTH,1),
        XRGB32, scale > 1, DNofill);
    if(rowimg == nil)
        sysfatal("allocimage: %r");
    oldscale = scale;
}
for(y = 0; y < SCRHEIGHT; y++){
    pal2xrgb(cmap, s, buf, SCRWIDTH, scale);
    s += SCRWIDTH;
    loadimage(rowimg, rowimg->r, (uchar*)buf,
        4*scale*SCRWIDTH);
    draw(screen, r, rowimg, nil, ZP);
    r.min.y += scale;
    r.max.y += scale;
}
flushimage(display, 1);
```

This greatly reduces traffic compared to a request for one single giant image.[4]

This alone proved to not be enough and prompted several additional changes. First, drawing is now done asynchronously by rotating several framebuffers, using channels for synchronization. In this way, Doom can draw at 6x scale (a resolution of 1920x1200) or more without any issue and at full speed, even on older machines, such as Penryn-era Thinkpad laptops. Second, audio processing was also moved to its own process, avoiding low framerates due to stalling in the music pipeline, or any degradation in audio playback. Finally, some arm64-specific code has also been added for faster paletted image conversions, in order to gain adequate performance on such machines.

9. Conclusion and future work

Ever since it was imported into 9front, the Plan 9 Doom port has seen many improvements and enhancements. The use of typical Plan 9 programming techniques rendered many of them exceedingly simple yet very powerful. Performance has also been improved to the point where the game is playable without a hitch on most available machines. The tooling around Doom for editing and modding has grown considerably. Besides some external scripts and tools, of note is also *games/dmxdec*(1), which decodes Doom sound effect lumps.

Abstracting away the concept of WADs by exposing lumps as a simple file hierarchy has proven to be a simple yet very powerful approach. Combined with the use of small and composable single-purpose programs, the quasi-totality of the functionality of typical non-Plan 9 software can be achieved and even exceeded, all with practically no functional redundancy and at nearly no cost in complexity and development time. Most of the tools mentioned were developed in a matter of days or weeks and are considered complete.

Going forward, projects such as *NPE* [8] open the door to porting more recent source ports such as Chocolate Doom [9], filling the gaps where vanilla Doom lacks in features, such as support for games derived from Doom like Strife. This would also open the possibility of smooth online play with non-Plan 9 machines.

Lastly, with recent ports of standalone versions of Duke Nukem 3D, Heretic, Hexen and others, it has become evident that much of the Plan 9-specific code ends up being almost identical across ports. This is also the case for the game console emulators already present in 9front, for which *eui* (emulator UI, currently a single C and header file in `/sys/src/games`) was created, pulling common code for drawing on the screen, input handling, joystick support, writing audio, and so on. It now also includes the same asynchronous drawing method used in Doom. It is therefore conceivable that a `libgame` of sorts could be implemented, greatly simplifying porting efforts.

10. Acknowledgements

wadfs(4), the scripts *doom*(1) and *dmus*(1), *games/mus*(1), *games/dmid*(1), *games/opl3*(1), *games/dpic*(1) and *games/todpic*(1), *games/mid2s*(1), *dmap, sf2mid,* and most of the initial music code were my own work, with many contributions from the rest of the 9front community. None of these contributions or software would have been possible without the help and support of cinap_lenrek in particular. *games/mus*(1), *games/dmid*(1) and later MIDI utilities were largely inspired by aiju's earlier *games/midi*(1).

games/opl3(1) is based on code from the *MAME* arcade emulator. *sf2mid* is based on the *TinySoundFont* library.

[4] It is still unclear where this technique originates from.

11. References

[1] Nine Times blog, ‘‘Plan 9 Doomed’’, *http://ninetimes.cat-v.org/news/2009/03/04/0-9doom*, 2009.

[2] Doom Wiki, ‘‘Doom source code’’, *https://doomwiki.org/wiki/Doom_source_code*, retrieved March 2025.

[3] J. Tomaschke, in reply to a thread on the 9fans mailing list, *https://marc.info/?l=9fans&m=129525193127919&w=2*, 2011.

[4] A. Apted, ‘‘nanobsp, a simple internal node builder for classic DOOM ports’’, *https://gitlab.com/andwj/nano_bsp*, 2023.

[5] Doom Wiki, ‘‘DMX’’, *https://doomwiki.org/wiki/DMX*, retrieved March 2025.

[6] Doom Wiki, ‘‘Demo desyncing caused by bouncing lost souls’’, *https://doomwiki.org/wiki/Demo_desyncing_caused_by_bouncing_lost_souls*, retrieved March 2025.

[7] Modding Wiki, ‘‘MUS Format’’, *https://moddingwiki.shikadi.net/wiki/MUS_Format*, retrieved March 2025.

[8] J. Moody and S. Haflinudottir, ‘‘Portability has outgrown POSIX’’, 10th International Workshop on Plan 9, Philadelphia, 2024.

[9] Doom Wiki, ‘‘Chocolate Doom’’, *https://doomwiki.org/wiki/Chocolate_doom*, retrieved March 2025.

[10] S. Howard, ‘‘WadGadget, curses-based Doom WAD file editor’’, *https://github.com/fragglet/WadGadget,* retrieved March 2025.

[11] plan9front code repository, ‘‘games/doom: fix desyncing demo’’, git commit *e4c3f92c163dec741abeadeca80d6a938561dfb3*, 2015.

Toiled: webserver-internal routing via the filesystem interface

Jonathan Frech ⟨info@jfrech.com⟩

Keywords: **Plan 9, filesystem, Go, internet service, Web**

Abstract

Toiled (/twaldi/) is a work-in-progress server-side Go library which models a multi-domain, multi-protocol internet service atop an abstract Unix-style filesystem.

Basal to Toiled's design is Go 1.16's (released February 2021) `"io/fs".FS` interface which can be seen as a direct descendant of 9P. [Pik+92; CP20] It is in the language of slash-separated names [Pik00] that virtual filesystems are constructed as values to be translated to various internet-inhabiting protocols, first and foremost HTTPS.

Living with bloat

The Web is bloated. Not only its content is at (the grimly-driven might be lead to say ever-increasing) times unsightly, but its design reflects the over thirty years of cobbled-together hackery which is its history. Provoked by such blatant scars in such abundance, one is hard-pressed not to feel an urge to wish for a blank slate where one can design the protocol of one's dreams.

Nevertheless, Gopher is dead, FTP servers, mailing lists and electronic bulletin boards are shutting down and Gemini was nothing more than a hype blip, a mere fad [Bag21]. We nevermore live in a world of independent technologies springing up in and endless veldt of possibility. So it is paramount to accept and not blindly reject the already-present. This renders design a delicate undertaking: not only has one to fight their own arrogance in feature creep, one must additionally not remove a large source of complexity which is here to stay.

How then would it be wise to represent the Web using but the very reduced and feature-starved interface given through `"io/fs".FS`?

As the introductory document for `"io/fs".FS` outlines, Go's interface extension pattern [CP20] is a rich enabler for design. It is through this loose enforcement of API boundaries that system cohesion is encouraged yet not oppressively enforced. Fashioned from this is a dynamic escape hatch for the hodgepodge of features HTTP has to offer whilst reducing *effective complexity* through a conceptually soothing frame.

Toiled tries very hard *not* to be inventive: it picks up `"io/fs".FS` together with a small set of necessary manipulatory functions to combine filesystem values and provides a translation to the internet. Contrast this to `"net/http"`'s internal multiplexer, which adds yet another custom string-based DSL whose idiosyncrasies seem to, too, grow evermore with the years. [Ams24] That is to say, through the construction of filesystem objects as Go structures, one achieves a true imbedding of a DSL into the mother language as opposed to allowing a string-interpreting hook into a DSL.

Nonetheless, an inkling of rebellion does rest in Toiled's ethos: Not fully wanting to throw in the sponge, the Web's destructive potential as an aggressively singular entity is attempted to be fought by not reducing Toiled to but a Web server. Toiled's goal is to unify *internet services* through a filesystem-like view.

Technical/Losing namespace information

When designing towards the goal of embanking complexity, guarding all intricacies of an outside system at all times is impossible. As such, $\mathtt{squash}|_{\mathrm{WEB}} : \{\text{HTTP(S) requests}\} \to \{\text{lexical filenames}\}$ is necessarily not injective. It joins either the `"http"` name component with the host name claimed by the request and the request URI's path or the `"https"` name component with the host name verified through TLS and the request URI's path. All other facets of the request such as URI query parameters or cookies are discarded.

But most of the time, one wants the service to present a homogeneous namespace as to not overburden clients, leveraging widely-understood hierarchical ideas found in slash-separated names. [PW85]

For the few cases where one truly needs to differentiate between requests to detect slash-trailing names, slash-encoded names, look at query parameters or otherwise inspect the request, one can *locally* place a custom handler at designated fibres *in the filesystem*, leveraging the extension pattern alluded to above on `"io/fs".File`:

```
// rough outline of Toiled's [http.Handler] implementation
var (
  chi *tls.ClientHelloInfo
  w   http.ResponseWriter
  r   *http.Request
)
f, err := fsys.Open(squash("https", chi.ServerName, r.URL.Path))
if err != nil { panic(err) }
if h, ok := f.(http.Handler); ok { h(w, r); return }
```

In this sense, the strictness of the `"io/fs"`' interfaces is mellowed by dynamic casting. The rest of the filesystem may remain untouched.

Technical/Overzealous interface interpretation

Different to e. g. "net/http".FileServerFS (which requires every "io/fs".File to additionally implement io .Seeker as a relict of "net/http".File's signature), Toiled tries very hard not to overzealously interpret its abstract filesystem's dynamic type: Range HTTP requests, as an example, are satisfied *when* a file supports io.ReaderAt, with said support announced via the HTTP-intrinsic Vary. Thus, Toiled doesn't mysteriously refuse to serve when non-random-access files constitute the filesystem (an in my experience common quibble when trying to link exotic filesystems in with "net/http"). The only hard requirement beyond "io/fs".FS and descendent types' signatures is (effective) staticity in time[1].

Technical/Permission bits

Toiled ignores permission bits except for the x bit in symbolic links:

```
// rough outline of Toiled's [http.Handler] implementation (cont.)
info, err := f.Stat()
if err != nil { panic(err) }
switch m := info.Mode(); true {
  case !m.IsDir() && path.Join("/", r.URL.Path) != r.URL.Path:
    // 307: redirect to the cleaned path
  case m&^0777 == 0: // 200 "OK": serve
  case m&^0777 == fs.ModeDir: // 307: add suffix "/index.html"
  case m&^0677 == fs.ModeSymlink: // 307 "Temporary Redirect"
  case m&^0677 == fs.ModeSymlink|0100: // 308 "Permanent Redirect"
  default: // 500 "Internal Server Error"
}
```

Note that symbolic link permission bits are seldom respected in the Unix wild but indicate to Toiled to use either a temporary or permanent redirect. As long as redirects are internal to the abstract filesystem and of the same type (temporary or permanent), they are transitively reduced to cut down on client-server round-trips.

Symbolic links are—contrary to e. g. os.DirFS' behaviour, which is resolving everything to a potentially non-tree structure—interpreted as small files containing the exact bytes of the target path in absolute (e. g. "/https/...") or relative clean, slashed ('/') form. [Pik00] When the symbolic link's file is fully read as a string, it equals what a classical Unix OS-mounted filesystem's os.Readlink would have returned.

Note that this file only has a trailing newline when the target path's basename does. The version control system Git stores symbolic links the same way as outlined here.

Technical/Directories

Directories are only used as markers to redirect:

```
"path/to/a/directory" -> "path/to/a/directory/index.html"
"path/to/a/file/"     -> "path/to/a/file"
```

Importantly, "io/fs".FileInfo.IsDir is consulted for this decision and not the presence of a trailing slash (which HTTP preserves). Namespace hygiene to not pollute the *name* with what it names is pivotal for system cohesion. [PW85]

Looking elsewhere, lexicality of names is seldom respected in its entirety. Both WEB browsers (HTTP remains agnostic) and Git imbue the trailing slash with meaning (either identifying ".../" with ".../index.html" or sorting tree object entry names according to a key which adds an imaginary trailing slash on seeing directory mode bits).

Example/In situ development boons

Iterating on domain-spanning services is a joy with Toiled as the domain-path namespace split present in HTTP with its associated TLS certificate minutia is abstracted away: Fresh certificates are lazily registered and (re-)issued on client requests as needed.

Especially for nascent "io/fs".FS receivers which only support the minimal required feature set, not overzealously demanding random access on files heightens the development churn ceiling. It furthermore opens up the design space to hook in data sources which per their identity don't support random access.

Example/The hierarchical namespace often suffices

Although the "net/http".Handler escape hatch exists, many seemingly dynamic services emerge to be imbeddable into a rigid namespace structure, with all of their features passing through the "io/fs".File interface.

One example is DPFS, the dynamic picture filesystem, which facilitates the <picture><source/>*<img/> HTML markup by serving scaled variants in different formats with sanitized metadata of image resources via an "/https/dpfs.jfrech.com/IMG_ID.1280w.webp"-style endpoint. Other endpoint suffixes are ".best.jpg" or ".640.bmp".

[1]*Effective staticity in time* meaning, modulo execution time and not-to-be-inspected receiver properties, every call returns the same it did all previous calls or it degenerates to an error. Such degeneration is not required to persist.

Example/Pipeline

Restricting oneself to unwaveringly respect a small API shuts the door to many HTTP-specialized features, no doubt about it. And when one keeps to only following the established ways [Rit84] for which these intricate special-purpose solutions were designed, losing expressiveness is all one gets.

The true power drowsily behind all of Toiled's efforts are the gains in system cohesion. As an illustrative example, starting from some WEB endpoint serving a Git repository (over the 'dumb' protocol), one views it via `"toiled/httpfs".FS` as a filesystem[2]. Via an `"io/fs".FS`-respecting Git library (Aigoual[3] is chosen here), `"aigoual/repository/bare".FSRepo` interprets said filesystem as a Git repository. Reading a specific commit, `"aigoual".(*Commit).TreeID` is passed to `"aigoual/encoding/tree".FS` and yields again a filesystem from which an `"io/fs".Sub` filesystem is `xfs.Aggregate`-d[4] with a domain prefix into the Toiled-given filesystem.

An HTTP client request then may be realized as a direct view into a Git packfile not residing on the serving machine and never read in full. Intermediate in-memory caches at multiple of the described interface boundaries expedite the implementation for a truly snappy experience. The herein described pipeline has been successfully implemented and functioned as described.

From an entropy waste standpoint, such pipelines are of particular interest: Static site generation, as an example, may be implementable as a filesystem filter, making site compilation a part of the client fetch and obviating HTML artifacts on disk. Harking back to latent dreams about dethroning THE WEB, other protocols such as Gopher may turn out to be less of a burden to support once the focus is shifted away slightly from HTML artifacts.

References

[Ams24] Jonathan Amsterdam. *Routing Enhancements for Go 1.22*. Feb. 13, 2024. URL: `https://go.dev/blog/routing-enhancements#accessed:2024-11-26`.

[Bag21] Jo Bager. "Kleine Rakete ; Projekt Gemini: Das Retro-Web". In: *c't Magazin für Computer-Technik* 2021.6 (Feb. 27, 2021), pp. 128–130. URL: `https://www.heise.de/select/ct/2021/6/2102010535767251719#accessed:2024-11-26`.

[CP20] Russ Cox and Rob Pike. *File System Interfaces for Go — Draft Design*. June 2020. URL: `https://go.googlesource.com/proposal/+/master/design/draft-iofs.md#accessed:2024-11-26`.

[Pik+92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. "The Use of Name Spaces in Plan 9". In: *ACM SIGOPS Conference Proceedings* (Sept. 21, 1992), pp. 1–5. URL: `https://dl.acm.org/doi/pdf/10.1145/506378.506413#accessed:2024-09-03`.

[Pik00] Rob Pike. "Lexical File Names in Plan 9 or Getting Dot-Dot Right". In: *USENIX Conference Proceedings* (June 18, 2000). URL: `https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/general/full_papers/pikelex/pikelex.pdf#accessed:2024-08-18`.

[PW85] Rob Pike and P. J. Weinberger. "The Hideous Name". In: *USENIX Conference Proceedings* (June 11, 1985), pp. 563–568. URL: `https://archive.org/details/1985-proceedings-summer-portland/page/563/mode/1up#accessed:2024-11-26`.

[Rit84] Dennis M. Ritchie. "Reflections on Software Research". In: *Communications of the ACM* 27.8. (Aug. 1984). URL: `https://dl.acm.org/ft_gateway.cfm?id=1283939&type=pdf#accessed:2024-11-24`.

[2] A filesystem which has the unusual property that no name opens to a directory and yet some names open to a file. Although not expressly in the spirit of a filesystem, Git's `info/refs` discovery mechanism is strong enough to smoothen over HTTP's lacking directory discovery.

[3] Aigoual is a work-in-progress Go library reimagining Git.

[4] Package "`xfs`" stands in for any helper package which provides algebraic filesystem operations. Exact syntax or semantics do not matter for the present discussion.

Uglendix: Another Plan9/Linux Distribution

Arkadiusz 'Arusekk' Kozdra
arkadiusz.kozdra@cs.uni.wroc.pl

University of Wrocław, Computer Science Institute
ul. Joliot-Curie 15, 50-383 Wrocław
Poland

ABSTRACT

The new Plan9/Linux software distribution, Uglendix, is a modest-sized set of Linux programs able to run most unmodified Plan 9 user-space binaries. It provides a basic userspace implementation for 28 of 53 Plan 9's system calls and some synthetic filesystems like `/srv` traditionally only present in kernel. To showcase its use, this very paper was typeset under Uglendix. The name is a tribute to Glendix, a prior effort at achieving the same.

By implementing all of the Plan 9 abstractions in userspace, Uglendix can achieve great forward compatibility across Linux kernel versions.

1. Introduction

The most popular computer operating system in use today, excluding mainstream desktop computing, is GNU/Linux. Its origins can be traced back to UNIX (and its clones) — an operating system written specifically with software development in mind. The primary focus influencing its design was big multi-user time-sharing systems with low-performance user terminals, connected over low-bandwidth serial lines, which is embodied into today's design of text interfaces in an embarassingly intimate way [1]. The privileged part of this system is Linux, the kernel that is also the biggest open source project. The userspace part of this operating system is built upon GNU utilities [2]. The two strive to be compliant (where possible) to POSIX, a documented standard over UNIX-like systems.

Plan 9 on the other hand is a reiteration of the same core principles (simplicity, responsibility separation, textual interfaces) taken to the extreme. Created in the 1990s by the same research group that originally created UNIX, it reflected a cleaner approach to how computing has evolved since [3].

I found out about Plan 9 in 2024 and wanted to try my day-to-day development on it, but since it involved mostly Python 3 (not yet ported to Plan 9 to my knowledge), I gave up fairly quickly. Another problem was that the kernel of Plan 9 was missing drivers for my hardware. However, I was amazed by the simplicity at the core of Plan 9, so after learning about Glendix [4] I decided to create a userspace syscall emulation layer, and Uglendix was born.

In the course of development I learned about previous attempts at user-mode emulation of Plan 9 binaries [5][6], but none boasted any milestones further than launching `rc` (the Plan 9 shell program) to my knowledge. Neither of them targeted other architectures or old kernels.

Current milestones achieved by Uglendix are interacting with basic network services (currently limited to unencrypted/unauthenticated connections because of no `devtls` implementation), using fileservers (userspace filesystem programs) like `dossrv` or 9660srv and rebuilding the system. This paper was also typeset under Uglendix. I have not tried running other workflows, they might well work, unless they use graphics or system cryptography.

2. Implementation

The implementation is based on binfmt-misc executable format handler and Syscall User Dispatch mechanism for implementing userspace system call handlers under Linux (and falling back to a different method for older kernels, called seccomp; not as reliable, but sufficient for most uses). After starting, the binary is loaded at the correct address in virtual memory (its text and data), and low memory addresses range is marked as needing syscall handling in userspace. The kernel translates each emulated syscall event to a POSIX signal called sigsys, giving the userspace full insight and control over the machine context before and after the syscall. The signal handler translates most Plan 9 system calls one-to-one to POSIX functions. The rest is implemented as minimal-effort counterparts.

A particular challenge is *stat* and *wstat*(2). Inspired by drawterm [7] and u9fs [8], I decided to take certain decisions sacrificing correctness for implementation simplicity. I also used the code for emulating Plan 9 *read*(2) calls on directories. Numeric user ids are converted to textual with POSIX getpwuid/getpwnam functions, with a fallback to a straight decimal numer (group ids follow the analogy). It works well enough for my current use, but might need more attention.

Another challenge is namespace handling: although Linux provides a way to bind one path to another, there are no native union directories. In Uglendix, they are emulated using a combination of overlayfs and parsing `/proc/self/mountinfo` file in order to unmount overlays and mount other overlays in their place (in-kernel filesystem recursion is very limited). This was necessary for typesetting, surprisingly, since the relevant scripts set up /bin as a union of four directories, too deep for my kernel. Mounting is done via v9fs, native Linux 9p filesystem (9p being the standard remote filesystem protocol on Plan 9, and also the standard inter-process communication interface). This is another place where textual and numeric uids must be converted.

An additional `/srv` fileserver is provided, using FUSE userspace filesystem interface. FUSE was chosen because `devsrv` API is incompatible with 9p: a process is supposed to write a number (referring to a file descriptor) to `/srv/somename`, which captures said file descriptor for later access by other processes reading from or writing to `/srv/somename`. 9p file servers have no reliable way of knowing who talks to them. There is another fileserver for `/net` or in other words `devip`, adapted from drawterm and Plan 9's vnc server.

The extra kernel patches are only optional improvements to the quality of Plan 9's *readdir*(2) and running under user namespaces and chroots (beware: the 9p user namespace patch is a deliberate security hole; exploiting it is left as an exercise to the curious reader).

A unique feature of Uglendix is portability: it currently supports two architectures (amd64 and arm) and can be extended to support more. This can be achieved only because of excellent design of Plan 9 system call interfaces, serializing data in an architecture-independent way.

The Glendix distribution contains a chroot script, which can enter a Plan 9 distribution and launch rc inside. There is also an example script demonstrating a full system bootstrap, starting with just several basic binaries.

3. Comparison to Glendix

A natural comparison to Glendix shows several differences: first, while Glendix was a kernel implementation, tied to Linux 2.6 specifically (thousands of commits ago as of Linux 6.12), Uglendix is an entirely userspace implementation (except for entirely optional Linux patches), independent of Linux version (portable to older kernels using a seccomp hack instead of syscall user dispatch). Second, Uglendix supports multiple architectures (as demonstrated by the arm port) and works in QEMU user-mode emulation with the work-in-progress patches for syscall user dispatch. I plan to help upstreaming the qemu patches. Third, no code patching is required, as the machine register context is fully controlled in the sigsys handler.

4. Future considerations

It might be possible to run a full Plan 9 CPU server under Uglendix. It might be beneficial to extract more fileservers from drawterm for graphics, keyboard and mouse, or write some simpler ones using either direct rendering API provided by Linux, or a portable library like SDL. It might already have been done by something like *Wsys(4)* [9] or plan9port [10]. I have not explored these topics much, as I focused on the simplest goals for the initial release.

5. Conclusion

In conclusion, Uglendix provides sufficient emulation to a do full system rebuild of 9front Plan 9 distribution, and can provide files from fileserver programs such as `dossrv` thanks to an additional `/srv` server. It can serve as a basis for using Plan 9 userspace on a broader set of hardware thanks to wider Linux kernel adoption. You can get latest sources of uglendix from <https://sr.ht/~arusekk/uglendix> and redistribute it as described.

Acknowledgements

Huge thanks to bt (err0.net) for introducing me to Plan 9. Thanks to halfwit (hlfw.ca) for letting me know that there might have been previous related work I should check out.

References

[1] Eric S. Raymond, "ncurses." *Linux Journal* 1995.17es (1995): 1-es.
[2] "GNU's not Unix," https://gnu.org
[3] Rob Pile, et al. "Plan 9 from bell labs." *Proceedings of the summer 1990 UKUUG Conference.* 1990.
[4] Shantanu Choudhary, et al. "Glendix: A Plan9/Linux Distribution." *Proceedings of 3rd International Workshop on Plan 9,* 2008, https://glendix.org
[5] Michael Forney, "Nine: wine for 9," https://github.com/michaelforney/nine
[6] Siegmentation Fault, "9aout: Running native amd64 Plan 9 binaries through Syscall User Dispatch (Linux 5.11 onwards)," https://github.com/forked-from-1kasper/9aout
[7] "u9fs," https://bitbucket.org/plan9-from-bell-labs/u9fs
[8] "drawterm," https://github.com/9fans/drawterm
[9] Jesus Galan Lopez, "Wsys(4): hosted window system." *Proceedings of the 6th International Workshop on Plan 9,* 2011
[10] Russ Cox, "Plan 9 from User Space," https://swtch.com/plan9port

WIP: Nix Reborn

Minnich, Ron
rminnich@gmail.com

Laronde, Thierry
tlaronde@kergis.com

Lalonde, Paul
plalonde@acm.org

May 8, 2025

Abstract

Nix is a derivative of the Plan 9 operating system[4], focusing on many-core performance. It allows tasks to be assigned to a particular core after which that core will not be interrupted or assigned more work until the process yields cooperatively. This leads to a very low overhead for compute on that core. The original implementation of Nix [1] was done as a full fork of the Plan9 environment. Since then hardware has continued to evolve and that branch is no longer a viable system. Futher, 9front has emerged as the Plan9 distribution that "just works", with the addition of many device drivers, kernel structure improvements, improved boot method support, and many useful userland changes. This paper presents a new port of the Nix ideas into 9front. The changes are mostly non-invasive, limited to a dozen or so files. Only one significant change is required to the Mach structure for tracking the scheduling class of a core.

1. Introduction

Nix originated in 2011, a first attempt at examining manycore OS design starting from Plan9. The result was achieved in about one month of effort with a small team[1]. This is also a tribute to the simplicity of the Plan9 architecture. The results were promising but the effort faltered, due to funding, Plan9 licensing constraints, and modern hardware support.

When the processor industry hit a limit when trying to improve efficiency by inceasing frequency improvements had to come from another source: increasing the number of cores, enabling parallel processing. But operating systems have lagged behind. Though all modern operating systems support multi-core systems, no significant modifications to the timesharing model have occurred. As multi-core machines gain more and more cores — now reaching into the thousands in some cases [2] — the question arrises if it remains sensible to treat all cores equally in terms of distribution of user workloads and operating system support work.

Nix was the answer to a question: what if, rather than having CPUs with thousands of cores, each running a kernel and programs; we had, e.g., 32 kernel-capable cores and 1024 cores that could only run programs? What would that system look like?

We learned that "user-only" cores would be hard to implement; it seems we also need some minimal capability to handle a trap. We also learned that it is valuable to have shared memory, which makes it possible to easily switch a program from "running on an application core" to "running on a kernel core" – without this, Nix is much harder to write in a transparent way. Finally, we learned that "it always works" is a key idea – Nix apps designed to work on application cores should always work even if they can only run on kernel cores; and using application cores should be so easy as to be invisible. Users should never need to think about where their code will work, and where it will not: it should just work. As a result, Nix is very transparent to the user. Nix added one new system call – execac – which allows a user to indicate "whatever core is available" or to pick the core to use. Standard exec is layered over execac. Nix also added an rfork option to allow processes to easily switch to running on an application core. Nix also minimally extended rc to allow users to pick a core to run a command on, or a set of cores in a pipeline, one core per pipe element [this code is lost]. With these small changes, Nix became an easy and fast environment in which to run with differentiated cores.

This naturally leads to the modern question: how do we write an operating environment for systems of "differentiated cores", such as x86 CPUs with "performance" cores and "power efficient" cores; or Esperanto with its Maxions and Minions? It seems Nix still has an answer to questions of today.

1.1. Modern Hardware

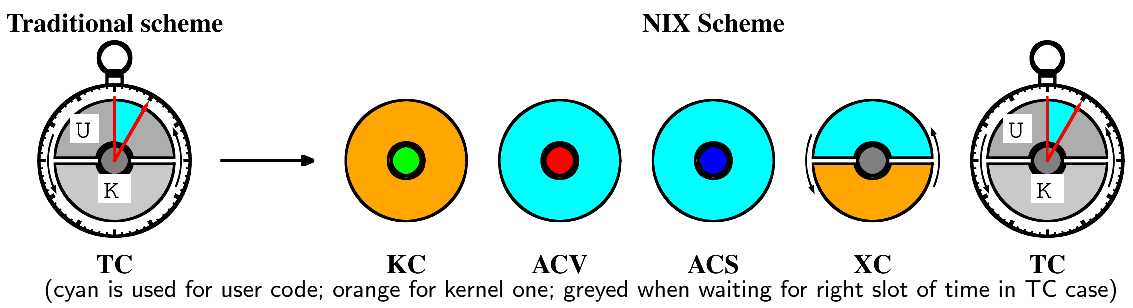
The environment in which Plan 9 now runs has changed. 9front now provides good support for a range of common hardware. CPUs have evolved to where even a commodity PC solutions offer eight to 16 cores. Further, though much HPC work remains tightly tied to the GPU compute model, this model is evolving more closely to modern many-core CPUs with the advent of CPU dies now promising thousands of small cores with significant vector math hardware at each core . And the prices make such a solution affordable by a single individual. Even low cost solutions are now inherently multi-core and NUMA (Non-Uniform Memory Access).

1.2. Timesharing cost

One of the critical tasks of a modern operating system is timesharing: spliting the compute resource among multiple processes, often running at varying priorities. In particular, much of the housekeeping work of the operating system happens on the same cores that are being used for throughput computation. These housekeeping tasks cause perturbation to the running processes. These in turn, though minor on a single compute resource, scale into terrible waste when applied to computing clusters [3].

2. Assigning roles to cores

With the advent of manycore NUMA architectures we can consider dedicating cores to workloads. This designation can be done dynamically, allowing a mix of timeshared cores and application cores. The original Nix implementation introduced:



(cyan is used for user code; orange for kernel one; greyed when waiting for right slot of time in TC case)

- Timesharing Core (TC): a common core running kernel and user code in a time sharing fashion;
- Application Core (AC): a core running user code without any interrupt (even without clock interrupts)—on the illustration, the ACV and ACS are just to emphasize that when using multiple cores, there can be cores with varying capabilities, and a task assigned, because it fits, to a core with vectorial capabilities (ACV), while another task could be assigned to some other unspecified specialized (ACS) one;
- Kernel Core (KC): a core that only runs kernel code on demand;
- eXclusive Core (XC): experimental; a core that is wired to a process instead of the usual reverse, and runs both user and kernel code for this process.

As exemplified by the last experimental role, testing various workloads is necessary to identify the roles needed depending on the nature of the tasks. This would allow maximum flexibility while still maintaining transparency for the user. The communication between cores was done by sending active messages. The initial tests were rather good and the purpose is to restart evaluating both the questions and the solution on now-current hardware.

3. The port to 9front

The initial port was started with a "pull on the thread" inventory. We pulled the Nix tree into a 9front installation, pared away most of the libraries and userland, and started building the Nix kernel in that environment. This identified the core required changes: The Mach structure needed a field to identify the core type, the kernel needed the addition of the Application Core microkernel that runs on Application Cores to schedule the next job or to receive the cooperative system calls and forward them to a Timeshare Core, and the Proc needed support to manage the core residency. We did not implement the idea of a Kernel Core.

Once we had a thorough understanding of the work, we threw it away, and proceeded to make similar changes in the modern 9front kernel. In the end, the bulk of the work was re-working the AC code to work in 9front. Most of this work is just changing the syscall trap, refusing IPI interrupts, and MMU handling for the isolated process.

3.1. Rfork

In the 9front port we chose to prioritize the rfork path. The old execac was a useful development path but treating the AC core as a process resource makes integration more uniform than supporting both a system call variant and the rfork management of a process. Rfork support is via the addition of a new flag $RFACORE$ that indicates the process should run on an Application Core (AC). The corresponding flag $RFTCORE$ (still unimplemented as of this writing) indicates the process should return to Timesharing Coare (TC). These flags can be used alone to transition the current process, or combined with the various process forking flags to modify the process. When RFACORE is set, the Proc's variable $procctl$ is set to $Proc_toac$, which triggers a new case in $_procctl()$ that pushes the process to an AC. RFTCORE does the converse.

3.2. Squidboy and Nix process isolation

Nix isolates processes from the OS using two mechanisms: by refusing external interrupts at Application Cores, and by using a diffirent interrupt and syscall table for Application Cores than for Timesharing Cores. At time of writing we have not yet examined external interrupt routing in 9front. We hope to address this during the 2025 IWP9 hackathon. As far as interrupts and syscalls, the change here was straightforward: we provide a second handler tables $acidthandlers$ in the file nix.s, directly analogous to the usual $idthandlers$ which $squidboy()$ then installs for those cores that are ACs. $Squidboy()$ is the per-core starting entry point of the kernel. Nix modifies Squidboy to read a kernel configuration variable $nixac$ containing a comma-separated list of cores that should be ACs. When starting on a core named in this list $squidboy()$ jumps to $acsched()$ instead of following the usual path through $schedinit()$. $acsched()$ is largely as described in the original Nix paper, waiting in a loop for a process continuation to execute. Once the continuation is launched the loop does not return, and is only re-entered after AC syscall or trap processing.

3.3. Scheduling

The scheduling process is an in the original Nix: when transfering control from TC to AC, the Proc state is set to $Exotic$, which stops the TC from scheduling the process; we call this process stub the "handler" for the Proc, and it remains on the TC waiting to be scheduled again. When a process hits a trap or syscall on the AC that needs to be hanled on the TC, $ready()$ is called to signal that it should be scheduled again on the TC, effectively clearing the $Exotic$ state, and passes on a continuation for the executing process to the TC. The AC then exits handling that Proc and waits for its next unit of work, which is only the same Proc if the syscall or trap was handled successfully and wasn't an exit.

3.4. Sysycalls and Traps

Nix's trap handlers are minimal: in practice only traps which are errors (double-faults, IPIs, IrqTimer are all considered errors if they are delivered to an AC) are handled on the AC. The remainder, along with syscalls, are forwarded to a TC for handling. This is accomplished by calling $ready()$ on the Proc, after which the AC waits for the next workload. Meanwhile, the Proc now being in the ready state causes a return from $sched()$ in the handler process's $runac()$. This return includes the cause of the exit, either a trap or a syscall, and a continuation of the AC process to call to continue the process if needed. The $runacore()$ function then dispatches to the appropriate trap or syscall handler and on successful handling invokes the continuation using $runac()$ once again.

4. Current state

Nix runs again! Two versions exist, one close to the original NIX integrated in 9legacy, and the smaller 9front version described above. It is possible to launch a process in both versions using the acexec command assign a task to a core.

This is still very much a work in progress.

- The $RFTCORE$ rfork flag is not yet implemented/tested.
- Multi-core programs that execute on multiple ACs are untested.
- We do not yet implement Nix's optimistic semaphores or tubes.
- Interrupt routing still needs to be addressed.
- Efficient producer-consumer relationships across ACs probably require a better mechanism than spin-locks; monior/mwait instructions aren't available in userland, so some extensions in that direction are likely required.
- Nix threads and system call queues are not implemented.

5. Bibliography

References

[1] Francisco J. Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ron Minnich, and Enrique Soriano-Salvador. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[2] D Ditzel and Esperanto Team. Accelerating ml recommendation with over 1,000 risc-v/tensor processors on esperanto's et-soc-1 chip. In *Hot Chips 33*, pages 55–55, 2022.

[3] F. Petrini, D.J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 55–55, 2003.

[4] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *EUUG Newsletter*, 10(3):2–11, Autumn 1990.

To hell with WebAssembly

Edouard Klein

Abstract

Work is underway to write a WebAssembly interpreter in Limbo. The interpreter will implement at least the filesystem portion of the WASI specification, which will let the WebAssembly code access the underlying Inferno functionalities. As of May 2025, this WebAssembly runtime is able to run a simple Hello World wasm binary.

1. Rationale and motivation

In the past year, encouraged by Stuart (2023)'s talk in 2023, and by the help from the Plan 9 community [1] I have started using Inferno as the underlying operating system in the various computer science courses I'm giving.

Inferno offers many advantages: simplicity, portability, good design, comprehensive and well-written documentation. It also has the added benefit of being in the blind spot of all the large language models I have tested, which means that the students can not just rely on them, but have to actually read the documentation and think for themselves.

That being said, Inferno has some drawbacks as well, the main one being that it is so unfamiliar to the students that quite a good deal of time must be spent explaining Inferno instead of the course's subject matter. This is a drag, expecially when the subject of the course is something else that "Operating systems".

Even students who have more than a passing familiarity with UNIX (which I must admit to much sadness, seem to be less and less numerous as years go by) are tripped up by Inferno's shell's idiosynchratic syntax. Its input-output redirections being the hardest part to grasp for them.

In a cohort of around 100 students this year, only two managed to write a significant piece of software in Inferno's shell, and another two learnt Limbo. The vast majority just did not write any significant program, and the rest called a Linux binary using the `/cmd` escape hatch Inferno offers when run as a guest, in order to access the host operating system.

To alleviate these issues, I have started writing a WebAssembly interpreter in Limbo.

The goal is to sidestep all the syntax problems my students encounter and let them use whatever language they are already familiar with (mainly Python these days, but runners up are JavaScript, C#, and Java, what a world).

This way, I'll only have to explain the Inferno/Plan 9 trifecta of concepts:

- **Every** resource is a file (*e.g.* the network, the screen, the mouse).
- Each process gets its own view of the filesystem (its *namespace*).
- One can import and export any part of the filesystem, using 9P, locally or remotely.

And students will be able to use it with syntax such as:

```
with open("/dev/pointer") as mouse:
    for event in mouse:
        x, y, _, _ = map(int, event.split()[1:])
        # Now do something with the pointer's position on screen...
```

[1]especially Charles and Ron, thank you !

2. Some context about WebAssembly

JavaScript was added to the browser in the mid-90's. It should have been Scheme or at least Tcl, but alas because of Sun we ended up with JavaScript (Eich 2008).

Since then, billions of dollars and many engineer-centuries have been spent on making JavaScript fast and secure. These last few years, finally, we can certainly say that JavaScript is fast.

My history is a bit fuzzy, but from what I can reconstruct (the best source I have found for this early history is an interview of Alon Zakai: `https://www.youtube.com/watch?v=cv5uQ_hQVE0`), Google was doing its own thing with NaCl in Chrome, trying to find a way to have other progamming languages work on the web, while Mozilla used emscripten and asm.js.

Emscripten could translate LLVM IR to a subset of JavaScript called asm.js that could be more easily optimized as it forewent the most difficult parts, such as on-the-fly type coercion and the garbage collector.

Firefox's JS engine then was optimized to run this asm.js subset of Javascript faster than the rest of the language, and Google's effort in their V8 engine followed a similar path until the stars aligned and all major vendors agreed to standardize on what would become WebAssembly.

WebAssembly is an open standard (World Wide Web Consortium 2024) that describes a bytecode and a text format for software. All the major browsers know how to execute this bytecode efficiently, and all the most popular programming languages have a toolchain that will output WebAssembly, which allows one to use any language to write code for the Web and interface with the existing JavaScript ecosystem.

Another set of standard was then devised, called WASI (WASI Subgroup 2025), which aims at presenting a unified interface for WebAssembly bytecode that does not target browsers. Indeed quite a lot of WebAssembly runtimes have popped up, allowing one to run WebAssembly code outside of the browser. With WASI, these runtimes all agree on how one is to make HTTP calls, or access sockets or files.

We now have a ubiquitous software platform, that implements an efficient VM that provides a normalized interface to the underlying hardware allowing for seamless decentralized interactions between clients.

3. Inferno's time to shine

It is a shame that it took 30 years to coerce what was initially a document-sharing infrastructure to fit the above description. Inferno was right there from the get go.

But Inferno still has a card to play, as it shines particularily well where the web sucks most.

3.1. Security

Client security was never actually deployed on the Web. TLS adoption upticked after the Snowden leaks thanks to initiatives like Let's Encrypt (Stand With Snowden 2017), but client certificates never took off, pushing client authentication too high on the stack, up to the application layer. The only kind of security this smearing of authentication up and down the abstraction layer provides is job security for penetration testers.

Inferno's security, while kind of obsolete now (it would be quite easy to upgrade, though), is very clean and authenticates both client and server at the correct layer (below 9P).

New protocols like Gemini(Gemini developers 2025) try to fix that as well.

3.2. Consistent API

For security reasons, browsers are isolated from the host operating system. This makes sense, but it makes actually using the OS facilities quite a hassle, with vendor-dependant APIs for webcam, or audio, or hardware acceleration access. We also see some timid forays into filesystem access as well.

Inferno, by contrast, can expose these as files, a clean and universal interface. The security and access control is handled using the good old UNIX permissions and ownership system, familiar to almost everyone and that has passed the test of time.

3.3. Decentralization

The web was supposed to be decentralized, but its decentralization is quite relative, as most traffic is handled by a handful of multinational companies whose combined power exceeds those of most of the Earth's nation-states. But this is not a technical problem, and the Web could technically be decentralized on a dime if there was a political will or economic incentives to make it so.

It is still technically easier to do that with Inferno, thanks to its seamless capacity to import other device's peripherals on one's own with 9P, without the local processes having to care nor know about it.

4. Scope and implementation

For the project to be demo-able at the conference, the scope has been reduced to the bare minimum. The target is to have a simple WebAssembly interpreter (no compilation down to Dis bytecode), with at least the filesystem part of the WASI specification implemented (in order to leverage the rest of Inferno).

The actual state as of May 2025 fell a bit short, with only an Hello World demo being available. Nevertheless, this means that the wasm binary format parsing was implemented, as well as all the scaffolding needed for full wasm and WASI compliance.

The implementation is done in Limbo, following skanehira (2025)'s Rust tutorial. The parsing is done using Inferno's yacc. WASI functions are implemented as dis modules dynamically loaded at runtime.

The code is available at

```
git clone git@gitlab.com:edouardklein/acheron.git
```

5. Perspectives

Now that the basic implementation is done multiple perspectives open.

5.1. Performance

Obviously, there should be very interesting interactions between WebAssembly and the Dis VM. One could work on a WebAssembly to Dis compiler, then leveraging the ability to go from Dis bytecode to native CPU instructions, gaining significant speed.

5.2. Graphics

WebAssembly and Graphics is still kind of mess right now, but things are coalescing a bit around mapping SDL calls into WebGL calls, or using the brower's JavaScript interop to put images on canvas. We could look into how to make it work by mapping these calls to reads and writes into Inferno's `/dev/draw`.

5.3. Common APIs

Being able to use any language with a WebAssembly toolchain in Inferno would be nice. But what would be an even bigger thing would be to implement the most common APIs used by current WebAssembly applications so that they could run unmodified in Inferno as if it was a browser.

5.4. More out-there ideas

Maslowski (2024) presented some refreshing ideas at the last IWP9, and subsequent discussions with him led to a general feeling that porting WebAssembly to Inferno will probably spawn weird, but fun and interesting things.

References

Eich, Brendan (Apr. 2008). *Popularity*. https://brendaneich.com/2008/04/popularity/. Accessed: 2023-10-05.

Gemini developers (2025). *TLS, client certificates, TOFU, and all that jazz*. Accessed: 2025-02-10. URL: https://geminiprotocol.net/docs/tls-tutorial.gmi.

Maslowski, Daniel (2024). "centre, left and right*, beyond the stereotype". In: *10th International Workshop on Plan 9*. URL: http://10e.iwp9.org/10iwp9proceedings.pdf.

skanehira (2025). *Writing A Wasm Runtime In Rust*. Accessed: 2025-02-10. URL: https://skanehira.github.io/writing-a-wasm-runtime-in-rust/01_intro.html.

Stand With Snowden (2017). *Let's Encrypt and the Snowden Effect*. Accessed: 2025-02-10. URL: https://www.standwithsnowden.com/news/lets-encrypt-and-snowden.html.

Stuart, Brian L (2023). "Plan 9 and Inferno go to school". In: *9th International Workshop on Plan 9*. URL: http://9e.iwp9.org/9iwp9proceedings.pdf.

WASI Subgroup (2025). *WebAssembly System Interface (WASI)*. Accessed: 2025-02-10. URL: https://wasi.dev/.

World Wide Web Consortium (Dec. 2024). *WebAssembly Core Specification, Version 2*. W3C Recommendation. URL: https://www.w3.org/TR/wasm-core-2/.

Rethinking PKI on Plan 9

Ori Bernstein
ori@eigenstate.org

ABSTRACT

Today, Plan 9 implements PKI using thumbprint files. This does not work well in today's world of rapidly expiring certificates and short lived private keys. As a result, we propose a factotum-scented file server, *auth/pki*, which will handle the X509 certificate chain validation that today's internet infrastructure expects.

1. How does Plan 9 Handle X509 Certificate Validation?

Today, Plan 9 punts on the problem of verifying certificates. While we implement X509 certificate parsing, and programs can verify thumbprints for themselves, we do not implement the verification algorithms.

In the past, this approach was more viable than it is today. TLS itself was used far less frequently. Furthermore, when it was used, the certificates were issued for a longer period, and the private keys used in the certificate were also rotated relatively rarely. However, automated tooling around letsencrypt, tools used in practice at larger web services, and the general web ecosystem has moved towards reissuing not just the certificates associated with a private key, but the private key itself.

As a result, it's become less and less viable to verify certificates by manually pinning the hash of their public key: The public key of many certificates in use is unnecessarily rotated on a regular basis, forcing system administrators to update their lists of allowed certs.

Therefore, we need to either stop verifying certificates for systems that we do not administer entirely, or we need to verify them the *proper* way.

To date, we have selected the first option. We do not validate certificates, outside of small numbers of use cases. In those cases, we pin them directly. This WIP explores what changing the situation may look like. There is code available, but it is incomplete. In particular, it's ignorning many of the subtle complexity of name validation, which means that it will be overly strict in some circumstances, and can be fooled by splitting resource names in a particular fashion in others. Do not use in production.

2. What's a PKI?

Before diving into what certificates are, and how they're verified, let's review how the X509 PKI infrastructure is designed to work.

X509 PKI is a system used to manage the distribution and verification of public keys. The goal of PKI is to establish a method to bind an identity to a public key, via a centrally managed trust delegation system. The verification and identity binding is designed to be delegatable, such that not all trust is centralized in one authority. Instead, a cartel of certificate authorities is able to control the issuing of trusted X509 certificates.

When discussing PKI, we need to start with several definitions.

Certificate authorities issue and sign certificates. An example of a certificate authority may be Verisign or Lets Encrypt. **Registration authorities** are responsibile for verifying the identity of entities who wish a certificate to be issued. Often, but not always, they are the same entity as the certificate authority. The issued certificates are registered in a "Central Directory".

On the client side, the clients keep their **roots of trust**
in a **certifcate store**. When validating a certificate, a **certificate**chain is constructed. A certificate chain consists of a sequence of certificates, each signed by the next certificate in the chain. For a valid certificate, all certificates in the chain must be valid, must not be revoked, and the final certificate must be a root of trust.

3. An Introduction to X509 Certificate Validation

Once a certificate has been issued, servers will present them via TLS to clients, and clients may optionally present them to servers. These certificates are presented in order to verify the identity of the server to the client, and if so desired, of the client to the server.

Conceptually, this verification is simple: First, the verifier ensures that the provided certificate is, in fact, issued for the resource being verified. If a user is attempting to access *https://bank.com*, but an an attacker intercepts the TLS handshake, and presents a certificate for *https://stealyourmoney.com*, then it would do no good for the browser to verify that the certificates have been signed.

Next, the certificate chain is built. A certificate, or certificate chain, is presented. The parents of the presented are pulled from the certificate store or chain, until either there are no more parents that can be identified or the certificate pulled is trusted. If the certificates are is a known trusted certificate, or a *root of trust*, then the validation is complete. If it is not, then the next element of the chain -- ie, the signer of the certificate, is located, and appended to the verification chain. This continues until the chain reaches a root of trust. Once the full chain is discovered, the signature of each element in the chain is verifed against the signature of the next element. If all signatures pass, the names are considered valid.

4. Introducing Auth/pki

When Plan 9 security was initially redesigned, factotum was introduced to remove cryptographhic code from individual programs. The goal was to isolate it in an auditable system that would keep secrets away from buggy code.

When revisiting the way that X509 certificates were handled in Plan 9, similar considerations seemed relevant. Initially, the idea was to put certificate validation into factotum directly, as checking whether a certificate is valid looks similar to checking whether a user should be permitted. However, as it became clear how X509 got validated, it became clear that X509 should not be allowed near any security critical components of the system, and the validation was separated into its own system.

Similar to factotum, auth/pki mounts itself as a file system. The file system contains a *ctl* file, which is opened in order to write the *PEM* chain into. Each certificate in the chain is written, from the leaf to the root, and auth/pem verifies it using the expected verification rules. The protocol looks something like:

```
    -> verify host 'hostname'
    -> cert der nbytes
    -> <bytes of cert data>
    -> cert der nbytes
    -> <bytes of cert data>
    -> done
    <- accept
```

5. Changes to the System

While auth/pki has a simple interface, the way we handle TLS throughout the system is not structured in a way that can take advantage of it. In order to take advantage of auth/pki, a number of changes are needed.

The *tlsClient* call currently has a very poor interface, making changes unnecessarily difficult. Specifically, it uses a struct where the TLS interface allocates members, but where the caller of the API is expected to free them. This means that if anything changes about the contents of the tlsConn struct, all callers must be updated to free new fields. While callers could be updated to match additional certificate chain and hostname fields, we chose to go through and fix the API properly, allowing future revisions of TLS handling to be implemented with less pain.

Additionally, in order for TLS to validate a connection, we need to not only show that the TLS connection presents a certificate chain, we also need to ensure that the certificate chain is issuesd for the resource that we are accessing. This means that we need to know the name of the resource, as presented by the certificate chain.

This is made difficult to do in the general case, due to the way that dial strings work, and the indirection that ndb and the connection server provide. In order to solve this, at the minimum we need to provide a way to parse dial strings and extract the host name from them. This seems sufficient for now, but in the longer term we may need help from the connection server to provide the canonical name for a resource.

6. Current Status

Currently, code exists to implement some level of PKI validation. Several of the changes to the system mentioned above have been implemented, and some programs have been patched in a private branch in order to make them work with *auth/pki*.

However, the name validation is currently done very crudely. The pki validation assumes that names are unstructured blobs of data, and that they only match if they are an exact byte for byte match. This is the way that things should work in a reasonable world, but it's not what happens in our world.

As a result, before this can be shipped, the full name validation algorithms from *RFC 6125* and *RFC 9618*. Additionally, it would make sense to add support for offline certificate revocation lists.

7. References

[1] P. Saint-Andre, J. Jodges, *''RFC 6125''*, 2011

[2] D. Benjamin, *''RFC 9618''*, 2024

[2] D. Benjamin, *''RFC 9618''*, 2024

[3] Recommendation X.509, *''Information Technology--Open Systems Interconnection--The Directory: Authentication''* ITU

Plan 9, the Raspberry Pi, and the ENIAC

Brian L. Stuart

ABSTRACT

As part of an ongoing project to virtually recreate the ENIAC, we have developed a detailed and thorough simulator of the machine. To augment the realism of the simulator, we have implemented a couple of modes of physical interaction. For these, we use Plan 9 on the Raspberry Pi. Here, we discuss the details of these implementations and some issues that have arisen in this work.

1. Introduction

In 1955, the ENIAC was shut down for the last time, bringing to an end nearly 10 years of productive life. One really cannot overstate the influence this machine had on the development of computing. Yet today, the machine no longer exists in one piece. Consequently, no one can experience what it was like to use the ENIAC or to test our understanding of it or its programming on the real machine. This leaves simulation as the only way that we have to approximate the ENIAC experience. Here, we discuss one aspect of an ENIAC simulation, taking advantage of Plan 9's facilities for controlling external devices from a Raspberry Pi.

2. ENIAC Operation

While the acronym ENIAC stands for Electronic Numerical Integrator And Computer, it can be controversial whether to classify the ENIAC as a computer in the modern sense. Although it is not our purpose here to attempt to answer that question, we do point out that throughout its life, the ENIAC was a Turing-complete machine from its initial construction and remained so as various modifications and enhancements were made.

Where the machine differs from our modern conception of a computer is that it originally did not involve a stored program. Instead the selection of operations and their sequencing was specified in terms of cable connections and switch settings. Suppose the next step of the algorithm is to add a number stored in one accumulator to the current value in another accumulator. This operation would be initiated by both accumulators receiving a pulse that triggers one accumulator to transmit its contents and the other to receive it. Upon completion of the addition, one of the accumulators would be configured to output a pulse that would then be carried to the units performing the next step of the algorithm[3].

The transmission of data is also done by sending pulses. The data trunks (also called digit trays) contain 11 cables, one for each digit of a 10-digit number and one for the sign of the number. A digit's value is transmitted over its cable as a serial pulse stream. If a digit has the value seven, then seven pulses are transmitted when transmitting the number additively. When transmitting subtractively, the 9's complement of the digit is transmitted, two pulses in this example.

Each digit of an accumulator consists of a ten-stage ring counter. The active stage of the counter advances one position for each pulse received. When the counter wraps around from the nine position to the zero position, a carry flip-flop is set and later causes a pulse to be routed to the next more significant digit counter. The ten flip-flops in a ring counter each have a neon indicator lamp that shows the flip-flop state. The visual effect is that a digit is represented by a column of ten positions, one of which is illuminated according to the value of the digit.

Later in its life, this form of direct programming was used to implement an instruction set interpreter (a CPU) where the numeric instructions were read from the function tables. This functionality was implemented in 1948, and most of the problems that were run on the ENIAC were done using this form of programming[1, 2, 4].

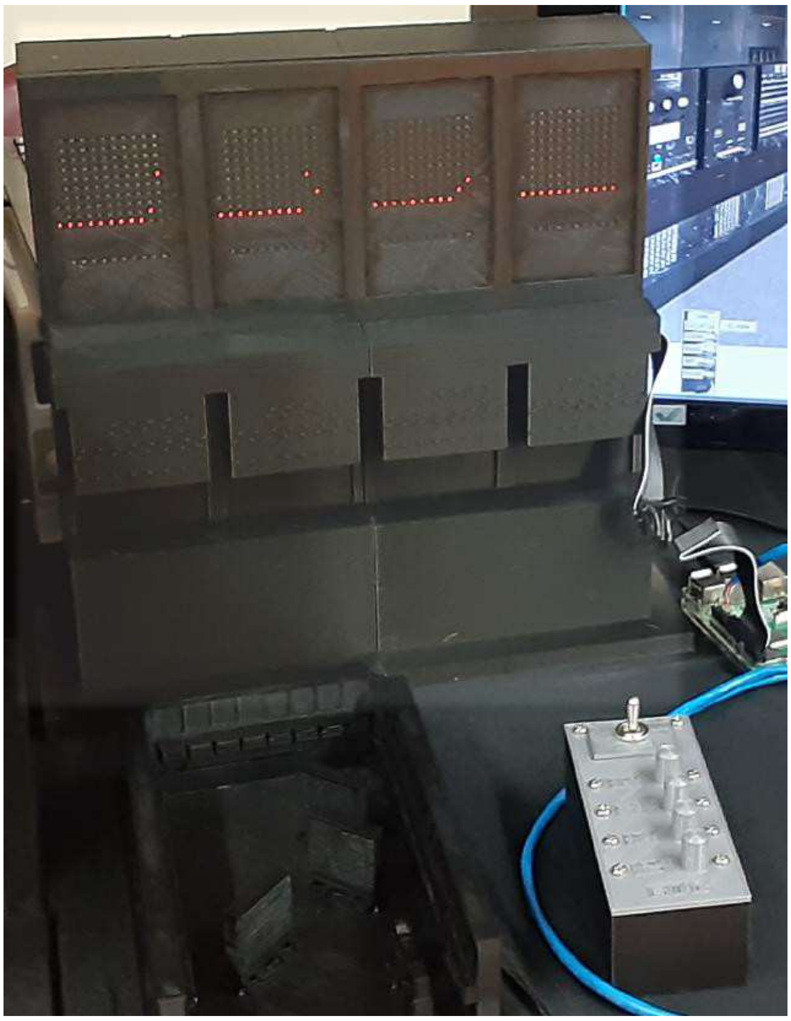


Figure 1: LED Display and Control Box

3. Simulator

The core of the simulator itself is written in the Go programming language, and can thus be easily run on a wide variety of platforms. It operates at a relatively low level of detail, simulating each of the pulses transmitted through the machine during its operation. The wiring of the machine is represented in the simulator by channels, and pulses are represented as messages sent on those channels. Within each accumulator, the simulator represents a digit ring counter by a single integer identifying the currently active stage of the counter. Similarly, there are flags the represent the states of the number's sign flip-flops and of each of the carry flip-flops[6].

In addition to a textual interface, the simulator supports several forms of visualization, with more under development. Each provides a distinct perspective on the machine and collectively they provide as complete a picture of the experience of using it as is feasible without reassembling the original or constructing a full replica. The form of visualization discussed here are physical models of portions of the ENIAC. One element is a full-scale reproduction of the portable control station which allowed programmers to perform some control of the machine from anywhere in the room. The other component consists of a 1/8 scale 3D printed model of four accumulators with LEDs substituting for the neon lamps contained in the original. The simulator command set allows the user to select any four of the accumulators to be shown on the LEDs of the model. Figure 1 shows the two models in operation.

4. Portable Control Station

To assist in checking and debugging, the ENIAC supports a small control box that can be carried anywhere in the room and which duplicates several of the controls found on the initiating and cycling units. The control station includes a three-position toggle switch selecting among continuous clocking, single operation, and single pulse modes. Additionally there are four buttons mirroring the clear, initiate, read, and pulse buttons on the control panels.

We have created a 3D printed replica of the control station. For the four buttons, we use typical momentary contact switches that ground the GPIO line when pushed, and those GIPO lines are configured with pullups. The Clear button is connected to GPIO 6, Initiate to 21, Read to 6, and Pulse to 20.

The four-terminal switch we use for the clocking mode is a little bit of an unusual device. As oriented here, in the left-hand position, the common terminal is connected to both the left and center contactor terminals. In the center and right-hand positions, the common terminal is connected to the center and right terminals, respectively. We connect the common terminal to ground, the left terminal to GPIO 13, the center terminal to GPIO 19, and the right terminal to 26. As with the buttons, all three of these GPIO lines are configured with pullups. When set to the continuous clocking position, both GPIOs 13 and 19 are pulled low. When in the 1 pulse position, only GPIO 19 is pulled low, and when in the 1 add position, only GPIO 26 is pulled low.

The GPIO lines are read every 10ms. For debouncing, actions are only taken when four consecutive reads of the same state are received. The Go code for processing the control box in Plan 9 is shown in Figure 2.

When a change in switch setting is detected and confirmed, the control operation is triggered by a call to `proccmd` with a string argument. The strings in these calls are exactly the same as those that can be entered by the user in the textual interface and that can be included in configuration files. In general, the approach taken in this simulator is for all forms of user interface to mimic the actions of the user of the textual interface using a common language.

5. LED Matrix

Each accumulator has 112 neon lamps used to represent the digits, carries, and sign. One of the most effective ways to control a large number of LEDs is that found in commonly available LED matrix panels. Therefore, rather than wiring 448 individual LEDs for the model, we use an off-the-shelf 64×16 matrix. It is true that over half of the 1024 LEDs in the matrix go unused, but this is balanced against the additional cost and time required to create a custom design that has only the necessary number of LEDs.

Commonly available matrix assemblies use interfaces such as HUB08, HUB12, or HUB75. The device discussed here uses the HUB08 interface. Attempts to locate a datasheet-like specification of the operation of the HUB08 interface were not successful. Searches for the interface generally result in how-to tutorials consisting of instructions on which wires to connect and showing code fragments for an Arduino. We do not find this approach to engineering satisfactory. Information on such sites to describe the functions of pins on the connector do, however, provide a good starting point for reverse engineering the interface.

The interface to this board includes connections labeled LA, LB, LC, LD, R1, R2, G1, G2, EN, LAT, and CLK. The signals LA through LD form a 4-bit row address with LA being the LSB and LD the MSB. LA, LB, and LC are fed to the address lines of two 74HC138s, one for the top half of the array, and one for the bottom half. The LD line is connected to an active high enable on one of the '138s and to an active low enable on the other. The EN signal is routed to an active low enable on both '138s, from which we infer that EN is active low in the HUB08 interface. The net effect is that when EN is low, one side is activated on the 64 LEDs that made up the row selected by LA–LD. Thus only one row can be illuminated at a time, meaning that a driver must continually multiplex the display.

Data is shifted into a row through the R1 line. (Lines R2, G1, and G2 are used for larger arrays and ones that support both red and green LEDs.) It is routed to the SER input of the first of a chain of eight 74HC595 shift registers. The CLK line drives the SRCLK pin of all of the '595s. We know that the HUB08 shifts data on the rising edge, since that's the behavior of the '595 SRCK signal. The RCLK signal on the '595s comes from the LAT input on the HUB08 interface and latches the shifting

```go
for {
    time.Sleep(10 * time.Millisecond)
    newstate := 0
    n, err := fdg.ReadAt(buf, 0)
    if n != 16 {
        fmt.Print(err)
    }
    s := string(buf)
    x, _ := strconv.ParseUint(s, 16, 64)
    if x & (1 << 6) == 0 {
        newstate |= 0x01;
    }
    if x & (1 << 5) == 0 {
        newstate |= 0x02;
    }
    if x & (1 << 21) == 0 {
        newstate |= 0x04;
    }
    if x & (1 << 20) == 0 {
        newstate |= 0x08;
    }
    if x & (1 << 26) == 0 {
        newstate |= 0x10;
    }
    if x & (1 << 19) == 0 {
        newstate |= 0x20;
    }
    if x & (1 << 13) == 0 {
        newstate |= 0x40;
    }
    if newstate != filterset || newstate&0x70 == 0 {
        filtercnt = 0
        filterset = newstate
    } else {
        filtercnt++
    }
    if filtercnt == 4 {
        if newstate != curstate {
            diff := newstate ^ curstate
            if diff&0x70 != 0 {
                switch newstate & 0x70 {
                case 0x10:
                    proccmd("s cy.op 1a")
                case 0x20:
                    proccmd("s cy.op 1p")
                case 0x60:
                    proccmd("s cy.op co")
                }
            }
            if diff&0x01 != 0 && newstate&0x01 != 0 {
                proccmd("b c")
            }
            if diff&0x02 != 0 && newstate&0x02 != 0 {
                proccmd("b r")
            }
            if diff&0x04 != 0 && newstate&0x04 != 0 {
                proccmd("b i")
            }
            if diff&0x08 != 0 && newstate&0x08 != 0 {
                proccmd("b p")
            }
            curstate = newstate
        }
    }
}
```

Figure 2: Go Code for Reading Control Box Inputs

flip-flops into the output register. These are also rising-edge triggered. Experimentally, we find that the shift register output is inverted with respect to the state of the LED illumination. This inversion is handled in how we write to the "pixel" array that represents the current state of the matrix.

6. Controlling the LED Matrix

We only need a single data connection to feed the bits into the shift register since we are using a single color for all LEDs. That, along with the rising-edge clock signal, make it suitable to use the SPI inferface on the Raspberry Pi to drive the data to the matrix. In particular, we use the default SPI mode 0 to drive the shift registers. For this application, the Pi's SCLK line is connected to the display's CLK input, and the MOSI line is connected to R1. For the other controls, we use GPIO lines as follows: 17 to LA, 27 to LB, 22 to LC, 18 to LD, 24 to LAT, and 25 to EN.

The basic sequence for each row to display is as follows:

1. Transmit 64 bits of row data via SPI.
2. Deassert EN.
3. Assert LAT.
4. Set up row number on LA–LD.
5. Assert EN.
6. Deassert LAT.

At first, it was thought that a 33ms period (30 refreshes per second) would be sufficient. After all, NTSC television refreshed the full frame 30 times per second. (Each interlaced half-frame of alternate lines was transmitted in 1/60s.) That rate, however, results in noticable strobing, and we need 60 refreshes per second to ensure a stable display. We suspect the reason for the difference is that unlike CRT phosphors, red LEDs do not have any persistence after their period of excitation.

7. Real Time Considerations

Several variations of this basic approach have been tried with an eye toward creating the most stable display. The first design choice is whether periodically all 16 rows are updated together, or on a faster update period, one row at a time is updated. The advantage for the first option is that there is only one instance of the overhead to activate updates per "frame." However, ensuring that the last row is illuminated for the same amount of time as the others is tricky at best. Updating one row at a time on a faster period has the advantage that the LEDs are left on for longer, resulting in a brighter display. These options did not have a significant effect on the amount of flicker observed in the display. Early implementations use the per-frame period, but more recent experiments have been built around the per-line period. In the remainder of this discussion, all implementations operate at a rate of 1ms per row, resulting in an update rate of 60 refreshes per second.

7.1. Time Shared User Space Implementation

The first approach tried is coded in a C program that is run as a child process of the simulator proper. This LED matrix program splits itself into two processes with an rfork. One process, the parent, receives messages from the simulator containing updates of the states of the neon indicators. These are used to update a data structure shared betweent the two processes. The child process contains the main update loop. The core loop code looks like that shown in Figure 3.

When the system is effectively unloaded, this approach works fine. However, when the ENIAC simulator is running including all of the pulses from the cycling unit, the scheduling of the display updates becomes somewhat sporadic. The effect is significant flickering of the LEDs which is visually quite unfaithful to the original neon lamps.

```
while(1) {
    pwrite(spidat, buf[row], 8, 0);
    fprint(fdg, "set 25 1\n");
    fprint(fdg, "set 24 1\n");
    fprint(fdg, "set 17 %d", row & 01);
    fprint(fdg, "set 27 %d", (row >> 1) & 01);
    fprint(fdg, "set 22 %d", (row >> 2) & 01);
    fprint(fdg, "set 18 %d", (row >> 3) & 01);
    fprint(fdg, "set 25 0\n");
    fprint(fdg, "set 24 0\n");
    row = (row + 1) & 0xf;
    sleep(1);
}
```

Figure 3: Main Loop for User-Space LED Matrix Update

```
pid = rfork(RFPROC|RFMEM);
if(pid == 0) {
    drive(nil);
    exits(nil);
}
path = smprint("/proc/%d/ctl", pid);
cfd = open(path, OWRONLY);
free(path);
fprint(cfd, "period 1ms\n");
fprint(cfd, "cost 1ms\n");
fprint(cfd, "admit\n");
```

Figure 4: Set Up Real-Time Scheduling

7.2. Real Time User Space Implementation

In an attempt to improve the steadiness of the display, the second approach uses the EDF real-time scheduler in Plan 9. After the child process is created, the parent configures it as a real-time process as shown in Figure 4 (less error handling).

In the child's loop, the only change is to replace the `sleep(1)` with `sleep(0)` to yield the CPU until the next period.

Using this approach does improve the flicker, but it does not completely eliminate it. Again, with no load on the system, the display is steady, but as soon as the display process contends for the CPU, the scheduling is not longer fully reliable.

7.3. Timer-Based Kernel Implementation

The most reliable and precise way of controlling tight timing like this is a driver in the kernel connected to timer interrupts. This leads to the third approach to implementing the display. To that end, we have a very simple kernel device in `devled.c`. It serves a single file called `ledmat` that accepts only four messages. The message `start` uses `addclock0link` to fire a routine every milli-second for updating a line of the display, and the `stop` message deletes the timer. The other two message are the same as the input taken by the user-space application for clearing the display and setting the value of an LED. The per-ms routine is mostly a translation of the user space code as shown in Figure 5.

One surprising aspect of this is the polled communication over the SPI port, rather than calling the `spirw` routine in `spi.c`. In fact, without thinking ahead to the implications, the first attempt was written in exactly that way. However, the implementation in `spi.c` uses the DMA controller and thus blocks the calling process until the DMA transfer is complete. This cannot be done in code that is running as part of an interrupt handler, which routines registered with `addclock0link` do. The

```
spi->cs = Rxclear | Txclear | Adcs | Ta;
i = 0;
while(1) {
    spi->data = ledbuf[row][i];
    if(i >= 7)
        break;
    while((spi->cs & Txd) == 0) ;
    i++;
}
while((spi->cs & Done) == 0) ;
spi->cs = 0;
gpioout(25, 1);
gpioout(24, 1);
gpioout(17, row & 01);
gpioout(27, (row >> 1) & 01);
gpioout(22, (row >> 2) & 01);
gpioout(18, (row >> 3) & 01);
gpioout(25, 0);
gpioout(24, 0);
row = (row + 1) & 0xf;
```

Figure 5: In-Kernel LED Matrix Update

solution, then, is to use the SPI controller in polled mode.

As one would expect, the kernel driver approach does produce the most stable and flicker-free display, even with the rest of the system heavily loaded. Initially, there was some concern that this approach would put a significant load on one of the cores and slow the system down. However, observations via `/dev/sysstat` show the additional interrupt handling load to be at about 1%.

8. Future Work

Although the in-kernel solution is entirely sufficient for this specific application, it does raise some questions regarding the real-time scheduler. In particular, we would like to investigate the degree to which it is possible to adjust the scheduler so that milli-second scale processes, such as we have here, are not adversely affected by the normal time-sharing load on the system. We believe that such an enhancement could improve the value and utility of the real-time scheduler. Aside from this rather interesting diversion, continuing work on the ENIAC simulator is focused on additional visualization methods and implementing the enhancements made to the machine during its time at the Ballistics Research Laboratory.

9. Conclusion

As icing on the cake for this connection between Plan 9 and the ENIAC, there is another connection worth relating. One of the mathematician/engineers who worked with the ENIAC project was Harry Huskey[5]. Later Huskey went on to join the Computer Science department at the University of California at Berkeley. One of the students for whom Huskey was a master's degree advisor was Ken Thompson. Thompson's contributions to both Plan 9 and the Go language bring an indirect ENIAC connection to the tools used in the project. Furthermore, the CAD package used to create the 3D model of the ENIAC was developed at the same Ballistics Research Laboratory where the machine spent most of its life, furthering the connection between the reconstruction and its anticedent.

The basic functionality of this work was straightforward in Plan 9, but would have also been pretty straightforward in other environments. As often happens, when writing up this work, details arose that had previously only been minor annoyances, but that would have been gaping holes in a complete report on the work. In particular, the investigation of controlling the timing of updates quickly became a significant focus. It was here that Plan 9 really shone. No other environment would have provided the same ease of implementation that would allow for such a quick comparison of different techniques.

References

[1] R.F. Clippinger. A logical coding system applied to the ENIAC. Technical report, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, September 1948.

[2] W. Barkley Fritz. Description of the ENIAC converter code. Technical report, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, December 1951.

[3] Adele K. Goldstine. Report on the ENIAC: Part I technical description of the ENIAC. Technical report, Moore School, University of Pennsylvania, 1946.

[4] Thomas Haigh, Mark Priestley, and Crispin Rope. *ENIAC in Action: Making and Remaking the Modern Computer*. The MIT Press, 2016.

[5] Harry D. Huskey. Report on the ENIAC: Part II technical description of the ENIAC. Technical report, Moore School, University of Pennsylvania, 1946.

[6] Brian L. Stuart. Simulating the ENIAC. *Proceedings of the IEEE*, 106(4):761–772, April 2018.

What I Saw at the Evolution of Plan 9

Geoff Collyer
geoff@collyer.net

ABSTRACT

This is a sort of oral history of Plan 9 and related work at Bell Labs and my home. It is necessarily a subjective view; others may disagree. It is subject to my recall of events; dates may be off.

Dramatis Personae

These are the main people mentioned (perhaps implicitly) in this paper. Their roles were not as limited as they appear here. Those mentioned above the dividing line worked at one time at the Bell Labs Computing Science Research Center (at times Center 1127) or its successors.

login	name	roles (abbreviated)
ken	Ken Thompson	Unix, Plan 9 & C creator, compilers, filesystems
dmr	Dennis Ritchie	Unix & C creator
rob	Rob Pike	Unix developer, Blit, Plan 9 & Inferno creator
rsc	Russ Cox	rob's summer intern, Plan 9 developer
philw	Phil Winterbottom	Plan 9, Brazil, Inferno, Alef developer
seanq	Sean Quinlan	cached-worm inventor, venti creator
sean	Sean Dorward	Limbo compilers, venti creator
presotto	Dave Presotto	*upas* creator, Plan 9 networking & compilers
brucee	Bruce Ellis	Plan 9 & Inferno fan, compilers
jmk	Jim McKie	Plan 9 developer & maintainer
geoff	Geoff Collyer	Plan 9 user & maintainer; researcher, support
forsyth	Charles Forsyth	Plan 9 fan, compilers, Inferno maintainer
miller	Richard Miller	Plan 9 fan, compilers, Raspberry Pi port

Prehistory

Unix energised me rather intensely, starting in 1976 with Sixth Edition: [Tho1975] a powerful, coherent system with

- simple, regular interfaces,
- device access via special files,
- novel pipes,
- a hierarchical file system with simple, regular names, [Pik1985]
- relatively free of arbitrary limits and restrictions,
- written in a high-level system implementation language,
- with source and documentation on-line,
- with sophisticated tools like *yacc*, [Joh1975]
- and able to run on modest and popular hardware.

I was an undergraduate at the University of Western Ontario in London, Ontario, Canada, and my point of contact was Eric Gisin at the University of Waterloo, who also introduced me to *Software Tools*. [Ker1976] I persuaded the UWO Computer Science department to obtain its first Unix licence, for a DEC PDP-11/34 [Cor1973]. Charles Forsyth, then one of the informal student administrators of Waterloo's PDP-11/45 Math/Unix, was a great help in upgrading our Sixth Edition system.

Contemporary interactive systems (e.g., DEC's TOPS-10, [Cor1977] BBN's TENEX, [Bob1972] Honeywell's GCOS) [Hon1980] provided less (and less interesting and elegant) service on considerably larger machines; Multics [Cor1965] required *much* larger machines, but was influential: ideas and some people moved from Multics to Unix.

I followed the progress of (mainly DEC VAX) Research Unix through Tenth Edition [Com1990] and managed to run Sixth through Ninth Editions [Lab1986]. By contrast, the ongoing mauling of Unix by the Unix vendors in the 1980s was demoralising. Features were more important to them than general mechanisms [For1990]. POSIX then standardised even the mistakes [Eng1990].

I first heard of what became Plan 9 when interviewing with Rob Pike in 1986. He was recruiting others to assist with developing software for the *Gnot* terminal, [Loc1987] a successor to the Jerq, Blit, DMD 5620, and Teletype 630 and 730 bit-mapped displays but with a network interface instead of a serial port, and a 68020 CPU, and later an AT&T CRISP CPU, and a virtual-memory operating system this time [Dit1987]. Rob gave a talk on the Gnot software in 1987 at the University of Toronto, where I was on staff, and which I found encouraging and inspiring in the period that System V Release 4, the Grand Unified Unix, was being created by smashing System V into 4BSD at high speed.

First Contact

I found the 1990 UKUUG Plan 9 papers [Pik1990] as electrifying as the 1974 CACM Unix paper [Rit1974]. Plan 9 seemed to be a continued evolution of Research Unix but rethought for modern hardware, with the benefit of 20 years' experience, and different where it made sense (e.g., per-process file namespaces, no filesystem links, automatic unattended permanent on-line backups). Notably, it was designed for distribution, with specialised systems connected over a variety of networks, including Ethernet and Datakit [Che1980]. The initial terminals were Gnots.

I got the single-floppy bootable PC distribution at the 1993 Usenix conference in San Diego, and later the 3-floppy distribution. These provided a tantalising taste of the system, overcoming the ugliness of PC hardware.

Arrival at Murray Hill

In July 1994, I arrived at Murray Hill, NJ to work for AT&T under Ted Kowalski creating the operating system for a TV set-top box called Homecenter, borrowing bits from Plan 9 [Col1996, AMD1994]. The Second Edition of Plan 9 was issued in 1995 for a nominal fee but not as freely-available source, and supported systems such as the IBM PC, MIPS and SGI systems, and the Nextstation, based on the Motorola 68020. A few components of the system, such as user-mode IP processing, the *8½* window system, and *help* (now *acme*) were written in the *Alef* language, [Win1995] created by Phil Winterbottom, which focussed on concurrent programs and implemented CSP channels [Hoa1978].

In 1995, AT&T split, Homecenter was cancelled, I moved to Unix support in Research in the newly-created Lucent Technologies, and most of center 1127 (Computing Science Research) moved to Lucent.

Multiprocessor PCs and Other Systems

IBM PCs were initially resolutely uniprocessor systems. Companies such as Compaq began making multiprocessor PC systems out of Intel x86 processors starting with the 386, but the Pentium Pro was the first x86 processor really designed to be built into multiprocessors, particularly with more than two processors. The memory ordering model from the 486 on has been the same, but starting with the Pentium Pro, CPU pipelines and store buffers became long enough to cause Plan 9 *sleep/wakeup* problems due to queued memory writes [Cox].

Intel guarantees cache coherence [Pat2009] (though not DMA coherence) on x86 CPUs; the problem is worse on CPUs that require manual cache flushing (e.g., ARM, PowerPC). Decent RISC-V systems provide cache and DMA coherence, which is very helpful.

Brazil

A development branch, Brazil, was already underway, driven by Phil Winterbottom. The emphasis was on greater efficiency (e.g., streamlined queues instead of Multiprocessor Streams™), but also included a few experiments, such as raw uncompressed raster graphics (`/dev/graphics`) instead of Bitblt, ultimately replaced by *draw*(3). `/dev/graphics` assumed a 300 Mb/s network that did not materialise. Nevertheless, the first version of *rio*, distinct from *8½*, was written for raw raster graphics. Sean Quinlan created a *rio* variant called *brio* that implemented something closer to Bitblt or *draw*(3) on top of raster graphics, for better performance (e.g., when rubber-banding a new window). IP processing was moved into the kernel, partly with an eye to new 100 Mb/s Ethernet. Brazil eventually become main-line Plan 9 before Third Edition.

Inferno

Inferno was a response to the needs of set-top box developers, a Plan 9 kernel with user-mode processes replaced by an interpreter for Limbo byte codes [Dor1997a]. Inferno itself could run on bare hardware or hosted on another operating system. Limbo in turn was influenced by CSP and Alef, intended for concurrent programming [Tec1997, Dor1997b]. The *draw*(3) interface first appeared here.

In 1996, I moved to a Research department building a multi-media message system, named Mesa, and incorporating fax, mail, and voice input and output. Today Apple's Siri is similar, but our system deliberately lacked any so-called 'artificial intelligence'. The system was built on Inferno, hosted on Plan 9 quad-core Xeon PCs (for ECC memory). I designed and implemented an intended replacement for SMTP, RSMTP [Col2001]. We saw a much bigger improvement in Ethernet performance from changing to switched Ethernet than from moving from 10 to 100 Mb/s Ethernet.

Phil Winterbottom lead the *Pathstar* project, a combined telephony and IP router, positioned as a possible 5ESS phone switch replacement. It was built on a Plan 9 kernel, but running Inferno. In 1999, my department moved to Pathstar development, in the hope that a Siri-like replacement for dial tone would be appealing. In 2000, Pathstar was cancelled; Phil and then Ken Thompson left Lucent, which was in decline, and then the 2001 'dot-com' crash happened.

Lucent's Inferno Business Unit could not figure out what to do with Inferno, after making a hash of its documentation. So Lucent sold it to Vitanuova, who made a serious attempt to sell it, including converting Framemaker documentation back to manual pages.

New 9P and File Servers ...

Alef was replaced by *libthread* in C. Third Edition was issued in 2000 under an open-source licence with *draw*(3) graphics.

A new version of 9P, the file server protocol, named *9P2000* was created and Ken's file server kernel was adapted to speak it. A few system calls were changed in conjunction with *9P2000*. A new authentication mechanism was included in *9P2000*. notably supported by *factotum*(4) and *secstore*(8). Sean Quinlan and Sean Dorward created *venti*, a hash-addressed block store with deduplication.

... and Return to California

In 2001, Lucent offered to buy out senior researchers; Rob Pike and others left for Google. I left for a start-up in California. Fourth Edition was issued in 2002.

Sean Quinlan, Jim McKie, and Russ Cox created a new, user-mode file server, named *fossil*, that could use *venti* as its block store.

I wrote early gigabit Ethernet drivers and bought an optical-disc jukebox. I was not ready to rely on *venti* nor *fossil* yet, but wanted to be able to handle archives larger than 2 GB, so modified Ken's file server kernel to implement 64-bit file and device sizes [Tho2005]. Bruce Ellis had recently improved code generation for the 386 to in-line most `vlong` operations, which made their use cheaper. 9P had always used 64-bit file sizes, but the original file server only implemented signed 32-bit sizes.

I implemented optional greylisting [Har2003] in *smtpd*, which greatly reduced the volume of mail spam. The arms race continues, however.

In 2004, Dave Presotto left for Google.

DEC Alpha PC

I bought a used DEC Alpha PC and ran Plan 9 on it. This port was a 32-bit system despite the Alpha being a 64-bit processor. David Hogan (`dhog`) ported Plan 9 while a student at the University of Sydney, before joining Lucent's Inferno Business Unit. The Alpha was a fast CPU for its day, though power-hungry.

The Alpha was, in some respects, a second system relative to the VAX. It had a kind of loadable microcode, called PALcode, that was OS-specific: there were versions for VMS, NT and Unix, implementing different instructions. So there wasn't a single porting target. There were other variations by model, notably how to access the console serial port, which you really do not want when porting an operating system to it.

Return to Murray Hill

In 2006, Alcatel bought Lucent and became Alcatel-Lucent. Jim McKie recruited me to return to Research, to the remains of center 1127. Russ Cox had been holding the fort, but was looking to spend more time in academia. With Russ's help, I got the latest *venti*, using *libthread*, to work and to survive restarts. Previously, shutting down and restarting this version of *venti* had destroyed its block store.

I then copied the contents of the optical-jukebox file servers into a *venti* store on a RAID 5 volume on magnetic disks. The history goes back to the end of February 1990. Making it generally available would be difficult since it contains home directories and private files, which would be exposed. At Russ's suggestion, I converted the file server kernel into a user-mode file server, *cwfs*(4), and then retired the file server kernel, and the IL protocol along with it, as the file server kernel was the last remaining software that required IL. NAT devices tended to mangle IL headers anyway.

Ports

Jim McKie had long maintained the Plan 9 PC support (other than the compiler suite). He had already created the 64-bit *9k* kernel for AMD64 systems for a U. S. Department of Energy High Performance Computing contract under the project name 'HARE'* with participants Bell Labs, IBM Research, Sandia National Labs and Vitanuova [Col2010a]. Charles Forsyth produced the C compiler suite for it. Ultimately, the project instead used IBM Blue Gene PowerPC supercomputers at various U. S. government laboratories. *9k* initially only ran on AMD64, not Intel64, systems, and was intended as a CPU server for use with *drawterm* or other terminal. It is highly compatible at user source level with the old Plan 9 kernel, implementing the same system calls and many of the same devices, while dropping support for some old or annoying hardware.

We needed 10 Gb/s Ethernet interfaces on any machine that we sent to a national lab, and thus needed a 10 Gb/s switch to verify that our Ethernet drivers worked with it. This was probably around 2010 and 10 Gb/s equipment was scarce and expensive. Alcatel-Lucent made such switches and so we tried to buy one through internal purchasing channels. Nothing happened. We pushed and pushed and still nothing happened. Eventually I phoned Arista Networks, told the salesman who I was, that I was with Bell Labs of Alcatel-Lucent, and that I was having no luck buying internally. He said he would call me back after consulting others, and he did. He said approximately 'Some of the guys are afraid that you're going to take our switch apart and reverse-engineer it'. I promised that we just needed a working switch, could not buy one internally, and that we absolutely **would not** reverse-engineer their switch. And that is how, after much pleading, we got our 10 Gb/s switch.

I ported Plan 9 to machines of several 32-bit architectures: PowerPC (Xilinx Virtex 4 and 5 FPGAs), Arm (Trimslice and others), MIPS (Routerboard) [Col2010b]. The Virtex ports were in support of testing an encrypted memory system. For the others, we were looking for a small, portable terminal that could attach to Ethernet, mouse, keyboard, and monitor as needed. (Ultimately, the Raspberry Pi 4 and later probably come closest.)

Building a kernel in parallel with *mk* on a 24-core `386` Xeon system used so much system time (probably on lock contention) that by default it was slower than with `NPROC=4`. Since then, use of semaphores for locking and `monitor`/`mwait` instructions to idle, instead of looping, should have fixed that. Our quad-core Xeons built a kernel from scratch in 2 seconds elapsed time.

The PC bootstrap programs *9load* and *9pxeload* had various problems, including becoming too large to fit in 640KB due to additional Ethernet drivers, and the need for modified device drivers. I replaced them with the new *9load* and *9boot*(8), using a variant PC kernel and the normal PC kernel's drivers instead of modified drivers [Col2013]. The new bootstraps are compressed with *lzip*, are self-decompressing, and can load `386` and `amd64` kernels. *9load* can load *gzip*ped kernels.

Better IPv6

I worked toward the goal of being able to run a Plan 9 site using only IPv6 [Dee1998, Dee2017]. I had this working at home in California and it should be possible now, even with SLAAC address assignment, except for PXE booting from BIOS. I updated IPv6 support to match newer RFCs and modified *dial*(2) to try to connect to all IP addresses for a name at once, since it is common for a system to have multiple IPv6 addresses, plus any IPv4 addresses, but they may not all be reachable and up at a given time. IPv6 uses multicast for its equivalent of ARP, so some Ethernet drivers had

* Portions of this project supported by the Department of Energy under Award Number DE-FG02-08ER25847.

to be fixed to implement (mainly accept) multicast.

In Memoriam

In 2011, Dennis Ritchie died. Aside from all his work creating C and Unix, Dennis was a major supporter and booster of Plan 9. His good taste and judgement kept C small and simple, at least through 1990 ISO C (thereafter, obsessive-compulsives began adding their favourite, if largely pointless, hobby-horses).

Return to California II

In 2014, I moved to Google in Mountain View, CA. Alcatel-Lucent was still in decline and would soon be bought by Nokia. I bought 2 Lenovo ThinkServer TS140 Xeon servers, which took a while to tame. I never got the integrated Intel i217 Ethernet controller to function, but was able to add PCI-E Ethernet cards. I wrote an NVME driver for Plan 9. I retired at the end of 2018.

I wanted to further explore 10Gb/s Ethernet via RJ45, but Intel's X540 cards run hot, and require cooling by the TS140s' fans, which are very loud. In 2019, I bought quiet Puget Systems Xeon and PC Engines APU2 AMD64 servers with ECC memory, and adapted *9k* for them.

9k for AMD K10

9k as I had copied from the Labs was not really ready for serious use on current hardware. It got odd traps, had old drivers, and could only use about 600MB of RAM, despite being nominally a 64-bit system. I fixed the traps, updated the drivers, made it run on Intel64 systems, and adapted the system to use many-gigabyte memories. I widened *malloc* and *memset* size arguments to `uintptr`, the unsigned integer type as wide as a pointer, and adjusted *fossil* and *venti* to exploit more than 4 GB of memory. `9k10cpu` has run on systems with 4, 16, 32 and 64 GB of RAM; `9k10fs` on 32 GB. Once the kernel was stable, some programs still got odd faults due to now-too-small thread stacks. I fixed the relevant libraries to use bigger thread stacks or allocate local data on the heap. A recent fix in March 2025 from Russ Cox to correct `noted(NCONT)` handling has made *even gs* (ghostscript) work.

By using *lzip* to fit in the bottom 640KB and an uncompressing executable header inspired by Russ Cox, most PC kernels are now small enough to load by PXE directly, without *9boot*. PC hardware is sufficiently discoverable that `plan9.ini` can usually be dispensed with (`/cfg/pxe` files can still be read once the kernel is running).

Future Times

The year 2038 is not far off and signed 32-bit times will go negative in January of that year. I converted the system to consistently use `ulong`s to hold times instead of `long`s and adapted the `libc` functions and the few programs that compare times (e.g., *mk*, the *qer*(8) programs) to cast them to `vlong` first. Tests for failure now compare exactly with `−1UL`, not just any negative time. This extends times to the year 2106 without needing to use 64-bit times, and thus change disk formats, notably *venti* and *fossil*. I extended the `timezone` files for North America to 2106.

Escape from APE

I have tried to reduce reliance on APE, the ANSI/POSIX Emulation environment. My *awk* invokes *rc*, not *ape/sh*, so it is possible to build kernels without a working APE. Most other APE programs, such as *troff* and friends, were almost trivially converted to native programs. So far, only programs that use Unix's *select* require substantial changes. Only the `postscript` programs, *bzip2*, *gs*, *spin*, and *tex* still use APE. Of those, I rarely use *bzip2* and *spin* and never use *tex*.

In Memoriam II

In 2020, Jim McKie died. He had tackled the support in Plan 9 of the most obnoxious, recondite, and brain-damaged aspects of PC hardware, more-or-less cheerfully. Without his efforts and those of Russ Cox, I believe that Plan 9 support at the Labs would have collapsed after Dave Presotto left.

Licensing

In 2020, the Plan 9 Foundation, at `http://plan9foundation.org`, was formed. I was one of the initial directors. In 2021, Nokia, the successor to Alcatel-Lucent, transferred ownership of Plan 9 to the foundation. The historical releases can be found there, re-licensed under the MIT open-source license.

9k for RISC-V

In 2020, I ported *9k* to 64-bit RISC-V systems. (The 32-bit ones seem to be aimed at microcontrollers and other sub-Unix uses.) [Col2023] Richard Miller supplied the C compiler suite [Mil2020]. RISC-V systems are becoming faster and larger, and some resemble x86 PCs, with PCI-E slots. Some even have adequate programming documentation. So far, RISC-V is relatively simple and free of the historical mistakes made in other architectures, though there are vigorous efforts underway to add bad ideas and unnecessary complexity, usually just to win micro-benchmark contests. With luck, I may be able to replace my complicated Puget Xeon systems some day with simpler RISC-V servers.

Before RISC-V, ARM64 seemed like a potential x86 replacement, but a variety of inexpensive systems has not appeared (other than recent Raspberry Pis), and RISC-V is considerably simpler. ARM64 Macs are locked down to ensure only MacOS can run, and the situation is getting worse.

What I Saw Just a Lttle

There were various developments that I was aware of, but was not involved with much, notably AoE (ATA over Ethernet) and *Nix*, a modified *9k*. At Bell Labs, we used AoE to connect our main file server to a Coraid™ RAID device, which held our *venti* arenas and indices, and *fossil* file systems *main* and *other*.

What I Opposed

I think that Ethernet jumbo packets are a bad idea [Ste] and discouraged their use everywhere. Use of jumbo packets on some but not all devices on a network is awkward at the least, there is no agreement on maximum jumbo packet size, jumbos consume more memory than normal packets (obviously), and the IP checksums and Ethernet CRC are less effective as packet size increases. The Ethernet controllers will transmit continuously if there are packets to send, so increasing the maximum RPC size in `devmnt` (which I have done at home) is probably more effective at increasing throughput; I doubled MAXRPC (by negotiation) and saw 30% improvement on large copies.

Developments Elsewhere

Go is an interesting language, [Tho] but I have not spent much time with it to date.

Some people feel that Plan 9 needs a web browser capable of being used for banking and the like. The infeasibility of creating such a browser is discussed here, [DeV2020] but Plan 9 is a distributed system, so running a browser on a Unix machine using some combination of VNC and *u9fs* is feasible.

In related news, C++ continues to grow in size and complexity. It may be the largest programming language ever. IBM's 1965 PL/I specification was 164 pages, and PL/I was considered to be a huge and complex language. ISO's 2017 C++ specification is 1,605 pages. I am certainly never going to understand it.

Acknowledgements

I have been lucky to observe or work with some brilliant people, including those mentioned above, and they were generally helpful and generous. Ken Thompson volunteered to help me set up a Plan 9 optical-jukebox file server.

David du Colombier has incorporated some of my changes into the *9legacy* distribution at `http://9legacy.org`. Adrian Grigore is preparing a bootable installer image, of my system, for CD or USB disk.

Closing Thoughts

Plan 9 has not taken the world by storm, alas. UTF-8 has been a success, despite the silliness of recent Unicode additions (e.g., emojis) [Pik1993]. There have been by-products: `plan9ports`, ports or re-implementations of *sam*, *acme*, and *rc*. But some powerful parts have not been generally understood and adopted: e.g.,

- the *cpu* command and underlying machinery,
- networks in the file namespace,
- TLS as kernel device,
- remote access to devices via remote file system protocol,
- easy and routine cross-compilation,
- starting daemons as `none` and using cryptographic authentication to change user-id.

The evidence suggests that the popularity of an operating system (indeed, of most software) is inversely proportional to its technical excellence. Operating system choices are also somewhat religious: it is very difficult to change someone's choice. Apple managed it with Mac OS X, but there was moaning, wailing and rending of clothing from some old users for quite a while, despite the obvious superiority of OS X.

Perhaps in time, the lessons of Plan 9 will be broadly absorbed, though needless complexity and bad taste [Fou, Tor] will always oppose them, including in hardware design [Int]. For example, the ever-changing administrative interfaces and increasing access restrictions in MacOS. It took time for the lessons of Unix and predecessor systems (e.g., not writing in assembler, process protection via memory management, user-space command interpreters, filters, pipelines, hierarchical file systems) to be absorbed, and then in some cases misunderstood (e.g., `/proc` in Linux). And of course, we must defeat *git*.

References and Further Reading

AMD1994. AMD, *Am29200 and Am29205 RISC Microcontrollers User's Manual*, Advanced Micro Devices, 1994.

Bob1972. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson, ''TENEX, a Paged Time Sharing System for the PDP-10,'' *Comm. Assoc. Comp. Mach.* **15**(3), pp. 135-143 (March 1972).

Che1980. G. L. Chesson, A. G. Fraser, ''Datakit Network Architecture,'' *Compcon*, pp. 59-61 (Spring 1980).

Col1996. Geoff Collyer, ''Setting Interrupt Priorities in Software via Interrupt Queueing,'' *Computing Systems* **9**(2), pp. 119-130 (Spring 1996).

Col2001. Geoff Collyer, *A Really Simple Mail Transfer Protocol*, `http://www.collyer.net/~geoff/rsmtp/rsmtp.pdf`, April 2001.

Col2010a. Geoff Collyer, Charles Forsyth, ‘‘A Cache to Bash for 9P,’’ *Fifth International Workshop on Plan 9*, Seattle (October 2010).

Col2010b. Geoff Collyer, ‘‘Recent Plan 9 Work at Bell Labs,’’ *Fifth International Workshop on Plan 9*, Seattle, invited talk (October 2010).

Col2013. Geoff Collyer, ‘‘Bootstrapping Plan 9 on PCs,’’ in *Plan 9 Programmer’s Manual* (December 2013).

Col2023. Geoff Collyer, ‘‘Plan 9 on 64-bit RISC-V,’’ *Ninth International Workshop on Plan 9*, Waterloo (April 2023).

Com1990. Computing Science Research Center, AT&T Bell Laboratories, Murray Hill, New Jersey, *UNIX Research System Programmer’s Manual, Tenth Edition,* Saunders College Publishing (1990).

Cor1965. F. J. Corbató, V. A. Vyssotsky, ‘‘Introduction and Overview of the Multics System,’’ *Proceedings of the AFIPS Fall Joint Computer Conference*, Las Vegas, pp. 185-196, ACM, ISBN 978-1-4503-7885-7 (1965).

Cor1973. Digital Equipment Corporation, *pdp11/45 processor handbook*, 1973.

Cor1977. Digital Equipment Corporation, *decsystem10 monitor calls*, May 1977.

Cox. Russ Cox, *Spin & Plan 9*, `http://swtch.com/spin`.

Dee1998. Steve E. Deering, Bob Hinden, *RFC 2460: Internet Protocol, Version 6 (IPv6) Specification,* Internet Society (December 1998).

Dee2017. Steve E. Deering, Bob Hinden, *RFC 8200: Internet Protocol, Version 6 (IPv6) Specification,* IETF (July 2017).

DeV2020. Drew DeVault, *The reckless, infinite scope of web browsers*, `https://-drewdevault.com/2020/03/18/Reckless-limitless-scope.html`, 18 March 2020.

Dit1987. D. R. Ditzel, H. R. McLellan, ‘‘The CRISP Microprocessor: Collected Papers,’’ Comp. Sci. Tech. Rep. No. 138, AT&T Bell Labs (February 10, 1987).

Dor1997a. S. M. Dorward, R. Pike, D. M. Ritchie, H. W. Trickey, P. Winterbottom, ‘‘Inferno,’’ *Proc. IEEE Computer Conference (COMPCON)*, San Jose, California (1997).

Dor1997b. S. M. Dorward, R. Pike, P. Winterbottom, ‘‘Programming in Limbo,’’ *Proc. IEEE Computer Conference (COMPCON)*, San Jose, California (1997).

Eng1990. Institute of Electrical and Electronics Engineers, *Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language] (IEEE Std 1003.1-1990) = ISO/IEC 9945-1:1990,* IEEE, New York (1990).

For1990. C. H. Forsyth, ‘‘More Taste: Less Greed? or, Sending UNIX to the Fat Farm,’’ pp. 161-172 in *Proceedings of the Summer 1990 UKUUG Conference*, UKUUG, London (July 9-13, 1990). `http://www.terzarima.net/doc/taste.pdf`

Fou. Free Software Foundation, *GNU software,* anonymous ftp from `prep.ai.mit.edu:/pub/gnu`.

Har2003. Evan Harris, *The Next Step in the Spam Control War: Greylisting*, 2003.

Hoa1978. C. A. R. Hoare, ‘‘Communicating sequential processes,’’ *CACM* **21**(8), pp. 666-677 (1978).

Hon1980. Honeywell, *Level 66/Series 6000 General Comprehensive Operating Supervisor*, August 1980.

Int. Intel, *Intel 64 and IA-32 Architectures Software Developer Manuals*, `https://-www.intel.com/content/www/us/en/developer/articles/technical/-intel-sdm.html`.

Joh1975. S. C. Johnson, ‘‘Yacc — Yet Another Compiler-Compiler,’’ Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

Ker1976. Brian W. Kernighan, P. J. Plauger, *Software Tools,* Addison-Wesley (1976).

Lab1986. AT&T Bell Laboratories, in *UNIX Programmer's Manual, Ninth Edition, Volume One*, ed. M. D. McIlroy, Murray Hill, New Jersey (September 1986).

Loc1987. Bart N. Locanthi, ‘‘This is Gnot Hardware,’’ 11276-870629-04TM, AT&T Bell Laboratories (1987).

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

Pat2009. David Patterson, John Hennessy, *Computer Organization and Design (4th ed.),* Morgan Kaufmann (2009).

Pik1985. Rob Pike, P.J. Weinberger, ‘‘The Hideous Name,’’ pp. 563-568 in *USENIX Conference Proceedings*, USENIX, Portland, OR (Summer 1985).

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

Pik1993. Rob Pike, Ken Thompson, ‘‘Hello World,’’ pp. 43-50 in *USENIX Technical Conference Proceedings*, USENIX, San Diego, CA (Winter 1993).

Rit1974. D. M. Ritchie, K. Thompson, ‘‘The UNIX Time-Sharing System,’’ *CACM* **17**(7), pp. 365-375 (July 1974).

Ste. Richard A. Steenbergen, *The Case Against Jumbo Frames*, `https://-archive.nanog.org/sites/default/files/wednesday_general_steenbergen-_antijumbo.pdf`.

Tec1997. Lucent Technologies, ‘‘The Limbo Language Definition,’’ in *Inferno User's Guide* (1997).

Tho1975. K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories (May 1975). Sixth Edition

Tho2005. Ken Thompson, Geoff Collyer, ‘‘The 64-bit Standalone Plan 9 File Server,’’ in *Plan 9 Programmer's Manual* (2005).

Tho. Ken Thompson, Rob Pike, Russ Cox et al., *The Go Programming Language*, `http://go.dev`.

Tor. Linus Torvalds, *Linux*, `http://www.linux.org`.

Win1995. Phil Winterbottom, ‘‘Alef Language Reference Manual,’’ in *Plan 9 Programmer's Manual, 2nd ed.* (March, 1995).

Lola: A new Window System for Plan 9

Angelo Papenhoff
aap@papnet.eu

ABSTRACT

Lola is a new window system for Plan 9. While not a direct fork of rio it is based on the same general principles and is for the most part compatible with it. We discuss the architectural differences between rio and lola, the new features implemented, and some ideas for the future.

1. Introduction

The user experience that rio provides has not significantly changed from its immediate predecessor 8½ on Plan 9 [1] and even earlier ancestor mux for the Blit terminal [2]. At the same time the code shows clear signs of evolution, and certain aspects are not easy to change. Lola can be considered a new take on the same basic architecture and principles. The goal was to separate out various components in a cleaner way, make it easier to modify the code for personal customization or to experiment with new features, and then to actually implement a few new features. Development took place in two stages: The first goal was to achieve feature parity with rio so that it could be used as a drop-in replacement for rio. The next step was to make use of the more maintainable code base and implement some new ideas and features. In the interest of not breaking too many assumptions in the rest of the system (say, about the semantics of the `wctl` file) these new features are relatively conservative. The hope is that a code base that is easier to work with will inspire more experimentation with window systems, and that more audacious ideas may be explored or at least considered as well.

2. Architecture

While lola is based on the same principles as rio a few things are handled differently. The general architecture of processes, threads and channels are shown schematically in Figures 1 and 2. This section highlights the most important differences.

2.1. Concurrency – 9p

A key implementation feature of rio (as opposed to 8½) is the use of concurrency and channels. Notably, the 9p protocol is implemented by hand (rio predates `lib9p`) and much of it is handled in a separate process. Synchronization with the main process is achieved over channels to the window and Xfid (executing fid) threads, with a thread allocating and freeing Xfids executing in the main process (see figure 1).

Lola differs in this regard in that a separate process is solely used to read raw 9p messages. These are then handled in the main process by the 9p thread, which implements the protocol using `lib9p` (see figure 2). Because 9p requests can block, a thread to handle read or write requests is still a clean solution, called an Xreq in lola. Xreqs are somewhat more focused than rio's Xfids because their only purpose is to deal with (potentially) blocking requests, as opposed to also having to execute fid-specific code in the main process. Because 9p is handled in the main process there is no need for something like rio's Xfid allocation thread.

Although the architecture is otherwise relatively similar to rio's the general tendency has been to reduce the use of threads and channels.

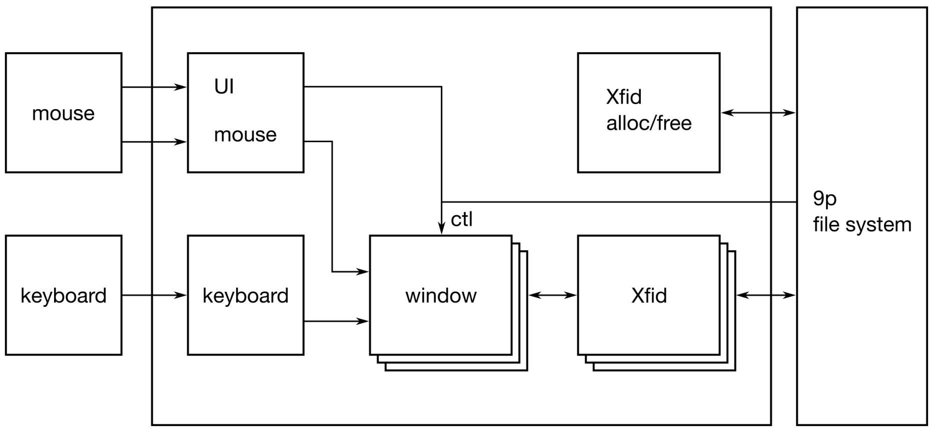


Figure 1: Structure of rio

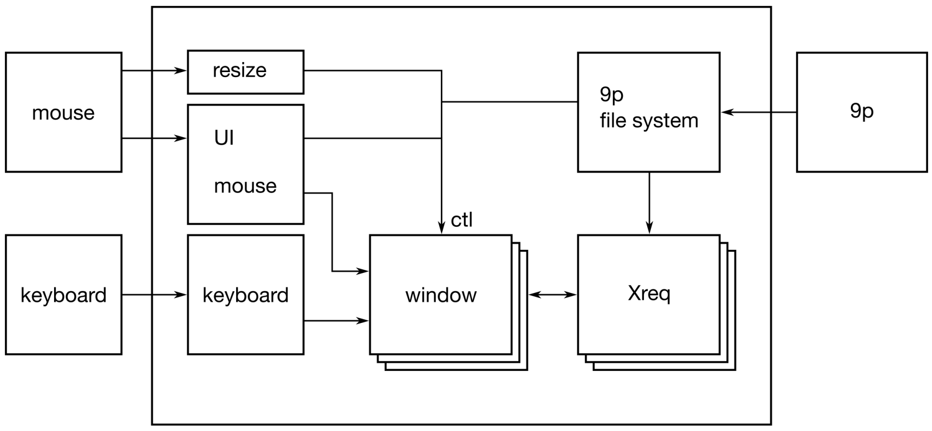


Figure 2: Structure of lola

2.2. Anatomy of a window

The core data structure in a window system is of course one representing a window. One aspect of a window is that of a rectangle on screen whose input and output is multiplexed by the window system (and the draw device). Another aspect is its two-sided nature of being either a text or a graphical window. The data structures of course reflect this.

In contrast to rio, lola separates these two aspects: everything related to the text content of a window has been put into a `Text` structure and the related code has been disentangled from the `Window` as well. In fact, with tabbed windows (see below) the rectangle and the content aspects of a window have been further broken up into `Window` and `WinTab` structures.

The resulting `Text` structure and code implements something like an extended Frame (see `libframe`) capable of dealing with text beyond the visible region. Factoring out this component of the window system not only makes it easier to understand the window related code, but also made it possible to reuse it and write a simple text editor called jot.

As for the graphical nature of a window, a major kludge was introduced into the window system when Plan 9 moved from `bitblt` to `devdraw`: while 8½'s multiplexing of `bitblt` was conceptually very clean in that an application had no idea whether it was running in a window or not, the new approach of creating and naming a `devdraw` image, and having client programs read this name from a file raises the question of how to handle window borders. The path that was taken is a shared assumption between rio and `libdraw` that a window has a 4px border that the client makes sure not to draw over by creating a screen over the window image, and then creating a second image on that screen (see `gengetwindow()` in `libdraw`). Unfortunately this makes it harder for a window system to implement arbitrary decorations, and it would be conceptually cleaner if a program simply never saw the border of its window in the first place.

Fortunately there is a solution: no 4px border is assumed if the window name starts with "noborder". Further, the image-screen-image dance can be done by the window system instead, at the cost of some overhead:

```
// create actual window on screen
w->frame = allocwindow(wscreen, w->rect, Refbackup, DNofill);
// create window-screen to allocate content window on
w->screen = allocscreen(w->frame, colors[BACK], 0);
// the image that clients will use. this excludes any decorations
w->content = allocwindow(w->screen, w->contrect, Refbackup, DNofill);
// publish image. w->name starts with "noborder"
nameimage(w->content, w->name, 1);
```

Of course it would be preferable to only build these two layers of images once. Because the original image (called `_screen` in `libdraw`) is hardly used by anything, it is expected that one could get rid of it on the client side and leave the creation of the content image to the window system instead, while retaining backwards compatibility in cases where the window is not of the noborder-type.

3. New Features

Lola implements several new features. It is remarkable that they fit rather neatly into the model established by rio, such that it was possible to avoid breaking too many assumptions and to retain backwards compatibility for the most part. The most significant features are explained in the following.

3.1. Virtual desktops

Virtual desktops are a common way to extend the workspace. This is typically done in one of two ways: either the desktop's logical size is extended, of which a certain region is visible physically, or the user switches between completely separate desktops. One may think of this as moving the visible region in xy directions in the former case, and in the z direction in the latter. Lola implements the former kind, which is rather straightforward because `devdraw` already supports off-screen coordinates. The screen offset can be changed with a 1-2 chord on the background, or through a `wctl` message.

3.2. Decorations

Window decorations are a standard feature in almost all window systems, usually providing a border around the window, and often a title bar as well. These can give some visual indication of the state of a window, like whether it is active, or in hold mode, also to display its title. Secondly, they can provide ways for the user to interact with the window, like dragging its border, or the very common minimize, maximize, and close buttons.

Like its predecessors, rio only implements a simple border. As has already been discussed, this border is a hardcoded assumption in `libdraw` and a solution to the issue has also been described already. With that change, the way for some more interesting decorations in lola was clear. To make a case for the relative ease with which new styles can be implemented and also to get a good idea of what part of the code is style-specific and what is general, four styles have been implemented, as shown in figure 3. These can also serve as a starting point for personal customization.

For the sake of simplicity only one visual style is supported at a time, by linking against the respective object file. Integrating all styles and making this a runtime option would only require small changes and is something that may happen in the future.

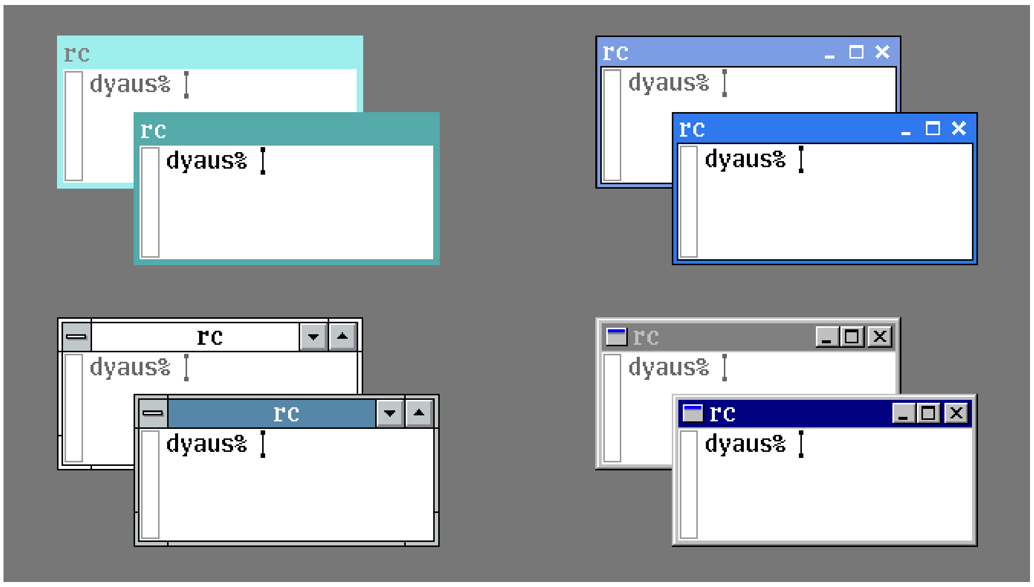


Figure 3: Different visual styles

3.3. Color schemes

A common modification of rio is a change of the color scheme. A relatively popular patch implements a `theme` file in rio, which then automatically redraws everything with the new colors whenever the file is written to. Lola has an alternative approach that directly exploits a feature already built into `devdraw`: named images. Compared to the implementation of a `theme` file this makes the code for supporting user-definable color schemes in lola very simple:

```
background = getcolor("background", 0x777777FF);
colors[TEXT] = getcolor("text", 0x000000FF);
```

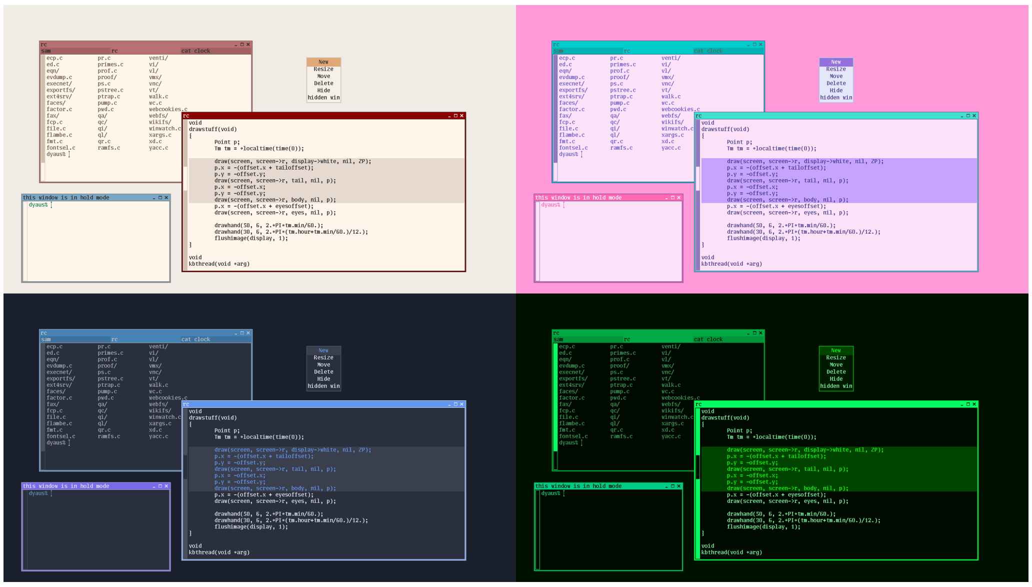
The function `getcolor()` gets a reference to an already existing named image, much like programs get the image of their window for drawing, or if it does not exist yet, creates it with a default color and publishes it under that name. Other programs like sam and acme only required small changes to use `getcolor()` as well. Using shared images frees lola (and other programs) from having to deal with details of how color schemes are applied by users. The very simple program `settheme` handles that job by reading name-color pairs from stdin and setting the colors accordingly. It also tells lola to redraw everything with a `wctl` message. Changing the color scheme requires nothing more than executing a command like `settheme < simple_vapor`, where `simple_vapor` is a color description file. Here is an excerpt:

```
background      FF99D8
back            FCE2F9
high            C5A3FF
bord            8E78BE
text            483D8B
htext           483D8B
```

Some color schemes are shown in figure 4.

The fact that all programs using `getcolor()` on the same draw device share colors can also be a drawback, because these colors are also shared across different instances of the window system. A user may however want to pick a different color scheme for a nested window system for instance, perhaps if it's running on a remote machine. One possible way out could be to maintain an environment variable holding a string that is prefixed to the raw color names. This would essentially replicate a simple

https://ftrv.se/14



```
rc
sam                 rc                  cat clock
ecp.c          pr.c          venti/
ed.c           primes.c      vi/
eqn/           prof.c        vl/
evdump.c       proof/        vmx/
execnet/       ps.c          vnc/
exportfs/      pstree.c      vt/
ext4srv/       ptrap.c       walk.c
faces/         pump.c        wc.c
factor.c       pwd.c         webcookies.c
fax/           qa/           webfs/
fcp.c          qc/           wikifs/
file.c         qi/           winwatch.c
flambe.c       ql/           xargs.c
fmt.c          qr.c          xd.c
fontsel.c      ramfs.c       yacc.c
dyaus%
```

```
this window is in hold mode
dyaus%
```

```
rc
void
drawstuff(void)
{
	Point p;
	Tm tm = *localtime(time(0));

	draw(screen, screen->r, display->white, nil, ZP);
	p.x = -(offset.x + tailoffset);
	p.y = -offset.y;
	draw(screen, screen->r, tail, nil, p);
	p.x = -offset.x;
	p.y = -offset.y;
	draw(screen, screen->r, body, nil, p);
	p.x = -(offset.x + eyesoffset);
	draw(screen, screen->r, eyes, nil, p);

	drawhand(50, 6, 2.*PI*tm.min/60.);
	drawhand(30, 6, 2.*PI*(tm.hour+tm.min/60.)/12.);
	flushimage(display, 1);
}

void
kbthread(void *arg)
```

Figure 4: Different color schemes

kind of namespacing for the global `devdraw` image names.

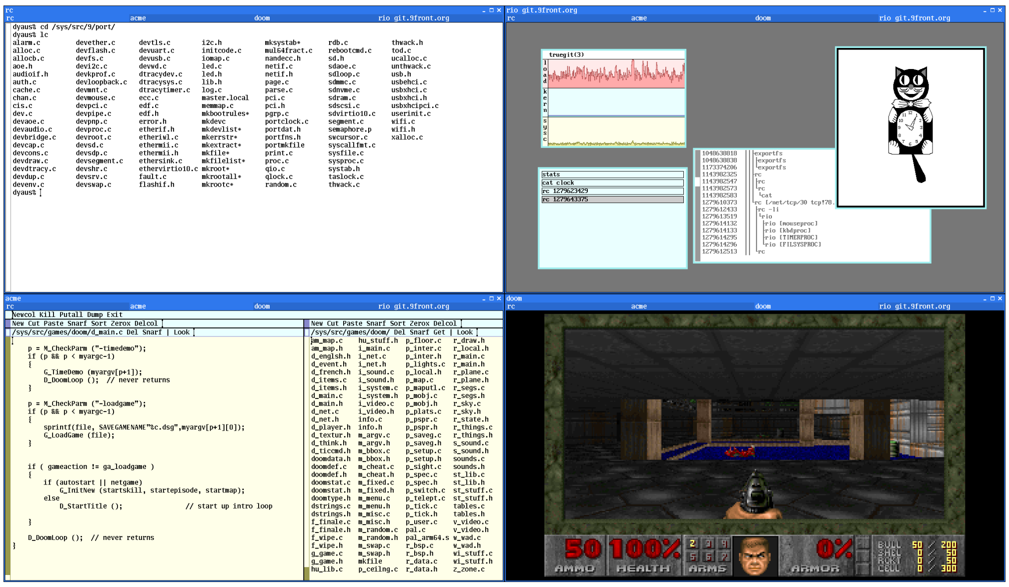
3.4. Tabbed windows

The new features of lola discussed so far are common in other window systems. Tabs on the other hand are typically application-specific, ubiquitous in web browsers and text editors for instance. Lola in contrast provides a completely general tab system at the layer of the window system: all windows no matter the content can be grouped together as different tabs of a single window.

While this feature is not completely unique to lola it is still fairly uncommon and indeed no inspiration was drawn from existing implementations. Instead, the necessary change that made window decorations possible immediately suggested to go even further: If the content of a window is an image on a screen for that window, then what could one do with multiple images on that screen? From a `devdraw` perspective tabs are completely trivial and seamlessly extend the windowing model without having to modify existing programs.

From an implementation perspective, this required a split of the single `Window` data structure into two: a new `Window` structure for the actual window rectangle and associated `devdraw` screen, and a new `WinTab` structure for the content. Tabs have a reference to the window that contains them and are organized as a linked list. Windows maintain that linked list, and also reference the currently active tab.

Figure 5 shows four tabs inside a single window: a normal text tab running a shell, rio running on a remote machine, acme exploring the source code for DOOM, and the actual DOOM game. The tab bar is shown under the title bar, but only if a window has more than one tab. It indicates the currently visible tab and is used to switch, rearrange, close, and detach tabs.
The user interface is quite simple: left-clicking on a tab switches to that tab, middle-clicking closes a tab, and right-clicking requires a second click to move a tab to a different window or detach it to create a new window. The latter can also be done on the title bar as a single-tab window does not have a tab bar to click on. Moving a tab left or right is achieved with the 1-2 and 1-3 mouse chords on that tab. To create a new tab the New menu option is used, but instead of dragging a rectangle for a new window one clicks on an existing one.



```
dyaus% cd /sys/src/9/port/
dyaus% lc
alarm.c       devether.c    devtls.c        i2c.h          mksystab*     rdb.c          thwack.h
alloc.c       devflash.c    devuart.c       initcode.c     mul64fract.c  rebootcmd.c    tod.c
allocb.c      devfs.c       devusb.c        iomap.c        nandecc.h     sd.h           ucalloc.c
aoe.h         devi2c.c      devwd.c         led.c          netif.c       sdaoe.c        unthwack.c
audioif.h     devkprof.c    dtracydev.c     led.h          netif.h       sdloop.c       usb.h
auth.c        devloopback.c dtracysys.c     lib.h          page.c        sdmmc.c        usbehci.c
cache.c       devmnt.c      dtracytimer.c   log.c          parse.c       sdnvme.c       usbxhci.c
chan.c        devmouse.c    ecc.c           master.local   pci.c         sdram.c        usbxhci.h
cis.c         devpci.c      edf.c           memmap.c       pci.h         sdscsi.c       usbxhcipci.c
dev.c         devpipe.c     edf.h           mkbootrules*   pgrp.c        sdvirtio10.c   userinit.c
devaoe.c      devpnp.c      error.h         mkdevc         portclock.c   segment.c      wifi.c
devaudio.c    devproc.c     etherif.h       mkdevlist*     portdat.h     semaphore.p    wifi.h
devbridge.c   devroot.c     etheriwl.c      mkerrstr*      portfns.h     swcursor.c     xalloc.c
devcap.c      devsd.c       ethermii.c      mkextract*     portmkfile    syscallfmt.c
devcons.c     devsdp.c      ethermii.h      mkfile*        print.c       sysfile.c
devdraw.c     devsegment.c  ethersink.c     mkfilelist*    proc.c        sysproc.c
devdtracy.c   devshr.c      ethervirtio10.c mkroot*        qio.c         systab.h
devdup.c      devsrv.c      fault.c         mkrootall*     qlock.c       taslock.c
devenv.c      devswap.c     flashif.h       mkrootc*       random.c      thwack.c
dyaus% 
```

```
1048638818   ││├exportfs
1048638838   ││ ├exportfs
1173374206   ││ └exportfs
1143982325   ││├rc
1143982547   ││ ├rc
1143982573   ││ └rc
1143982583   ││  └cat
1279610373   ││└rc [/net/tcp/30 tcp!78.
1279612433   ││ ├rc -li
1279613519   ││  └rio
1279614132   ││   ├rio [mouseproc]
1279614133   ││   ├rio [kbdproc]
1279614295   ││   ├rio [TIMERPROC]
1279614296   ││   └rio [FILSYSPROC]
1279612513   ││ └rc
```

```
Newcol Kill Putall Dump Exit
New Cut Paste Snarf Sort Zerox Delcol
/sys/src/games/doom/d_main.c Del Snarf | Look

    p = M_CheckParm ("-timedemo");
    if (p && p < myargc-1)
    {
        G_TimeDemo (myargv[p+1]);
        D_DoomLoop ();  // never returns
    }

    p = M_CheckParm ("-loadgame");
    if (p && p < myargc-1)
    {
        sprintf(file, SAVEGAMENAME"%c.dsg",myargv[p+1][0]);
        G_LoadGame (file);
    }


    if ( gameaction != ga_loadgame )
    {
        if (autostart || netgame)
            G_InitNew (startskill, startepisode, startmap);
        else
            D_StartTitle ();                // start up intro loop

    }

    D_DoomLoop ();  // never returns
}
```

```
New Cut Paste Snarf Sort Zerox Delcol
/sys/src/games/doom/ Del Snarf Get | Look
am_map.c    hu_stuff.h  p_floor.c    r_draw.h
am_map.h    i_main.c    p_inter.c    r_local.h
d_englsh.h  i_net.c     p_inter.h    r_main.c
d_event.h   i_net.h     p_lights.c   r_main.h
d_french.h  i_sound.c   p_local.h    r_plane.c
d_items.c   i_sound.h   p_map.c      r_plane.h
d_items.h   i_system.c  p_maputl.c   r_segs.c
d_main.c    i_system.h  p_mobj.c     r_segs.h
d_main.h    i_video.c   p_mobj.h     r_sky.c
d_net.c     i_video.h   p_plats.c    r_sky.h
d_net.h     info.c      p_pspr.c     r_state.h
d_player.h  info.h      p_pspr.h     r_things.c
d_textur.h  m_argv.c    p_saveg.c    r_things.h
d_think.h   m_argv.h    p_saveg.h    s_sound.c
d_ticcmd.h  m_bbox.c    p_setup.c    s_sound.h
doomdata.h  m_bbox.h    p_setup.h    sounds.c
doomdef.c   m_cheat.c   p_sight.c    sounds.h
doomdef.h   m_cheat.h   p_spec.c     st_lib.c
doomstat.c  m_fixed.c   p_spec.h     st_lib.h
doomstat.h  m_fixed.h   p_switch.c   st_stuff.c
doomtype.h  m_menu.c    p_telept.c   st_stuff.h
dstrings.c  m_menu.h    p_tick.c     tables.c
dstrings.h  m_misc.c    p_tick.h     tables.h
f_finale.c  m_misc.h    p_user.c     v_video.c
f_finale.h  m_random.c  pal.c        v_video.h
f_wipe.c    m_random.h  pal_arm64.s  w_wad.c
f_wipe.h    m_swap.c    r_bsp.c      w_wad.h
g_game.c    m_swap.h    r_bsp.h      wi_stuff.c
g_game.h    mkfile      r_data.c     wi_stuff.h
hu_lib.c    p_ceilng.c  r_data.h     z_zone.c
```

Figure 5: Four views of a tabbed window. Only one tab is visible at a time.

4. Conclusion

Lola demonstrates that the architecture established by devdraw and rio are still fertile ground for experimentation: changes to the concurrent architecture made the code easier to understand while retaining the same basic structure, making use of lib9p and moving the text handling of a window into a separate component relieved the window system proper of some of its duties and allowed for the quick implementation of the simple text editor jot. Implementing the features described above turned out to be a relatively smooth process. It is perhaps remarkable how naturally the implementation of color schemes and tabs emerged out of devdraw's design, and how well they integrate with the system.

Since tiling window managers enjoy quite some popularity (espectially on Linux) it would be interesting to see how a tiling system could be implemented based on lola. While wctl messages can already replicate some of the functionality with rio, a dedicated tiling window system is likely to have a somewhat better user experience.

A further direction might be to reconsider whether the text mode of a window should really be the job of the window system in the first place. Perhaps implementing a simpler set of window-files would make it possible to implement conventional window behaviour on top of these, similar to acme's win, though support for fully graphical windows may turn out to be a challenge.

To conclude, lola has a few users already and feedback so far has been positive. It is fun to experiment with new ideas for window systems with lola and on Plan 9 in general. The challenge is to avoid feature creep and to find a nice set of features that work well together and with the rest of the system.

4.1. References

[1] Rob Pike, ‘‘The Blit: A Multiplexed Graphics Terminal’’, *Bell Labs Tech. J.,* V63, #8, part 2, pp. 1607-1631.

[2] Rob Pike, ‘‘8½, the Plan 9 Window System’’, *USENIX Summer Conf. Proc.,* Nashville, June, 1991, pp. 257-265,

Nile: A More Transparent Window System

Anthony Sorace
a@9srv.net

ABSTRACT

Nile is a window system for Plan 9. It is derived from *rio*, the default window system for the 4th Edition of Plan 9, with which it shares many properties. Nile offers a simplified user interaction model by making common actions easier, removing almost all menu interaction, and updating the scrolling behavior. It also offers a non-interactive mode useful for constrained environments and makes a number of smaller changes. The effect is to reduce the cognitive load of using the system by making it more *transparent*, in the sense of rio's original author.

1. Introduction

Window systems in Plan 9's lineage have always reached for a form of simplicity described by their author as *transparency*. A description of an early example of the line, *mux*[Pike88], is described by its author this way:

> Mux is successful because it is transparent and unobtrusive. This does not mean that its services are minimal, but that it presents them simply enough that they can be assimilated and used unconsciously. After some practice, a user of mux can exploit its facilities without thinking...
>
> — Rob Pike, *Window Systems Should Be Transparent*

Rio, the default window system for Plan 9 since the release of the 3rd Edition, embodies this principle well. It provides a very simple interface, both evaluated on its own merits but especially in comparison to nearly every other window system in use today. When taken as part of the lineage of mux and 8½, rio looks to many users like the simplest interface practical, without giving up fundamental things like arbitrarily-positioned overlapping windows. But reading descriptions of mux might lead a reader to similar conclusions if they had not experienced later systems.

With many years use of rio, some areas where it is more opaque than it needs to be begin to show. The principles which have led to simplifying menus and easing user interaction between generations in the line still hold, but the line has not advanced since the release of 3rd Edition in 2000. In addition to lessons learned from long use, the environment of use has changed in that time as well. More use of Plan 9's window system runs over a network, often a wide-area network like the public Internet. Resolution of both screens and mice have increased significantly.

Nile takes the principles that informed the creation of rio and re-visits some of the decisions those principles led to. By re-evaluating their application with the benefit of extensive use, it offers some experiments in advancing the state of window systems on Plan 9. Some of these experiments yield mixed results or expose the limits of their applicability, but overall the result is a more streamlined, *more transparent* window system that reduces the cognitive load on the user and makes it easier for the system's facilities to "be assimilated and used unconsciously". [Pike88]

Nile is available at http://a.9srv.net/src/nile/

2. User Interaction Changes

The main focus of work on nile has been a set of changes to the way the user interacts with the system graphically. The familiar menus for editing and window management are gone. Elements from those menus which were redundant with other interactive means of providing the same function are removed, while the most common remaining functions are tied directly to mouse buttons. Two remaining elements are provided via other means.

2.1. New window creation

A user creates a new window by pressing mouse button 3 anywhere on the background, dragging the mouse to describe the desired shape of the window, and releasing the button. No menu is involved. This has several benefits.

Window creation is often a no-look operation.

In the best case — a newly-launched instance of nile with no existing windows — the user may sweep out a window without looking at the point of origin. In the most common cases, with several windows on the screen but none extending beyond it, the way *draw*'s coordinate system works with nile's borders and Plan 9's mouse mean that the right-hand and bottom edges of the screen retain this "no-look" property. Common application[1] of Fitts' Law [Fitts64] to mouse-driven computer interfaces shows that the edges of the screen are the easiest targets for a user to hit. And in any case, provided any bit of the background is visible, the user must only give the cursor enough attention to position it over the background.

It is common for users who regularly use rio at the same workstation to develop significant muscle memory (or simply habits) for how creating a window of (roughly) a given shape feels. Nile makes that muscle memory more useful.

We can compare window creation in nile to how it was described in mux [Pike88], which matches how it is done in rio:

> Consider the actions required to make a new window: pressing button 3 anywhere on the screen, moving the mouse over New on the menu, releasing button 3, moving to one comer of the desired window, pressing button 3 again, moving to the diagonally opposite corner, and releasing button 3. Unless windows are created with default size or position, this sequence cannot reasonably be made simpler.

While this is a simple approach, it is 7 steps, with a required change of mental focus to check the menu status, a decision about whether New on the menu is above, below, or at the current mouse position, and a relatively precise motion likely required to select New. The equivalent in nile is to move to any visible background area, press button 3, drag out the window from there, and release button 3. Fewer steps, fewer fine motions, and fewer changes in mental focus.

No variability in function

In nile, button 3 on the background will always create a new window. This is in contrast to *rio* and its ancestors, where *New* must be selected from a menu — a menu where the default action, and the mouse's position in the menu, changes. Selecting the correct item from a menu, even if only to confirm that the correct item is selected, requires the user to divert some of their attention away from what they are trying to do and towards the window system. Failing to divert attention in this way will result in errors (e.g. selecting *Delete* instead of *New*).

Earlier systems in this lineage manage their menus with the principle *Remember what the user did last time*. While this is a good principle for selecting a default menu item, it is important to note that it only goes so far. The original articulation of this principle [Pike88] uses *send* as an example, and while that is valid, it also represents the best case scenario for the principle. It holds up better for editing operations than for window management; if a user has just created a window, it is more likely their next action will be to move or resize it than to create a new one. The stated principle might be the best way of handling many menus, but it is better yet to eliminate the ambiguity by eliminating the menu itself.

[1] See https://blog.codinghorror.com/fitts-law-and-infinite-width/ for an explanation of applying the original findings to user interfaces.

Efficient over high-latency networks

If the window system takes a significant amount of time to draw its menus, the delay will disrupt whatever muscle memory the user has developed and force them to reorient their attention on the menu. Also, time spent waiting for an interactive system is much more perceptible to users than time spent actively using the system. [Cald09][Kohrs16] While the delay is imperceptible when the window system is running on the same hardware the user is sitting at, and generally negligible over a fast local network, this is not always the case. A *cpu* or *drawterm* connection over the public internet will generally include enough of a delay to cause this disruption of attention, even over a high quality link. Over higher-latency links the delay increases,[2] moving the disruption from a subconscious loss of attention into conscious frustration.

2.2. Send and Plumb

Clicking button 2 on any window will now *send* either the selected text in that window or, if there is none, the contents of the snarf buffer: the text will be appended to the output point for the shell (or other running program) to interpret. The results are the same as the *send* menu item from *rio*'s button 2 menu. Clicking button 2 on the background will *send* to the active window, if any.

Similarly, clicking button 3 within a window will *plumb* the current selection or snarf buffer. The results are the same as the *plumb* menu item from *rio*'s button 2 menu.

In both cases, we see many of the same benefits from nile's method for creating new windows. By preserving the *no variability in function* property, there is no menu to select an item from, freeing up the user's attention and eliminating a class of errors.

2.3. Hidden windows

Pressing button 1 anywhere on the background will display a list of currently hidden windows, in the order in which they are hidden. This is the only remaining menu in the system; eliminating it would make the use of an external program for restoring hidden windows a requirement. See "Supporting Programs", below, for one approach to this.

2.4. Cut, Paste, and Snarf

Nile keeps the same mouse chording behavior inherited from rio: after selecting text with button 1 and keeping the button depressed, pressing button 2 will cut the text and button 3 will paste from the snarf buffer. The *cut*, *paste*, and *snarf* menu items in rio are already redundant ways to provide these functions, so they are eliminated in nile. The mouse chords are preferable to their equivalent menu selections for several reasons: they keep the user's focus on the data being acted on; since the chording actions are unambiguous, they can be performed more quickly with a lower rate of errors; and users are able to develop muscle memory around the actions in ways they cannot when using menus where the initially selected item shifts between invocations of the menu.

2.5. Scroll behavior

Nile imports the scroll behavior used by Acme and 9term in Plan 9 from User Space. Briefly, if the output point of a window is visible, text printed to the window may cause the window to scroll if it extends beyond the bottom of the window; if the output point is not visible, the window will not scroll. The user can always stop scrolling once the screen is full with a single up arrow, page up, or mouse click in the scroll bar, and restore scrolling by tapping End.

This eliminates the need to track the current scroll mode, both from the code and for the user. There is no "mode" in the sense of something toggled or selected, and to the extent that the behavior can vary (will output cause the window to scroll?), it varies based on something already visible to the user: the content they are likely focused on. Making the behavior dependent on already-visible information reduces the chances of a user being surprised by the system's behavior.

[2] At the time of writing, it consistently takes over 1.5 seconds for rio to display its menus in drawterm when connecting to a server over the public internet via the author's home DSL line.

In older systems in this lineage, scroll behavior is tracked by the system as a distinct mode, an explicit state set and managed by the user. This setting is long-lived, and entirely context dependent: each window has its own mode. Worse, in the normal course of using these systems, the user is given no indication of the current scroll mode of any window.[3] Surprise at finding themselves in a different mode than expected becomes a familiar — but still unwelcome — part of using these systems. By eliminating distinct scroll modes and making the system's behavior predictable based on casual observation, nile eliminates these surprises and the errors common to modes. [Tesler81]

2.6. Moving and Resizing Windows

Nile allows users to move and resize windows by dragging the window borders. The model is the same as rio's: using button 1 to drag an edge will resize the window by dragging that edge and using it on a corner will resize by dragging the adjoining edges, while using button 3 anywhere on the border will move the entire window.[4] Nile omits the redundant ability to also trigger these actions via a menu.

When a user wants to make small adjustments to a window, using rio's Resize menu option requires the user to carefully match the portion of the existing rectangle they want to maintain, so dragging the border has typically been the preferred method. Resize, then, can be seen as an optimization for larger geometry changes. Removing this makes larger reshaping of windows marginally less convenient, but the difference is typically only one extra mouse action and uses the existing border resizing operations.

2.7. Non-Interactive Mode

When launched with an argument *-I*, nile does not provide any menus (including the normal menu of hidden windows), and windows have no borders (and thus cannot be moved or resized through the user interface). It is, from a graphical perspective, entirely non-interactive, although programs running within certainly may be. Windows can still be manipulated via the file system. This usage is intended for constrained devices and is for use with external programs like *winwatch*, which can be used to at least switch between windows, and possibly perform other similar tasks. A fuller exploration of this non-interactive mode is outside the scope of this paper.

3. File System Changes

Nile inherits from rio the property of providing many facilities via a file system interface. It implements the same file systems as rio, with one addition: the *wctl* file system adds a file *winfont* file which holds the name of the font currently used in the window. The conventional way for programs to find this is to consult the *$font* environment variable, but this has no inherent tie to the reality of what is displayed, and there are several common ways for it to not line up in practice. Most commonly, if a user's *lib/profile* sets *$font* differently in different cases, such as using different fonts in the *terminal* and *cpu* case of the *$service* switch, *cpu*ing to another machine will often produce this effect. As cpu typically binds in the terminal's devices, programs which use *winfont* can avoid this problem.

4. Appearance Changes

Nile makes three sets of changes to its appearance relative to rio. While not as significant an impact for users as the changes described above, they represent informative experiments in their own right.

4.1. Border-only Window Creation

When interactively creating a new window, nile draws only the border of the window as it is being dragged out, in contrast to rio, which fills the rectangle as it is being created. When running over poor network connections this reduces perceived lag.[5] The overall reduction in draw operations sent over the network helps, but the more significant issue is that the draw operations remaining affect a much smaller amount of visual space. When the swept rectangle displayed lags what the user has selected, that

[3] This is in contrast to rio's "hold mode", which is clearly visually indicated to the user.

[4] Button 2 currently has the same effect as button 1, but this may change.

[5] This was originally pointed out by Andrey Mirtchovski, who observed the perceived lag even on local, unaccelerated hardware with high enough screen resolution and depth.

discrepancy is more obvious when presented as a filled-in rectangle as opposed to a thin border simply because more pixels are now "wrong". This is compounded by the fact that the window is typically redrawn several times, each (save the last) lagging user input.

4.2. Thinner Borders

Compared to rio, nile reduces the width of the border surrounding windows. The average modern mouse is more precise than its equivalent when nile's predecessors were written,[6] reducing the benefit of wider borders. Given a border of a given width, a more precise pointing device makes it easier to grab the border. When nile was first written, optical mice with good precision had already become ubiquitous; a small reduction in the border width was expected to have negligible impact on the usability of the system while recovering a little space and reducing non-content screen elements. For many years of use, this proved true: nile's borders were no more difficult to grab, a fact underscored by the increased need for this operation, since nearly all operations changing a window's geometry required grabbing a border. In recent years, though, two things have changed that call this decision back into question.

First, nile was originally created on a terminal using a monitor with a pixel density of 98 ppi, similar to the effective density of most monitors earlier systems in this lineage used.[7] In similar environments, the smaller window border remains comfortable, but it is common for modern screens to operate at double that density or more. In those environments, the borders become too difficult to grab. Increases in mouse sensitivity have more than kept pace with increases in screen resolution and pixel density, but human visual acuity and motor control have not. When running around double the pixel density of the original assumptions, even rio's borders can be too difficult to grab in some environments.

Second, while nile is designed primarily for a three button mouse, it is worth considering other common configurations. It is more common to find users with a trackpad, and only a trackpad, than in previous decades. Even high-quality trackpads are less precise than average modern mice, again making the narrower borders harder to grab than rio's. Worse, this combines with the previous issue; the impact is most noticeable when on a laptop with a high-resolution display.

Overall, the thinner borders were a useful experiment, but provided mixed results for usability, depending on the terminal hardware. Across a representative set of terminals, the effect is believed to be a net loss.[8] But we need not fit all cases to an average. If Plan 9 exposed the ppi of attached displays, it would be tempting to scale its borders by the ppi of the display, providing a constant-sized target for users to grab. However that would address neither the differences in input methods (e.g. a trackpad vs. a mouse) nor the differences between users in visual acuity or fine motor control. Instead, nile should allow users to override the default value to ensure grabbing a border remains a comfortable operation.

4.3. Darker Colors

Since the introduction of nile's predecessors, computer displays have gotten brighter and contrast ratios have improved.[9] At the same time, users are less likely to find themselves in well-lit shared office environments; working from home, each user may tailor their environment to personal preference, outside the constraints of users at neighboring desks or OHSA guidelines.

Nile dims rio's grey background and changes text windows to be white text on a black background.[10] This significantly reduces the overall light emitted which, combined with reducing contrast with the surrounding environment, reduces long-term eye strain. At the author's terminal, with a modern display at medium brightness, rio with half the screen covered in text windows produced 40-50 lux when measured 3'

[6] High-quality optical mice existed before 8½ and mux, but the device any given user was using was likely to be much less precise. The first optical IntelliMouse was considered a significant advancement in precision for consumer mice upon its release in 1999, offering 400 dpi; at the time of writing, the top 10 mice by sales on Amazon offer from 1000 to 8000 dpi.

[7] The blit, which used mpx and mux, had a 15" display at 800x1024 pixels for an effective ppi of 87.

[8] They remain in the system partly pending a satisfactory resolution to the broader questions of this paragraph, but mostly because of the rule *Design with a specific customer in mind*, where, in this case, that customer is the author, who's primary terminal involves a ˜100 ppi display and a 3-button mouse.

[9] Comparing CRT displays to any of the later panel types is difficult because of how different the technologies are, but this is certainly true within panel displays and *roughly* true when including CRTs.

[10] The colors of selected and held text are adjusted to ensure good contrast with the white-on-black text.

from the display in a dark room; nile in the same situation produced about 4-6 lux. Crucially, there is no perceived degradation in readability of text.

Like the border size changes, these color changes have been a useful experiment that worked well in the environment it was crafted in but represent a net loss across a set of representative environments. Reading white-on-black text windows is more pleasant in dim environments, marginally less pleasant in brightly-lit rooms, and significantly less pleasant in sunlight. Rio's choices here are, again, a better fit for an average sampling of environments, but, again, we don't need to constrain ourselves to an imagined average environment. Existing patches which add theming to rio should be brought into nile.

5. Supporting Programs

Since nile mediates access to Plan 9's devices in the same way as rio and presents a superset of rio's file system, all existing Plan 9 graphical programs run under nile unmodified. A few new programs and changes to existing programs make using nile even more comfortable.

A modified version of *winwatch*(1) provides a menu when button 3 is pressed on a window name with the options *Show*, *New*, *Delete*, and *Hide*, performing the expected actions. The *Hide* and *Delete* items cover the interactive functionality present in rio but absent in nile's graphical interface. This model gives up some usability by no longer having the menu local to the window it is acting on, but gains some by being able to act on any window, even if it isn't visible.

A short script, *Hide*, does the appropriate writes to nile's file system to hide the window it is run in. It is convenient to leave in a tag in Acme. Similarly, *Delete* will delete the window it is run in. They are both trivial wrappers around *wctl*, which finds the control file corresponding to the current window system and performs the write.

A modified version of *mc*(1) makes use of the new *winfont* file to determine the font for sizing. When connected to a remote server via *cpu*, commands which use *mc* will now display correctly, even in the common case where the *$font* is set differently on terminals and cpu servers.

6. Future Work

The *winfont* file should be writable to cause *nile* to redraw the window in the new named font, similar to how Acme uses the frame library for this effect.

Nile should be modified to support theming, in line with existing work to add theming to rio. This should include things like window border sizes to allow users to set a size appropriate for their hardware.

With the exception of the addition of the *winfont* file, nile provides the same file system interface as rio and understands the same language. For the first several years of nile's development, the source files providing the file system interface (and utility functions) were simply `#include` directives referencing rio's version of the same files. Now, nile duplicates most of that code. This suggests it might be interesting to more formally or fully decouple the file system interface from the parts handling user interaction, allowing easier revision of each and easing code maintenance.

The code for nile contains an unfinished implementation of a *Look* menu item for searching with a text window. It is unfinished because there was no obvious way to make the function available once the menus were removed. The implementation should be finished and exposed.

7. Conclusion

Nile, in approximately the form described here, has been in active daily use by the author for over a decade and has proven quite comfortable. The mouse changes for new window creation, send, and snarf actions have been particularly satisfying. A small number of additional users have provided similar feedback. Eliminating most menus, and the errors and distraction associated with them, has reduced frustration, while making the most common actions from those menus directly available with a single click makes using the system feel much faster. The experiments with more aesthetic elements of the system have been qualified successes, demonstrating that while good defaults are important, deviating from them can produce better results when the environment changes. Overall, making the window system *more* transparent makes using it even more pleasant: more direct, quicker to use, and occupying less of our attention.

8. Acknowledgements

Nile’s code is directly derived from rio, without which it would not have been possible. It is a testament to the quality of rio’s code and design that nile gets rather significant benefits from relatively little change in the code. In addition to being a comfortable environment for users, rio provides an accessible field for trying out new ideas in window management and user interface.

The overall outline of the menuless operation was suggested in a message from Russ Cox to the 9fans mailing list,[11] and the scrolling behavior is a straight-forward import of his implementation of scrolling in Plan 9 from User Space.[12] The border-only window creation is adapted from work by Andrey Mirtchovski[13] and the non-interactive mode is adapted from work by Jesús Galán López.[14]

References

[Pike88] R. Pike, "Window Systems Should Be Transparent", *Computing Systems*, Vol. 1, No. 3, 1988

[Fitts64] Fitts, P. M., & Peterson, J. R., "Information capacity of discrete motor responses", *Journal of Experimental Psychology*, Vol. 67, No. 2, 1964
doi: 10.1037/h0045689

[Tesler81] Tesler, L., "The Smalltalk Environment" *Byte Magazine*, Vol. 06 Num. 08

[Cald09] Caldwell BS, Wang E., "Delays and user performance in human-computer-network interaction tasks", *Human Factors*, Vol. 51 Num. 6, 2009
doi: 10.1177/0018720809359349

[Kohrs16] Kohrs C, Angenstein N, Brechmann A., "Delays in Human-Computer Interaction and Their Effects on Brain Activity", *PLoS One*, Vol. 11, Num. 1, 2016
doi: 10.1371/journal.pone.0146250

[11] https://9fans.topicbox.com/groups/9fans/T28ea22f6a6244a70-M630c2ec5279eae746ee05901
[12] https://groups.google.com/g/plan9port-dev/c/hTUKYay_HjQ/m/ItpdtNht8NIJ
[13] https://9fans.topicbox.com/groups/9fans/Te5bccc2756ef6b37-M6c7739d800aabc2a27997800/
[14] http://9p.io/sources/contrib/yiyus/cmd/rio/

Amber: CSP-driven TTY

Jonathan Frech ⟨info@jfrech.com⟩

Keywords: **Plan 9, Go, TTY graphics**

Abstract

Amber is a work-in-progress Go library exploring the design space of CSP-powered, state-meagre human-machine interaction software substrate. [Pik89]

Graphics have been at the forefront of technological driving forces since the early days of silicon use. Gaming has since its inception been intertwined with purpose-built graphics hardware and ever-fantastical advances in rendering. GPUs are the only non-minified (sound cards, FPUs, peripheral cards, network cards, etc. all got subsumed by motherboards SOC components) remnants of the once rich zoo of coprocessors. DMA considerations to this day tear apart graphics library separation concerns, driving up complexity.

Yet despite mind-boggling resolutions of new, the humble TTY has stayed fairly constant in its holdable entropy, only gaining a few rows and columns as well as font features and colours; in no small part due to written media employing the eversame, already-digital letter—no pixel had to be invented and continuously improved upon.

The potential which lies in said small state space is realized by combining value-centric operations with CSP to create a bare-bones, self-similar TTY driving API.

Amber can be seen as my diary following one mind's fascination exploring a romantisized view of '80s technology and wrestling with how to unearth its beauty in present-day computing systems.

The interface

Core to Amber's model is the `Window` interface. It tries to distill down what it means for any component to inhabit the screen and it is through strict adherence to this interface that the bedrock for self-similarity [Pik88] is laid.

```
type Window interface {
  Keystroke() chan<- keystrokes.Keystroke
  Suspend() chan<- chan<- chan<- struct{}
  Terminate() chan<- chan<- struct{}
  Dimensions() chan<- gfx.Dimensions

  Charmap() <-chan gfx.Charmap
}
```

`"amber/gfx".Dimensions` is `image.Point`-like (though its semantics include the origin to implicitly spawn a rectangle) as is `"amber/gfx".Point`. `"amber/keystrokes".Keystroke` models various keyboard-sourced input in an extendable way using weak grounding interfaces.

```
type Keystroke interface {
  Letter() rune

  fmt.Stringer
}
```

`"amber/gfx".Charmap` is `image.Image`-like. `"amber/gfx".Char` is an `"image/color".Color`-like representation of a singular twice-as-high-as-wide character (support for quadratic characters such as modern ideographs is planned to be supported by this design yet questions regarding z-fighting are not yet fully settled[1]).

```
type Charmap interface {
  Peek(Point) Char
  Dimsnsions() Dimensions
}

// more feature-rich characters announce themselves using
// receivers satisfying them in dynamic type inspection
type Char interface {
    ASCII() byte
}
```

[1]My current understanding is that XTerm would paint overwide characters placed too tightly on a line from left to right, though I am unsure of its true behaviour and expect intricacies to foil any rash implementations' hopes for correctness.

0-1-notion of time

Grounded heavily on channels, the system has a purely sequential notion of time. Thus, usability is tied to components not cumulatively exceeding a user's willing attention span.

As [Pik89, pp. 148–149: 8. Deadlock] notes, the solution to this is adding timeout-aware identity-purporting processes at interface boundaries which voluntarily decouple themselves from their fellow processes once their temporal strain exceeds acceptable bounds.

Care about strenuous parts of the system clogging channels needs to not only be taken based on fears of deadlock. Because `"amber/gfx".Charmap.Peek` returns individual characters as an interface, tying too much lazy computation to the atomic unit of display leads to a quadratic screen-wide load. Buffering charmaps of heavy computational complexity in a decoupled process is advisable to not inadvertently give up on responsiveness.

Ctrl-Z[2]

Via `Window.Suspend`, background-sending user intentions can penetrate to the leaf parts of the system, giving them a chance to do custom, provisionary cleanup (such as flushing buffers in an editor). A philosophical knot exists when talking about a program put to sleep deciding on how to sleep: OS-level job control is beyond the semantics providable by Go's runtime as it itself is subject to it in a non-negotiable hierarchical sense[3,4].

Such is vital for the Amber TTY driver to intercept `SIGTSTP`, provision for everything necessary and self-rest via `unix.Kill(0, unix.SIGSTOP)`.

Noteworthily to Amber's design is that the OS-performed split of pressed keyboard keys into key and signal events is logically kept and merged in with its API boundary.

Boons

Not a lot of software has yet been written using Amber. But the few example programs which have been written demonstrate remarkable responsiveness. Common behaviours such as Htop's[5] polling updates where system time and system resource usage are updated in fixed cadence are not exhibited by Amber-based designs. Separate sequential processes communicate and let the Amber TTY driver know when the currently displayed charmap is aged, updating unconstrained by some master clock. Seeing a player character controlled by the keyboard move and a sub-second-precise clock unperturbedly keep its rhythm truly transcends the user experience from that of a plodding terminal to something joyfully fresh.

Artifacts such as Htop—when banishing it to the background, changing XTerm's resolution and welcoming it back—for a brief moment displaying the old state only to violently jerk awake are also not expressed by Amber's TTY driver. Intricate Ncurses initialization chants, whose subtle misinterpretations are likely causes for the described unwelcome user experiences, are expressly absent from Amber's driver API.

Future work

The other side, translating terminal-displaying programs into Amber's interface, is planned via pseudoterminals, though not yet implemented. When both sides are some day satisfactorily available, it is planned to fashion a cross-machine, OS-agnostic, network-tunneled computing environment.

References

[Fre23a] Free Software Foundation. *The GNU C Library ; Job Control Signals*. 2023. URL: `https://www.gnu.org/software/libc/manual/html_node/Job-Control-Signals.html#accessed:2024-11-28`.

[Fre23b] Free Software Foundation. *The GNU C Library ; Termination Signals*. 2023. URL: `https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html#accessed:2024-11-28`.

[Pik88] Rob Pike. "Window Systems Should Be Transparent". In: *Computing Systems* 1.3 (1988), pp. 279–296. URL: `https://www.usenix.org/legacy/publications/compsystems/1988/sum_pike.pdf#accessed:2024-04-02`.

[Pik89] Rob Pike. "A Concurrent Window System". In: *Computing Systems* 2.2 (1989), pp. 133–153. URL: `https://www.usenix.org/legacy/publications/compsystems/1989/spr_pike.pdf#accessed:2024-04-01`.

[2]Or on a system of either abstract or ancient presence, Ctrl-`(*unix.Termios).Cc[unix.VSUSP]+'A'-1` of package `"golang.org/x/sys/unix"`.

[3]"The SIGSTOP signal stops the process. It cannot be handled, ignored, or blocked." [Fre23a]

[4]"[I]f SIGKILL fails to terminate a process, that by itself constitutes an operating system bug (...)"[Fre23b]

[5]Tested on Htop version 3.2.2, cf. `https://htop.dev/` [2024-11-28].

Static Initialization of Bitfields in the Plan 9 C Compilers

Jonas Amoson
jonas.amoson@hv.se

University West
461 86 Trollhättan, Sweden

ABSTRACT

Bitfields in C is one way of packing data to keep down memory footprint, but the Plan 9 compilers do not support static initialization of structs containing bitfields. A patch to the compilers is presented, that makes maintaining ports of external code easier. Modifications has been made separately for every architecture in the current implementation.

1. Introduction

The Plan 9 C compilers support bitfields in structs, but it is not possible to initialize them using constant arrays. This paper discusses an implementation to also allow static initialization of structs containing bitfields, with the aim to ease maintaining ports of evolving software with external sources, such as Netsurf [1]. The goal is to compile the cores of these software without having to revert to manual patching of the upstream source. These ports most often already make use of the Plan 9 POSIX environment (APE) or the upcoming NPE [2], and the bitfield patch described in this paper is included in Jens Staal's APExp [3] where it has already helped porting gmake.

2. Bitfields

Bitfields make it possible to pack multiple integer-types together in a struct specifying the length of each member down to the bit. This makes it possible to keep the footprint of the data smaller, without having to resort to bit manipulation in the source code using bit-wise *AND/OR* and bit-shifting. A struct with bitfields can look like:

```
struct Bitfields {
        unsigned int a : 5;     /* 5 bits */
        unsigned int b : 3;     /* 3 bits, together 8 bits */
} bf1;
```

Support for bitfields has been in C since its introduction [4, pp.132] but is not too frequently used in modern code, where most often speed of execution is more important than memory footprint. Details about how bitfields are to be implemented, such as their maximal size and exact memory layout are also undefined, which make them less useful for describing the exact byte and bit layout for protocols or when interfacing hardware.

Bitfields are handled like any other struct members, with some restrictions, and are accessed using the point notation:

```
bf1.a = 9;
bf1.b = 4;
```

It is also possible to statically assign values to all bitfields in a struct using a vector:

```
    struct Bitfields bf2 = { 9, 4 };
```

in which case a constant array of bytes representing the memory of the struct can be calculated in compile-time, avoiding run-time bit manipulation for the initialization.

3. The Plan 9 compilers

The Plan 9 compilers support bitfields, but it is not possible to initialize a struct containing bitfields using a constant array vector, as with `bf2` above, and the compiler stops with a “cannot initialize bitfields” error, if encountered.

When initializing a struct (without bitfields) using an array, the compiler generates DATA lines, each containing a 32-bit constant, for each member of the struct, which end up in the data segment in the *a.out* executable. Bitfields are to be packed denser, so in the example above, a and b should end up in the same 32-bit constant, even if they are separate members of the struct.

4. Implementation

The compiler patch [5] disables the error message that it can’t initialize bitfields, and instread generates a DATA line for the first bitfield in a struct leaving the remaining bits of the 32-bit constant zeroed. For subsequent bitfields, the last constant is updated in the output tree if they fit within the last 32-bit constant. More precisely, cc/dcl.c and [5678]c/swt.c are modified so that:

1. if a bitfield is encountered in a struct, a DATA line for a 32-bit constant is generated, but a pointer is kept to the place of the constant in the output tree.
2. if the bitfield is followed by subsequent bitfields, the constant in the output tree is updated using the pointer, and no new DATA lines are generated.
3. if a subsequent bitfield doesn’t fit in the remaining bits of the previous bitfield constant a new DATA line is instead generated, and the pointer in the output tree is updated to point its constant instead.
4. `gextern()` in swt.c is modified to update a global pointer to the constant (vconst) for which an assembly line is generated. Note: the global pointer is of type `long*` on all architectures (bitfield constants are four bytes), even though vconst is `vlong` on 64-bits architectures.

The current implementation requires separate modifications in swt.c for every architecture and supports currently used architectures in 9front (arm, amd64, arm64 and 386) which are all little-endian. It would be possible to extend the patch to work on big-endian architectures as well, with some further modifications in swt.c.

5. Discussion

The Plan 9 compilers were developed to try out new ideas, deviating from the ANSI standard by both omitting from, and extending to, the standard [6]. Several features have been added over the years, and the compilers today have partial support for C99 [7].

Bitfields, even as supported by the compilers, are not used in the core of Plan 9, but the compilers are also used for the POSIX environment, where bitfields can be found in third party software and imported code in /sys/src such as Ghostscript and audio players.

Patching the source of ported software is the easiest solution if the port can be done once and be done, but if the ported software is evolving, adding the needed support in the compilers can be easier, even if it means to maintain a patch against the compiler.

6. Future work

The goal so far has been to keep the bitfield patch as small as possible, with minimal changes to the compilers. It would be worth rewriting it to be architecture independent, even if that would lead to bigger changes in the compilers.

7. References

[1] Amoson, J. “Porting the Netsurf Web Browser to Plan 9”. *Proceedings of the 9th International Workshop on Plan 9*, Waterloo, Canada, 2023.

[2] Moody, J. Haflinudottir, S. “Portability has outgrown POSIX”. *Proceedings of the 10th International Workshop on Plan 9*, Philadelphia, USA, 2024.

[3] Staal, J. *APExp*. Plan9 (9front) APE with experimental patches. Github source: [https://github.com/staalmannen/APExp]

[4] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1988).

[5] Amoson, J. Patch to support static initialization of bitfields in kencc. Github source: [https://github.com/jamoson/kenccbitfields]

[6] Thompson, Ken. *A new C Compiler*. AT&T Bell Laboratories. [https://doc.cat-v.org/bell_labs/new_c_compilers/new_c_compiler.pdf]

[7] C99 changes vs C89. [https://9p.io/sources/plan9/sys/src/cmd/cc/c99] and /sys/src/cmd/cc/c99 in 9front.

Socarrat for Plan 9: building 9P WORM Devices

Gorka Guardiola Múzquiz, Enrique Soriano-Salvador
{gorka.guardiola,enrique.soriano}@urjc.es
GSyC, Universidad Rey Juan Carlos

May 20, 2025

Abstract

WORM (Write Once Read Many) devices permit data to be written only once. Subsequently, the system can read the data an unlimited number of times. These devices are used for logging purposes and are essential for a wide range of applications. In addition, the data should be tamper-evident: if committed data has been manipulated, the auditor must to be able to detect it. We propose to build local WORM tamper-evident devices with small and inexpensive single-board computers (SBCs). Currently, we are developing Socarrat, a system implemented in Go that follows a Reverse File System approach to provide ext4 and exFAT volumes with WORM properties. This approach is based on inferring the operations that are performed over a file system by analyzing the blocks written to the disk. 9p is an excellent candidate for exporting those WORM devices to other machines.

1. Introduction

A recurring theme in data regulations is the necessity for regulatory-compliant storage to ensure WORM properties that enable guaranteed retention of the data, secure deletion, and compliant migration [1]. Furthermore, the data should be tamper-evident, that is, any modification of the committed writes must be detectable.

The traditional approaches to provide WORM systems use continuous feed printers [2] optical devices [3], content addressed storage [4] or specially designed hardware not available for the general public [5]. There are also numerous distributed approaches to address this problem (e.g. [6]).

In addition to these considerations, the possibility of a cyberattack on the machine storing the data must be taken into account. If the attacker takes control of the system, she should not be able to delete or modify the WORM data of the log. At this point, all solutions based on file system capabilities may fail: If the attacker elevates privileges, she can change the file system configuration or simply modify or delete the files, the data blocks (directly from the block device), the address of the data blocks if it is content-addressed or format the file system.

We have been working on a local device to solve these problems. This device is based on three components:

- Sealfs [7, 8], a file system that provides forward integrity and tamper-evident storage.
- USB OTG, a USB port which can act as a device.
- Socarrat, a Go userspace program that provides the WORM garantieses to log files. Socarrat is based on a novel if somewhat contorted approach: The **Reverse File System**. This approach consists of analyzing the blocks written to the storage device to infer the file system operations executed at the upper layers. Socarrat only takes into account write operations at the end of the corresponding files, ignoring any other operation. This is suitable to keep genuine copies of *log files*.

We are emulating a WORM by using software, so it is important to state upfront what guarantees our system provides. Provided the integrity of the USB link (it can only be used as a mass storage device), the guarantees Socarrat provides, are formalized by what we have named the The Continuous Printer Model (CPM):

1. Once committed, data cannot be deleted/rewritten.
2. Liveness (now and then we commit something while the system works).
3. We do not guarantee that spurious or bad data is not committed in the future, or that the system cannot be stopped from *printing* by breaking it, but guarantee 1 is always preserved: What is *printed* cannot be *unprinted*.

Currently, we are using Raspberry Pi 4 computers to build the prototypes.

2. Usage scenario

Once the three components are present, the following scenario would work:

There are three distinct actors in the scenario, the auditor, Alice and possibly a malicious third party, Malice. Malice is an external intruder trying to manipulate the log in any way possible. Malice has APT capabilities, but is hampered by the need to conduct the attack over the network.

The auditor initializes a USB black box[1] with an specific tool. Then, once the device is properly configured, transfers it to the user, Alice.

Alice connects the USB device to her server. The operating system of the server detects an USB mass storage device formatted as an ext4 or exFAT file system. Then, this drive is mounted in the system. In the mount point, there is a file named `log`[2]. The applications running in Alice's machine are able to use this volume as a normal one, by performing the traditional system calls to work with the files (`open`, `write`, `read`, `close`, etc.). The only difference is that, when the applications access to `log` file, only read operations and append-only write operations

[1]Socarrat and SealFS running on a SOC like the Raspberry Pi with an SSD drive connected, in a box.
[2]There can be many files, we choose log as an example

are effective; the rest of write operations over the `log` file (and its metadata) are ignored within the USB mass storage device.

After normal operation, the auditor can then extract the `log` file from the device, along with an additional file that authenticates all the entries added to the log during this period. Using a diagnostic tool, the auditor can verify that the log data corresponds to the entries made during the logging period, ensuring that no records have been deleted or tampered with.

Alice and the auditor are also adversaries. Alice may try to delete or manipulate the logs, and has physical access to the device. She is hampered by the possibility of getting noticed doing so by the auditor.

3. Threat model and mitigations

Against each actor we have a different threat model and mitigation strategy:

Malice has APT capabilities and may completely compromise Alice's computer over the network. Alice has physical access o the device, and may compromise both devices, but faces repercussions if found manipulating the device.

Against Malice, we have as first line of defense the USB link as attack surface. If it does not fall (i.e. the integrity of the USB link is preserved and it only provides the mass storage device within specifications), we provide CPM properties. Data already committed cannot be rewritten or deleted in any way.

Against Alice, or against Malice if the USB link is broken, we provide the Forward Integrity Model. Data which was already committed cannot be manipulated without detection.

As it is now, in Plan 9 only the CPM properties are provided, as SealFS is not ported to Plan 9 and only through a 9P connection (not a USB link).

4. Overview of the architecture of the system

An overview of the architecture of the system can be see in figure 1.

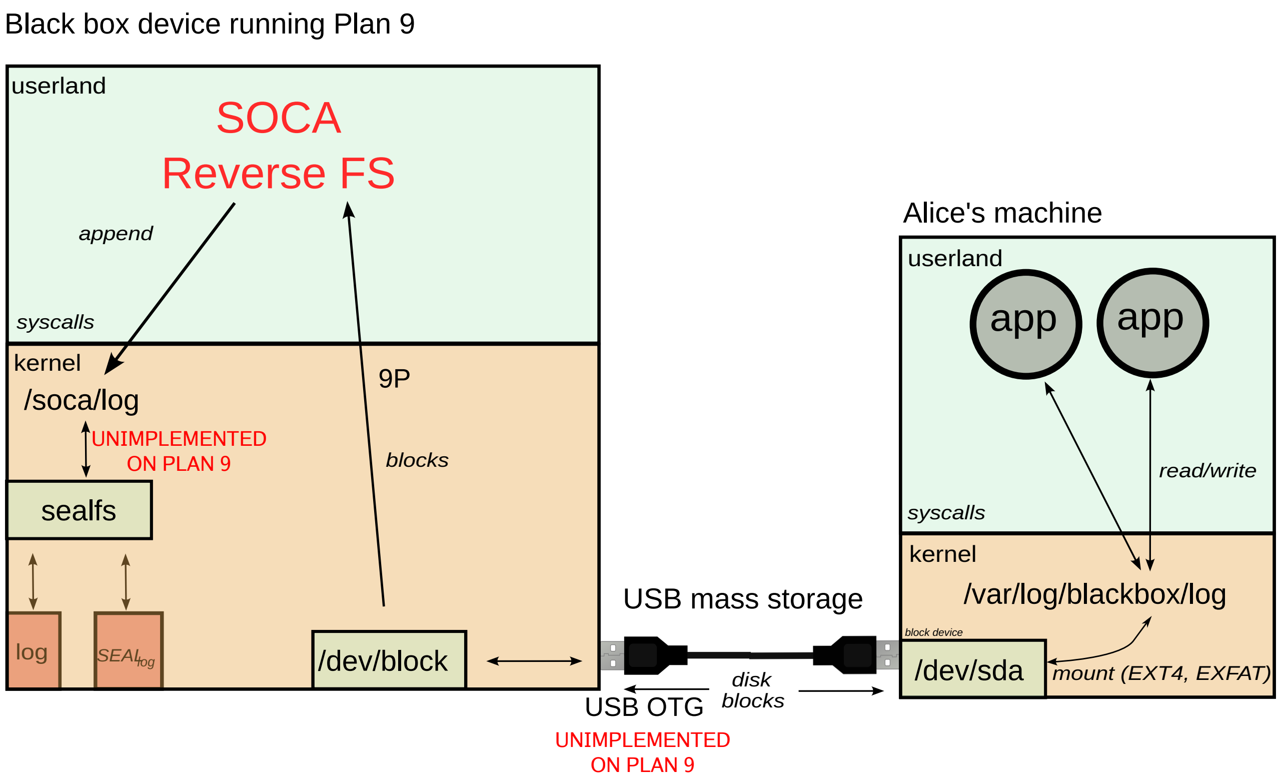


Figure 1: Simplified architecture of the system.

Socarrat (`soca`) is a program that watches a block device and operations on it (at the block level) and infers a subset (limited by the information available and possibly aggregated) of the operations updating the filesystem at a file/directory level of abstraction, as seen on figure 2. In our system, we use the inferred operations to generate a log in a separate filesystem controlled by SealFS [8], which will provide forward integrity.

Note that the USB link or, more precisely the block device interface, which we extend to the other machine via the USB mass storage device, acts as a choke point: the set of operations on it is limited. This enables the *reverse filesystem* to act as a kind of data diode by only honoring whatever operations it wants (writes at the end of the file) and ignoring the rest.

Note that the idea of a reverse filesystem is general and not limited to our application. We are using it to provide a WORM, but this approach can be used for OS fingerprinting (i.e. detecting the OS of the other machine by analyzing patterns in the block usage), debugging file systems, fuzzing, etc.

One of the benefits of the reverse filesystem approach is its portability. The machine where the device running the reverse filesystem is plugged in does not have to know anything about it. As long as the filesystem being reversed is supported, no special software needs to be run on it.

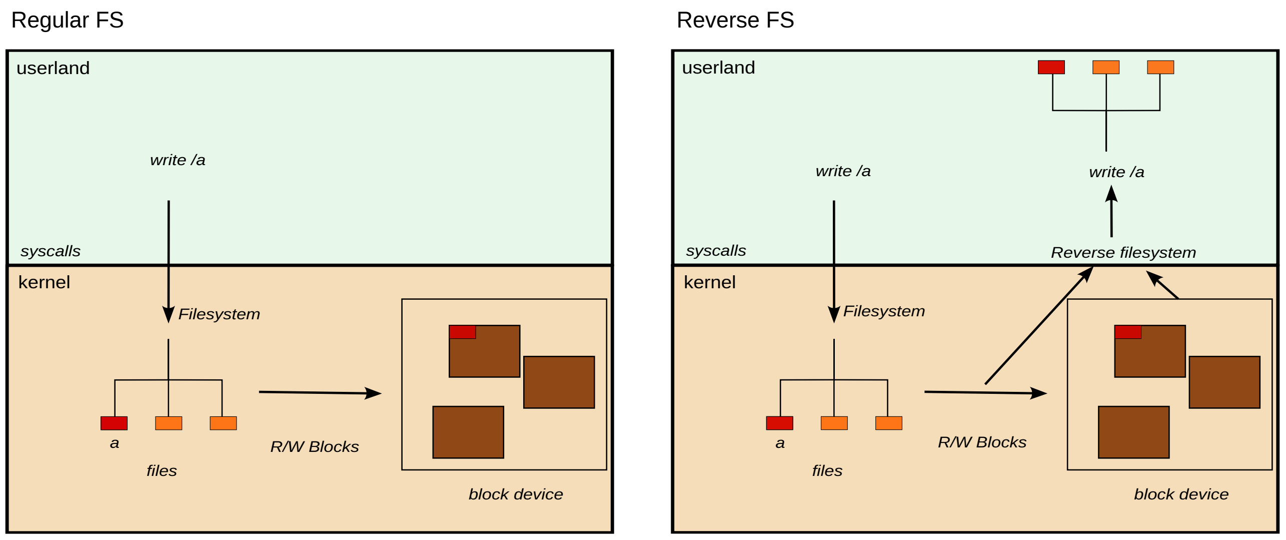


Figure 2: Reverse filesystem: recover (some) write operations from block operations.

5. Plan 9 port

While we built all the components originally for Linux, we have ported Socarrat to Plan 9. This has been a very simple endeavour due to the fact that it is written in Go with minimal external dependencies.

The original version for Linux exported the device using the NBD protocol [9]. This protocol is designed to serve a block device from userspace. The same can be done in a simpler and more general way with 9P. While NBD has several extensions and variants for special cases, 9P has several advantages over it.

First, 9P is a general purpose protocol not specially designed to export block devices. While this may seem like a weakness, it is not. As a consequence of it being general purpose, it has libraries to support it which have been tested thoroughly because they are needed for other uses.

Also, whenever a special operation has to be added, new operations on the `ctl` file, can be added. Therefore, extensibility is easier. It is also easier to model the behavior after it is implemented (for example, who can do which operations can be controlled by the permissions of the files).

All the general purpose debugging tools to spy on the protocol and the expertise dealing with the corner cases, the concurrency model, etc. can be applied from the general knowledge. When working with NBD, the protocol has to be learnt from scratch with all its special corner cases. In a fraction of time of what took to make the NBD version work, we had a 9P version working which was more robust and passed all the same tests.

As it is now, Socarrat posts on `srv` a directory with two files, the device, and a `ctl` file. The `ctl` file can be read and will print the name of the backing image file, and the internal and external files mapped by the *reverse filesystem*. It also accepts a `shutdown` command.

The current version is runnable, but it is missing some of the other components to make the whole system work in Plan 9. The first one is SealFS [7, 8], which provides tamper-evidence following the forward integrity model. The second component is support for USB OTG, present, for example, in the Raspberry Pi. The third is a way to generate an image for an ext4 or exFAT filesystem to serve, but this can be done on another system if needed.

6. Related work

The idea of a reverse filesystem is quite unconventional and there is very little in terms of comparable systems or devices. The closest two we found are related to one another (one was inspired by the other). The first one is vvfat [10]. Vvfat is a block driver served by qemu which serves a virtual VFAT filesystem in which the files represent the files of an underlying directory and follow them when read an written. Inspired on this, there is the nbdkit floppy plugin [11]. These two devices implement a kind of *synthetic reverse filesystem*. It has two main differences:

1. These systems are unconstrained by a liveness property. They only need to guarantee the underlying files are completely synchronized when they are unmounted. Our reverse filesystem needs to extract valid operations continuously in order to work.
2. The second is an implementation difference. A *reverse filesystem* observes the data structures present in a block device image. A *synthetic reverse filesystem* invents these data structures on the fly.

As we stated in the introduction, to implement WORM devices the traditional approaches are using continuous feed printers [2] optical devices [3], content addressed storage [4] or specially designed hardware not available for the general public [5]. There are also numerous distributed approaches to address this problem, leveraging Cloud Computing and blockchain technologies, see as an example [6] (the list is too extensive to be enumerated here).

We can classify all these approaches as:

1. Requiring special expensive, which provide strong guarantees but are cumbersome or unavailable hardware (printers, optical devices).
2. Software based which do not provide any guarantees in the cases of a cyberattack (content-addressed systems).
3. Distributed, which may stop working if the network is unavailable or there is a denial of service attack.

4. Closed systems like [12], for which the inner workings and it is difficult to know what real guarantees they provide.

Our approach requires inexpensive hardware, provides strong security guarantees and works locally without dependence on third parties or distributed systems. Our approach is also specialized for file systems containing log append-only files, not a general purpose file system.

References

[1] R. Sion, "Strong worm," in *2008 The 28th International Conference on Distributed Computing Systems*, pp. 69–76, 2008.

[2] M. Bellare and B. S. Yee, "Forward integrity for secure audit logs," tech. rep., University of California at San Diego, 1997.

[3] S. Quinlan, "A cached worm file system," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1289–1299, 1991.

[4] S. Quinlan, J. McKie, and R. Cox, "Fossil, an archival file server," *World-Wide Web document*, 2003.

[5] J. Leppäniemi, T. Mattila, T. Kololuoma, M. Suhonen, and A. Alastalo, "Roll-to-roll printed resistive worm memory on a flexible substrate," *Nanotechnology*, vol. 23, no. 30, p. 305204, 2012.

[6] M. Li, C. Lal, M. Conti, and D. Hu, "Lechain: A blockchain-based lawful evidence management scheme for digital forensics," *Future Generation Computer Systems*, vol. 115, pp. 406–420, 2021.

[7] E. Soriano-Salvador and G. Guardiola-Múzquiz, "Sealfs: Storage-based tamper-evident logging," *Computers and Security*, vol. 108, p. 102325, 2021.

[8] G. Guardiola-Múzquiz and E. Soriano-Salvador, "Sealfsv2: combining storage-based and ratcheting for tamper-evident logging," *Int. J. Inf. Secur.*, vol. 22, p. 447–466, Dec. 2022.

[9] "Network block device github," 2024.

[10] "Qemu block driver for virtual vfat." [Online; accessed jan-2025].

[11] "Nbd virtual floppy disk from directory." [Online; accessed jan-2025].

[12] "Wormdisk zt storage," 2024.

TEX: from a typographic system to the typography of the system

Thierry Laronde
tlaronde @kergis.com
2025-04-24

1. Stating the problem. Chesterton once said that « *a conservative is someone refusing to correct errors, while a so-called progressive individual insists on adding new ones* », so that he was, himself, neither one nor the other.

Concerning typography by computer, new partial solutions continue to pop up while various solutions have been created previously, some gone to oblivion, some still extent. Amongst the latter, Donald E. Knuth's TEX system is widely used, its layout result, specially concerning mathematics, is recognized as excellent, but its apparently arcane compilation process as well as the insane size of its most known distribution—6 GiB for the full TeXlive—have achieved F.U.D. so that no-one imagine it could be backing the documentation needs of an operating system, being incorporated in its base.

But it remains that there should not be any incompatibility between the own OS needs for its documentation and the users' ones. Instead of accumulating various solutions, each with its shortcomings, it's time to address the few remaining issues and to put the pieces together.

Because there is one feature that puts D. E. Knuth's solution apart: it is *a complete solution*: it allows to define fonts, without which there can be no layout; it computes the layout, giving an interpreter that can be programmed i.e. customized; and a program derived from `METAFONT` can do figures—John Hobby's `MetaPost`.

But if the D.E.K.'s typographic system provides fonts, this is because `METAFONT` is a rasterizer... a RIP. The solution is hence complete.

2. Small and portable. TEX and `METAFONT` have been written using the WEB system of literate programming. WEB is, now, an unfortunate name because of the meaning it has acquired with WWW. But WEB (as well as CWEB, the version for C programming) is just a way to write a program in a given language (Pascal for WEB; C for CWEB), splitting it in small chunks that can be interpolated since they are identified by a short description serving as key, and describing a chunk or a related group of chunks when it is easier to describe their purpose or their implementation, the description, meant for humans, using the full power of TEX. An utility: `tangle(1)` for WEB (resp. `ctangle(1)` for CWEB) simply extract all the programs chunks, interpolating chunks called in other chunks and assembling them in the correct order for the compiler. This program is Pascal—or C, depending on the version used.

In fact, TEX and `METAFONT` have been written less in Pascal than in some sort of Algol: what remains is only the flow control, *à la* C. Few routines, identified, have to be provided by the external implementation.

Once the pseudo-pascal (since it is not anymore any Pascal, but just a very common Pascal) code is obtained, since the Pascal compilers are not ubiquitous and Pascal was not really standardized, this code is translated in raw C, by an utility called in kerTEX `pp2rc(1)`—Pseudo-Pascal to Raw C, that is obviously only used on the matrix, not on the target, since it is used only to obtain the C source for the compilation.

The resulting programs depend only on a libc, since what kerTEX has added is also programmed in standard C, and, until very recently, the programs were even not relying on something outside the standard libc—some new primitives require, now, functions outside standard C. But it remains C, the system dependencies are identified and this compiles on everything.

3. One engine to rule them all. LaTEX has some supplementary requirements, exceeding both the TEX primitives, and the ε-TEX ones.

An extension has been written: PROTE, that offers the TEX compatibility mode; the ε-TEX compatibility mode; and further extensions. In fact, only one engine does the work: PROTE, for whatever: `tex` (D.E.K.'s standard set of macros); `etex` ($\mathcal{NTS}$' standard set of macros); or `latex`.

TEX+ ε-TEX+ extensions = PROTE

4. PROTE as engine for roff too. `Roff(1)` is not a programmable engine for general purpose (text and mathematics) layout. It is a limited engine, with few primitives, doing general (text) layout. All the macros are in fact implemented as specialized pre-processing text filters.

Absolutely nothing could prevent using the PROTE engine as the `roff(1)` engine for the layout, adding only support in DVI for two primitives that are available for `roff(1)` but that are not supported in legacy DVI. For preprocessing (the `man` or `mdoc` macros), a switch can be added to PROTE to recognize the beginning of the line as a prevision character (if what follows is a dot, it is a macro), so that PROTE will be used for preprocessing too.

There is no necessity to change the handy `mdoc` set of macros: TEX (in fact PRoTE) could be programmed in order to be called as `mdoc(1)` or `man(1)` to handle the `roff` standard macros and to do the layout, replacing all the filters by a set of macros à la LaTEX—LaTEX is not an engine: it is only a set of macros running on a compatible engine.

5. **More about fonts**. The Computer Modern series of fonts is provided. With support for virtual fonts, allowing a glyph algebra, one can combine existing glyphs in order to produce new fonts—for example to provide a font with accented letters. So a system can have an already rich set of fonts with these standard fonts.

TEX has been partially rewritten to be 8bits clean. This does mean that it can handle UTF8. Support for more than 256 codepoints can be added without a lot of modifications, in a compatible way: using a font not as a leaf filename but as a directory, with 256 glyphs chunks. Depending on the codepoint computed from UTF8 imput, the font will remain the same, but the DVI will have an instruction to select another 256 glyphs chunk in the directory. If the font is not a directory, then the legacy behavior will happen.

6. **... and more fonts.** But there exist also the *Hershey's fonts*. These fonts have not been designed by Dr Hershey, but have been digitized: a huge set of fonts, including Cyrillic, Greek and even three versions of Japanese ones, are described as line strings, with various resolutions. The result can be quite good.

These fonts were heavily used in CAD systems, early, because such glyphs composed only of linestrings can easily be subjected to an affine transformation—rotation and scaling of text were not available in the early windowing systems; CAD softwares did them using the Hershey's fonts, that are indeed still provided by some of them.

In fact, these fonts could be resurrected, borrowing their metrics (unknown) from the metrics published for standard PostScript fonts.

And the rectification of fonts (only linestrings) could be an intermediary format used in the DVI processing, allowing correct scaling in some defined range, without relying on PostScript and T1 or on some computing intensive (relatively) processing of quadratic curves.

7. **Extending the drawing capabilities**. Using as a template `METAFONT`, John Hobby wrote `MetaPost`. `MetaPost` uses the same language as `METAFONT` but produces basic PostScript as output.

The aim is to extend `METAFONT` to create, let's say, `MetaDRAW`, retaining the fonts capabilities and adding the general drawing capabilities present in MetaPost, but producing DVI as output format.

DVI can be extended. The legacy 256 range opcodes (some still non affected) can be extended to use one free opcode to mean: switch to another page of opcodes. Thus allowing extensions, that could be ignored for security or implementation reasons by the driver handling the DVI+. One extension number will be reserved for "private" use.

DVI is in the same league as PDF. But not everything present in PDF has to be present in DVI. And things can be done differently and remain unencumbered. But some things can certainly be added.

8. `METAFONT` **is a RIP**. All the solutions, today, require finally a PostScript interpreter or at the very least a PDF library able to rasterize the drawing primitives. But `METAFONT` is a RIP: it is its function to transform the drawing (of fonts) in a matrix of pixels, with an excellent result.

Instead of rendering only the fonts, at a size and for a resolution fixed, and having the DVI format only indicating where to put these predefined images on a page, the DVI format could be extended to allow more primitives, and `METAFONT` (or `MetaDRAW`) could be used as the RIP, rasterizing in post-processing the DVI.

9. **Rethinking the user interface**. TEX user interface is rudimentary and when using an instance of an engine for interactive use—for example for computing numbers, with Jean-François Burnol's `xintsession`—, it would be more than welcome to be able to recall a previous command or to edit it.

In fact, since all these command manipulations have to take place before feeding the line to the engine, that ignores control characters, the TEX engine proper is untouched and the control characters can be used to interpret line editing; and this can be done with standard C functions. Furthermore, since the engine has chewed all the available commands (macros), one could also ask the engine to guide the composition of the command line by displaying the expected parameters of a given macro or indicate clearly that one parameter has not the expected type of value.

10. **A solution at reach, with potential powerful possibilities**. Since all the pieces are almost already there, an OS could provide as its base a small powerful layout and drawing engine, accommodating not only the OS own documentation needs, but the user needs as well.

But with a DVI format extended; a layout engine, including for mathematics—TEX—; and a RIP, all the pieces could be there to build GUIs. If, for example, TEX (or PROTE) could be programmed to absorb HTML macros, an HTML browser could be built with a few lines of code.

11. The missing manual. Donald E Knuth has written what is in some sense the definitive description of his typographic system. But, in fact, there is one missing manual: how all these pieces fit together. Writing this manual, that should be considerably shorter than any of the five volumes of D.E.K.' series, is also a task that remains to be done.

12. Present status of the work. There is indeed some work to do, but a major part of the work has already been done. For the remaining part, there are things that can be started now because they will be needed whatever choice is finally made to achieve the goal, and things that are not fixed yet because they do depend on the integration of the solution in an Operating System.

What has been done and is already available anywhere, including in Plan9:

- Extracting the needle from the haystack: kerTEX provides the core (the "kernel") of the solution, without depending on anything. The size is quite reasonable since the initial installation can be reduced to not more, in size, than the current `roff` incarnations, while providing more: fonts and rasterizing;
- Depending only on a libc (neglecting some supplementary primitives needed by LaTEX—that have a default dummy implementation if needed routines are not provided by the libc—it depends even only on standard C, and not even on POSIX);
- Allowing extensions, added to the core: there is a packaging system, that has even the property to allow "live" updating since what is called "recipes" in kerTEX can work with updated upstream releases as long as the way things are unpacked is unchanged—the contents can change, as long as the entry point to generate the files to install is unchanged;
- Extending TEX to support every set of macros: this is done with PROTE, allowing full compatibility with D. E. Knuth's plain, ε-TEX and LaTEX.

What is currently worked on, because it will be needed whatever the final choices:

- Writing a DVI driver to handle DVI processing with the PostScript output, taken from `dvips(1)`, being only one of the output formats;
- Adding the generation of PDF from DVI, since it has less requirements for post-processing (all the programming capabilities of PostScript are useless, since the programmability, the flows of control and so on are available at the interpreters level: TEX, `METAFONT` or `MetaPost`);
- Extracting the rasterizing capabilities from `METAFONT` in a dedicated library;
- Resurrecting the Hershey's fonts;
- Gathering all the notes I have about the TEX system and digest them in a well organized, easily understandable small document, providing the missing manual.

And finally what needs more thinking in order to not waste time implementing things that will be dropped because they do not fit:

- Do I need to allow processing of a prevision character in TEX, in order to drop all the text preprocessing done with external utilities in the `roff` system and let the TEX engine (interpreter) do this work, recognizing a command starting at the very beginning of a line (a leading dot)?
- What commands to add to the DVI format in order, perhaps, to allow using DVI for a Graphical User Interface? (There is prior art in this area: X11 had a PostScript version; Windows has/had a GDI for drawing, whether a screen or a static image.)
- What choices to make for the input User Interface (line oriented)? From the current OSes User Interface in this area, there are mainly things to drop. But what to retain?

Modern Unicode Requires Modern Solutions

Jacob Moody

ABSTRACT

Plan 9 has a history with Unicode, the authors of which were directly involved with the creation and first system-wide implementation of UTF-8. This should come as no surprise with the focus on plaintext that permeated UNIX, and how that focus had carried over into Plan 9. However, the world around Unicode has changed much since its early versions, becoming more complex and detail oriented as the version number climbs higher. While Plan 9 from Bell Labs was never updated to any major version of unicode beyond the first, 9front has made incremental progress over the years in adopting newer standards. We aim to evaluate the current state of the union with Unicode support, and different natural languages at large, within the current 9front system, both where the system is at now and where we would like it to go.

Background

Computers do not handle abstract notions such as natural language particularly well, so we rely on standards for representing natural language as a sequence of numbers. The mapping of numbers to natural language characters is known as a character encoding. While there have been many character encodings invented and used over the years, the one that has seen the most widespread adoption now is Unicode. Unicode is unique among other character encodings before it in the fact that it plans to have a single encoding for all natural languages, done in such a way that a single document could contain language samples from all natural languages without out of band code page switching or the like (as was common before unicode).

With Unicode's goal of serving every natural language and the vast differences between natural languages there comes a need for careful definition of terms that the standard uses[Glossary]. To ensure some clarity let us get some of these definitions out in front:

Abstract Character	A unit of information used for the organization, control, or representation of textual data.
Code Point	Any value within the Unicode codespace
Encoded Character	A codepoint which has been mapped to an abstract character
Grapheme	A minimally distinctive unit of writing in the context of a particular writing system
Glyph	The actual, concrete image of a glyph representation having been rasterized or otherwise imaged onto some display surface.

With that being said, let us define how these terms relate. Broadly abstract characters are information that we wish to encode in the character encoding, this is everything from graphemes, to grammar markings, to components of a letter, to informational control characters. Unicode gives us a large bit space to work with, so a codepoint is a single index into that bitspace, the standard then gives us the mapping for relating these codepoints to abstract characters. A large majority (but not all) of encoded characters represent graphemes. Graphemes are just one possible category that an encoded character can be within. Unicode only really defines these encoded characters, the rendering and presentation of these encoded characters is the responsibility of the rendering layer who takes encoded characters as input and outputs glyph images.

The code space of Unicode is currently 21 bits, and with computing hardware mostly rounding that up to 32 bits, that can be a non-trivial amount of storage for large amounts of text. For this the Unicode standard also defines a few compact encoding schemes, the most popular of which is `UTF-8`, encoding the Unicode bit space within a variable number of bytes up to four.

Unicode also provides multiple codepoints (or a sequence of codepoints) for representing the same abstract character. A large chunk of these characters are those which have been identified as being composed out of two or more subcomponents. The most common example of this are characters that have diacritical markings, which can either be represented as one codepoint for the "precomposed" variant or as a series of codepoints with the base codepoint first followed by codepoints which denote individual diacritical markings.

Considering that there are multiple different representations of the same abstract character, there is a method to normalize a series of codepoints into a consistent representation for semantic comparisons. The Unicode standard[UAX#15] does define a process for conducting this normalization, and provides 4 different forms to normalize to:

NFC	Precomposed codepoints
NFD	Decomposed codepoints
NFKC	Precomposed compatibility codepoints
NFKD	Decomposed compatibility codepoints

The compatibility forms here do additional conversion of special formatted character sets such as codepoints for specific fonts, rotated variants, and so on into their base counterparts. This operation loses some information regarding the original codepoints thus is separated from typical normalization. There are some additional rules that are part of normalization so as to enable checking the semantic equivalence between any two sequence of codepoints.

Existing Unicode in Plan 9

Plan 9 was an early adopter of Unicode, but never reached full compliance for any standard version of Unicode during its history. Pike et al put in a lot of work for handling UTF-8 consistently within the system[Pike93] and a majority of the system was updated to handle codepoints. Plan 9's initial support could be defined succinctly as supporting "21-bit ASCII encoded as `UTF-8`", where each codepoint is assumed to be exactly one abstract character and interfaces broadly assume that mutations to text (ie backspace in a text editor and the like) operate only on single code points as well. The result of this is that the system can only really handle precomposed unicode in any sort of ergonomic way, however Plan 9 is very consistent in this level of support and has been ever since its last official release.

Plan 9 had initially only been written to support Unicode version 1.0 (16-bit code space), so one of the earliest contributions from the community was to update this to the modern 21-bit code space. This work was first done in 9atom[9atom], and was later done in 9front. This includes work to make more of the system Unicode aware that wasn't previously. The version of awk which ships with the system is a notable example. There were also some smaller changes like updates to the input "compose" sequences to allow a wider range of codepoints and allowing users to manually input them by typing out their direct value in hexadecimal. However even with that work there have been portions of 9front that had yet to still be updated past Unicode version 1.0.

Using Tables to Keep Pace

The work being presented in this paper largely started with wanting to update some of the existing interfaces within the system, namely the `isalpharune`(2) set of functions, in order to not only bring them up to date, but also allow their adaption to future versions of the standards. To that end, the Unicode consortium publishes a series of informational databases[UAX#44] in an awk-friendly format. Plan 9 users may be familiar with one of these data files as the original `/lib/unicode` was a cut-down and reformatted version of the Unicode Character Database (UnicodeData.txt). These files are updated and maintained as the Unicode versions change, so the natural course of action seemed to be to read these files as input to generate code that could be then used to implement whatever functions we may need.

To better illustrate the input data, below is an excerpt of UnicodeData.txt for U+2468. Each line of UnicodeData.txt corresponds to a single codepoint, with various properties of said codepoint separated by semicolons. Both the keys and values for each field are documented by the Unicode standard[UAX#44].

```
; look -b 16 2468 /lib/ucd/UnicodeData.txt
2468;CIRCLED DIGIT NINE;No;0;ON;<circle> 0039;;9;9;N;;;;;
```

The parsing of these text input files is largely unremarkable, we read in one line at a time and denote properties and conversions as we move ourselves along. The challenge comes in picking and generating a reasonable data structure for the function implementations to use, balancing both the size of the data structure in memory along with the runtime complexity for walking the data structure. Plan 9 binaries are currently quite small in comparison to the hardware available, so we had decided to err on the side of increasing the binaries size in exchange for better performance. Because of this we decided to implement direct lookup tables, using codepoints as an index and having an O(1) lookup time for the vast majority of keys.

With that said, multiple arrays each with 2^{21} entries is quite untenable even given the potential performance savings.

To mitigate this we can use properties of the distribution of abstract characters over the Unicode code space to our advantage. Unicode code space is distributed up into chunks, with natural language roughly being contained within their own contiguous range of codepoints. This means that for many of the properties we care about there are hotspots and coldspots throughout the code space. As an example, you could consider the `toupperrune()` function, which largely is only applicable to natural languages which feature distinct letter cases. Many eastern Asian languages have no concept of casing so those regions of the Unicode code space don't need defined values. This makes the lookup table quite sparse, and therefore a good candidate for some method of compression. This is not unique to letter casing, and in fact all of the lookup tables for `isalpharune(2)` end up being sparse tables.

To compress the lookup table, we use a data structure that [Gillam02] referred to as a compact array. We start by breaking up the lookup table into equally sized segments, then walk those segments and find any overlap (full or partial) between them. We can then remove any of the overlapped entries within the table. In order to access these overlapped segments we then need a higher level lookup table which points into these segments of the lower table. In order to use a single key (a codepoint in our case) for indexing in to both tables, we split up the bits within the key into two different portions, the top bits indexing in to the top table, the result of which is added to the bottom bits in order to index in to the bottom table.

Given how large the code space is for Unicode, it becomes beneficial to do not just one layer of compacting, but two, where the higher lookup table is compacted with the same method, resulting in a new top level lookup table. This splits up the codepoint into three parts, the result of the top level table being used to index the middle table which is then used to index the bottom table which contains our desired value. The following code illustrates an example of this lookup method:

```
#define upperindex1(x) (((x)>>(4+7))&0x3FF)
#define upperindex2(x) (((x)>>4)&0x7F)
#define upperoffset(x) ((x)&0xF)
#define upperlkup(x) (_upperdata[_upperidx2[_upperidx1[upperindex1(x)]
        + upperindex2(x)] + upperoffset(x)])
```

In this example, the top ten bits of the codepoint index the top level index, the result of which is combined with the next 7 bits of the codepoint, with that result of that being added to the remaining 4 bits of the codepoint. Which amount of bits to use for each section is not exactly clear, so we brute forced the smallest table size through testing all possible values.

Of course in order for this to be effective we need to ensure that the values of our lookup table are sparse. For simple property values this is typically already the case, and for functions like `toupperrune()` we can instead store a delta between the current rune and the desired rune, which gives us nice runs of zero values for anything without a defined transformation. Using this method we've gotten the following results for the `isalpharune(2)` set of functions:

Table	Size (bytes)
toupper	12436
tolower	11754
totitle	12180
is*rune	24184
total	60554

The `is*rune()` table here merges all of the different categories into one table as a bit field, one bit per possible category, so the result is less sparse than the other tables. We find the expense of 60KB to be well worth the benefits gained from the ease of upgrade, and constant time lookup speed.

Keeping Composure

Recall that normalization is the process of transforming a series of Unicode codepoints into a canonical variant for a given target format. In order to do this, we'll need tables that map individual codepoints to their transformed values for each format, their composed or decomposed variants. We can use tables largely similar to the ones described above, with some extras where needed to handle more specific edge cases.

For decomposition our lookup key is the same as before, the codepoint of the rune we wish to decompose. However the value of the table gets a bit more complex. The most straightforward approach would be to combine the two new runes as one value, but this is fairly painful for the table size. We'd need a 42-bit value, which would have to be rounded up to 64. Not only are we wasting over a third of the storage space due to the rounding, but using 64-bit values would be quite onerous for our 32-bit systems. As it turns out however, there is a heavy bias for the new decomposed runes to reside within the lower 16 bit code space. In the latest version of Unicode at the time of writing (16.0) there are only 158 runes that result in decompositions above the 16-bit code space. It becomes easiest then to have our compact array use 32-bit values (two 16-bit codepoints), and then have an exceptions table which stores those 158 runes. The base table values for these exception runes are instead given within a small portion of the Unicode private space (0xEEEE on up), with the difference from the start of the exception being used to index the list of exceptions. This allows us to maintain our constant time lookups, even when accessing members of the exception range.

Recomposition required a different approach with similar constraints as the decomposition. Instead of packing two runes into the output of the table we instead need to pack two runes into the input of our lookup, and produce one rune as output. The bias towards the 16-bit code space remains, so we can limit our main lookup table to taking two 16-bit runes as input and outputting a single 16-bit rune. For exceptions, (where either the input runes or output runes exceed 16 bits) we use a similar exception table. However in this case we have to search this linearly because otherwise we would need to increase the table size to accomodate the larger lookup keys. The differences from decomposition make it much harder to make use of our existing lookup table, so we instead opted to implement a hash table.

A hash table is going to have a worse average lookup time compared to our compact arrays (assuming moderate collision), but stays within the same ballpark of performance and size as our compact array. Because we're generating this table at build time instead of at run time, we have to find a way of embedding the entire structure in a contiguous block of memory. This is easy to accomplish for our first result of the hash function, where we have one large chunk of memory: we hash our input key then modulo it by the size of our initial memory buffer. For collisions however we need to find some way of chaining the results in a way that can be searched until a key matches. For this we create another array that is appended onto the first. The main entry points are chained to the collisions by embedding a numeric offset from the start of the collision space. This results in a hash table entry that looks like the following:

```
struct KV {
        uint key;
        ushort val;
        ushort next;
};
```

In the interest of saving as much space as possible for the generated array, we can compact this struct into a series of paired integers, the first being the merger of the two 16-bit codepoints, and the second being the merger of the value and optionally the offset to the collision space.

As with choosing the constants for compact arrays, we found an optimal size for the original hash array versus the collision array largely through brute force. Given the Unicode standard at the time of writing, we are using 512 initial buckets, which forces 555 entries into the collision array. With a hash function that provides a reasonable equal distribution, this should average to roughly only one collision per entry. Using these two data structures results in the following generated table sizes:

Table	Size (bytes)
decomp	20108
recomp	8536

Normalizing Theory

With composition handled we can then start to tackle normalizing a string to one form or the other. The implementation of Unicode normalization can be quite finicky and there are plenty of considerations for how the input must be fed to result in conformant output. However before we can get into those, we need some additional tables that will be used in the normalization process.

The first additional table we need is one that stores the *Combining Character Class* `(ccc)` of a codepoint, this is, the value which identifies the attaching property of the character. A value of non-zero signifies that the codepoint is an attacher, the specific value denoting the group of attachers that it belongs to. Typically only one attacher out of each group may be considered for attachment to a base codepoint. The ccc value is also used to ensure the ordering of attaching codepoints is consistent in `NFD` formatted strings[UAX#15]. The other value we need is the Quick Check (qc) value of a codepoint which is used to determine where a normalization context ends in certain situations. We'll elaborate on this further in the following section. Our existing compact array structure works well enough for these lookup tables as is, resulting in the following sizes:

Table	Size (bytes)
ccc	9738
qc	7878

The abstract logic for normalization is described in [UAX#15] and is as follows:

```
1. Decompose every rune possible in the string until it
   can no longer be decomposed
2. For codepoints without a ccc of zero, conduct a
   stable sort of them based on their ccc value
3. If we are normalizing to NFC, recursively recompose code
   points in the string until there is nothing more to recompose
```

The main challenge for normalization then is designing an algorithm that can do all of this on a single pass of the string. This is possible but requires careful consideration of what the context is for a given chunk of the string. In regards to this, there are two abstract cases of context that we need to consider. The first is the more traditional case of attaching codepoints. In such cases, the context is the base codepoint (ccc of zero) and all following code points which do not have a ccc of zero. The other context is more fuzzy and applies to natural languages which have letters composed of divisible pieces that are not described as attachers, in other words, cases where we have to recompose two runes which both have a ccc of zero. This happens with Hangul and Devangari scripts where single letters can be composed out of multiple other base runes (Jamo for Hangul and vowel markings for Devangari).

We reasoned about this context by implementing a stack. We scan through the rune and anything within one context is pushed on to the stack. Then when we do recomposition we are bound only by what our stack contains. With this reasoning the normalization process then becomes:

```
1. Take one rune from the string and push it onto our stack
2. If the next character has a ccc not equal to zero,
   push it on the stack. Repeat until we find a rune with a
   ccc of zero.
3. Sort stack by the ccc value of the codepoints
4. If we’re normalizing to NFD, we can output this stack and
   go back to step 1
```

Here we split into one of two recomposition strategies based on which type of context we have. Strategy A is used for sequences of codepoints which have attachers (ccc of non-zero), and strategy B for sequences of composable base runes (ccc of zero).

```
ccc of non zero/Strategy A
1.  Use the base rune as the left hand side input to our
    normalization
2.  Walk through each other codepoint in the string and attempt
    to use it as the right hand side.
2a. If a match is found, replace the base rune with the new result
    and remove the right hand size rune from the string.
2b. If a match isn’t found, skip through all other codepoints
    in the string that have the same ccc as the failed match.


ccc of zero/Strategy B
1.  Continue to pull runes off of the input string until we reach
    a boundary, as described by the Quick Check value.
2.  Walk through each codepoint in the string, attempting to
    compose it with the codepoint immediately to the right of it.
2a. If you find a match, replace the left hand side codepoint,
    remove the right hand side codepoint from the string and
    go back to step 1.
2b. If you don’t find a match, continue through the string with
    the new left hand side being the current right hand side.
3.  Go back to step 1 if any recomposition was found until we make a
    pass without any recompositions.
```

After either of these two steps the resulting stack can be flushed out as normalized output. Interestingly the stipulation to bypass other runes which have the same ccc (step 2b in strategy A) is stipulated by the standard[UAX#15] and required for a conformant implementation.

There does exist an edge case in normalization, as the standard itself puts no limit on the amount of attaching codepoints may be given to a base codepoint. This can be seen in what is typically referred to as "zalgo text", where a single base codepoint is given a large amount of attachers to produce a "glitchy" appearence. While there is no natural language which relies on a large number of attaching codepoints, it remains an edge case our algorithm must come to terms with. In order to perform this normalization within a bounded memory constraint we need to pick some reasonable maximum amount of attaching codepoints that we plan to support. Unicode defined a subset of normalized text referred to as "Stream-Safe text"[UAX#15] which dictates that no more than 30 attaching code points may be given to a base codepoint, and for sequences larger than that, attaching codepoints can be broken up with `U+034F` (Combining Grapheme Joiner). We implement this recommendation, which means our algorithm will insert a CGJ into a text stream if we encounter large enough runs. This is required if we want to output properly normalized text, as without it the ccc context would grow to a length that would be untenable to sort. The CGJ acts as a new base, and codepoints which attach to it can be sorted independently of those on the base codepoint. The rendering layer should be able to interpret the attaching codepoints on the CGJ as part of the previous base code point.

Normalization Interface

With the general algorithm for normalization figured out we need to decide on an interface for exposing this to other programs. Due to the nature of normalization this can be somewhat complicated to do well. As illustrated earlier, context expansion in both types of context continues until we find the beginning of the next context. In strategy A it is until we find a ccc of zero and in strategy B it is until we find qc boundary. This means that for most normalization we need to know when an input ends in order to flush the last context, since there is no next context to act as our terminator. This is relatively easy if we're operating on the entire string at once: we can walk until we reach the end of the input string and use that as a sign to flush whatever remaining context we may be holding on to. If we're dealing with a stream of input, from a file/pipe/network connection, then we may wish to normalize one chunk at a time. We then need to maintain the remaining context of one chunk into the beginning of the next chunk. Other implementations handle this issue by implementing a concatenation function, which carefully conjoins one normalized string to the other with only contextualizing the edges. We decided to instead implement only a streaming interface, which can be used for both concatenation and streaming input. While pushing two strings that had been independently normalized back through the streaming interface is not as efficient as only looking at the boundaries, we think that the normalization implementation itself should be sufficiently fast to have a negligible performance impact in a majority of cases.

The following streaming interface is what we had settled on:

```
typedef struct Norm Norm;
struct Norm {
        ... /* private */
};
extern void norminit(Norm*, int comp, void *ctx, long (*get)(void*));
extern long normpull(Norm*, Rune *dst, long sz, int flush);
```

In addition we provide the following wrapper functions which target normalization for smaller fixed set inputs:

```
extern long runecomp(Rune *dst, long ndst, Rune *src, long nsrc);
extern long runedecomp(Rune *dst, long ndst, Rune *src, long nsrc);
extern long utfcomp(char *dst, long ndst, char *src, long nsrc);
extern long utfdecomp(char *dst, long ndst, char *src, long nsrc);
```

With the streaming interface: the `flush` flag in `normpull`() specifies whether we should preserve the remaining context once we reach an EOF from a `get`() function. This allows callers to normalize fixed chunks at a time, then call with the flush argument set to non-zero when there is no more input to give. The `ctx` is a caller supplied pointer which is also passed forward on to the `get`() function. If `Comp` is non-zero the string is composed, meaning the output is `NFC` instead of `NFD`.

The Unicode standard publishes an extensive set of tests[UAX#44] to ensure that normalization implementations deal with all of the possible edge cases which show up within expected realistic input. We verified our implementation against this series of tests, and added support for normalized strings to `tcs`(1).

Preparation and Future Work

While normalization itself is not within the crucial path to generating a better conforming Unicode interface, the necessary infrastructure for implementing normalization provides the foundation for implementing more comformant interfaces. For this paper we want to focus on what we think is the next step, which is a more conformant interface for Plan 9 programs which use `frame(2).` This would be `rio`(1), `acme`(1), and `samterm`(1).

As implemented today, these programs assume that each codepoint is its own abstract character and extra attaching codepoints are treated as individual, separate characters. Instead of attaching combining characters to their base codepoint rio, sam and friends will render them on their own, resulting in a dangling diacritic in the best cases and a stray PJW in the worst. This results in decomposed text being largely unusable within these interfaces. If we would like these programs to handle such text better, we need interfaces that allow it to find logical breaks within the text. Unicode defines[UAX#29] two types of breaks, grapheme breaks and word breaks.

Grapheme breaks are boundaries between codepoints in which a user may consider it to be one logical abstract character. The codepoints between these breaks (referred to as a grapheme cluster) should be considered as one abstract character by editors. For example, an editor may choose to interpret a backspace to be the deletion of codepoints up until the previous grapheme break. Likewise an editor should not put a line break within a grapheme cluster. Another consideration may be whether `sam`(1)'s structured regular expression should be changed to interpret grapheme clusters as one logical character or as independent codepoints.

Word breaks are similar to grapheme breaks, but for exclusively finding where in a series of codepoints an interface may choose to place a line break. This prevents cases like placing a line break halfway through a series of numerics, or in general places which might confuse a native speaker when reading the text. Of course it is not always possible to abide by these rules, as the width that an editor is working with may be shorter than the full context of a word break (consider a file full of 1000 1's), so at best it serves as an optimistic placement. While nice to have for text editors, we believe that a more important application lies in typesetting documents.

The algorithms for finding these breaks are less involved than normalization, and we were able to again use our compact arrays to store the required lookup information for codepoints. As part of this paper we implement the following set of functions for finding grapheme and word breaks within an input string:

```
Rune*  runegbreak(Rune *s);
Rune*  runewbreak(Rune *s);
char*  utfgbreak(char *s);
char*  utfwbreak(char *s);
```

The return values of these functions point to the beginning of the next context.

Of course these functions are only the start of implementing proper support in our editors and interfaces. Future work may look to reimplement `frame`(2) to make use of these functions for implementing editors that work well in languages other than English. We have so far also largely ignored the rendering portions of these interfaces. It is another important step required for conformance but the rules for rendering extends beyond what is specified in Unicode and is best saved for a future paper which explores the full complexity of that problem space.

Acknowledgements

Special thanks to Ori Bernstein, cinap_lenrek, and the rest of the 9front community for their support in designing and thinking about this problem space. The work as it is presented would have not been possible without their advice and review.

References

[Gillam02] Richard Gillam, ‘‘Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard’’, January, 2002

[Glossary] Unicode Inc, Glossary of Unicode Terms, https://www.unicode.org/glossary/,

[Pike93] Rob Pike, Ken Thompson, ‘‘Hello World or Καλημέρα κόσμε or こんにちは世界’’, January, 1993

[UAX#15] Unicode Inc, Unicode Normalization Forms, https://www.unicode.org/reports/tr15/, Version 16.0.0, August, 2024

[UAX#44] Unicode Inc, Unicode Character Database, https://www.unicode.org/reports/tr44/, Version 16.0.0, August, 2024

[UAX#29] Unicode Inc, Unicode Text Segmentation, https://www.unicode.org/reports/tr29/, Version 16.0.0, August, 2024

[9atom] Eric Quanstro, http://www.9atom.org/, https://web.archive.org/web/20201111224911/http://www.9atom.org/, November, 2020

Glenda and Tux

Alyssa Marie
oglonut@proton.me

ABSTRACT

This experiment is an attempt to transport the non-Plan 9 features of the Linux file system over 9P2000.

This is done by using virtual control files within a 9P2000-exported Linux file system in a manner similar to the way Plan 9 sometimes controls irregular devices [Pike], and has its roots in the process file system. [Killian]

1. Introduction

The Linux file system (and those of other systems) come with features that don't map directly onto the 9P protocol, so if two Linux systems communicate using 9P, there's no obvious way for them to communicate those extra features.

An extension of 9P2000 - 9P2000.L - was designed, and implemented in the Linux kernel (as v9fs), as a way to communicate those extra things [Hens]. The drawbacks are that it's an *additional* protocol - it's not expected to replace 9P2000 - and is quite Linux-specific (albeit with perhaps 1 000 000 x as many installations).

While this project is currently about interoperating with Linux, the concept could be used with other systems to provide a transport for their unique features. Where there's overlap they can perhaps share properties and commands. But that's for the future.

2. Approach

I've built a 9P server for Linux which presents some virtual files and directories that give access to extra behaviour not normally accessible via 9P. In other respects its a conventional 9P server.

I've also built an adapter, which runs on the *client* machine, mounts itself as a 9P2000.L server, and implements 9P2000.L by making 9P2000 requests over the network to the 'enhanced' 9P server. A future version of the adapter may use FUSE, perhaps building on 9PFUSE. The adapter attempts to use the virtual directories when asked to do something it can't express with 9P. If that fails, it usually returns an error.

Both of these tools work conventionally with existing 9P servers and clients, and are fully transparent between Linux machines (the virtual directories are not made visible by to Linux by the adapter). Between the adapter and server the file system is a little different but usually not noticeably. A separate adapter could be built to hide the differences for 9P clients that can't tolerate them.

3. The protocol

Properties of Linux files are exposed by the server through virtual control files in a similar way to how some Plan 9 devices present settings. Each file 'foo' in the file system has a corresponding control file ',ctl/foo'. The directory containing 'foo' has its own control file ',ctl/,' (We can't use '.') The virtual directories (e.g. ",ctl") are *unlisted*, in the containing directory but can still be accessed. A control file contains a list of text lines, each consisting of a property name, some white space and a value. e.g.

```
mtimensec 743956295
```

is the nanosecond part of the mtime field. When the property is writeable, then a similar line of text will set the value. For Linux, the properties are currently: dev inode umode nlink uid gid rdev size blksize blocks atime mtime ctime atimensec mtimensec ctimensec name symlink, and any extended attributes with their names prefixed with X to avoid ambiguity, or Y and hex-encoded if not valid utf8.

On request, the control file can also include a description of a colliding locked range (see Getlock below):

```
rdlock <start> <length> <pid> <client>
wrlock <start> <length> <pid> <client>
unlock
```

The control files for directories also allow commands to be written:

```
symlink <name> <target>
link <name> <target>
linki <name> <inumber>
mknod <name> <mode> <major> <minor>
move <name> <oldpath>
movei <name> <inumber>
```

The control files for files allow these commands:

```
rdlock <start> <length> <pid> <client>
wrlock <start> <length> <pid> <client>
unlock <start> <length> <pid> <client>
```

These are all in addition to allowing any settable properties to be set.

9P requests all operate on fids. By walking a fid, you can leave modifiers for a subsequent message to use, thus parametrizing its behaviour. Each of the following modifiers (either separately or combined) will name a virtual directory fid that holds them. Operations on the files within are modified as requested.

- **,ctl** causes control files to replace normal files in the virtual directory.
- **,append** will cause the files to open in append mode
- **,gid**:**<nnn>** will provide a group id for operations that need one
- **,sync** will cause files to open with the O_SYNC option
- **,dsync** will cause files to open with the O_DSYNC option
- **,getlock**:rdlock+<start>+<length>+<pid>+<client> or:
- **,getlock**:wrlock+<start>+<length>+<pid>+<client>
  will cause control file contents to be replaced by the description of a colliding lock, or indicate that it is unlocked with an “unlock” line.
- **,statfs** will cause file system statistics to replace control file contents for this fid
- **,anonymous** is used for O_TMPFILE nameless files, to distinguish it from DMTMP files on Plan 9, which are not anonymous.
- ,**nofollow** causes any symlink encountered to not be followed. Actions will be taken on the link itself rather than its target. This also affects the listing of directory contents. Without this modifier symbolic links are followed by default.
- **,noxattr** suppresses the inclusion of extended attributes in ctl files.

Other modifiers can be added as needed.

So Twalk “foo” followed by Topen will open “foo”, but Twalk “,append” “foo” followed by Topen will open “foo” in O_APPEND mode. (This is different from DMAPPEND files on Plan 9. Linux does its own thing.)

“,append/foo” is a pathname for the file “foo” but opened with append-only behaviour.

4. Name encoding

Linux filenames have a wider character set than Plan 9 ones, excluding only / and NUL, though 9P itself forbids only the NUL byte. Plan 9 filenames also exclude any control characters and DEL. The encoding transforms exactly the bytes numbered 0x01-0x1F, 0x25, 0x2C, 0x7F, into “%01”-”%1F”, “%25”, “%2C” and “%7F”. Only those sequences will be decoded. This means that all sequences of bytes in filenames have a unique and reversible transformation. 0X25 is the percent sign, 0x2C is the comma, 0x7F is DEL. A comma is only a frog when it's the first character.

5. 9P2000.L Translation

While 9P2000.L is not a part of this design, it helps to understand how its messages are translated into this form, not least because the current adapter implementation relies on 9P2000.L to mount itself on the Linux client machine.

- **Tversion**, **Twalk**, **Tread**, **Twrite**, **Tflush**, **Tremove, Tclunk** all pass through essentially unchanged, except that **Twalk** encodes the filenames. And **Tversion** drops to 9P2000.
- **Tauth** and **Tattach** may include the numeric UID in the uname. TBD
- **Lopen** "foo" → **Twalk** ",gid:<nnn>,append,sync,dsync,nofollow" "foo" **Topen** – any properties may be omitted if not required.
- **Lcreate** "foo" → **Twalk** ",gid:<nnn>,append,sync,dsync,trunc,anonymous" "foo" **Tcreate** – any properties may be omitted if not required
- **Mkdir** "foo" → **Twalk** ",gid:<nnn>" "foo" **Tcreate, Tclunk**
- **Link** "foo" <fid> → **Tstat** <fid> , **Twalk** ",ctl" "," **Topen, Twrite** "linki foo <inumber>", **Tclunk**
- **Symlink** "foo" "target" → **Twalk** ",ctl" "," **Topen**, **Twrite** "symlink foo <target>", **Tclunk**
- **Mknod** "foo" <mode> <major> <minor> → **Twalk** ",ctl" "," **Topen, Twrite** "mknod foo <mode> <major> <minor>", **Tclunk**
- **Rename** "foo" <fid> → **Tstat** <fid> , **Twalk** ",ctl" "," **Topen, Twrite** "movei foo <inumber>", **Tclunk**
- **Renameat** <odfid> "oname" <ndfid> "nname" → **Twalk** <odfid> "oname", **Tstat, Twalk** ",ctl" "," **Topen, Twrite** "movei foo <inumber>", **Tclunk**
- **Unlinkat** <dfid> "name" → **Twalk** "name", **Tremove**
- **Statfs** → **Twalk** ",statfs,ctl" "foo", **Topen, Tread, Tclunk**
- **Readlink** → **Twalk** ",nofollow", **Topen, Tread, Tclunk**
- **Getattr** → **Twalk** ",ctl" "name", **Topen, Tread, Tclunk**
- **Setattr** → **Twalk** ",ctl" "name", **Topen, Twrite, Tclunk**
- **Lock** → **Twalk** ",ctl" "name", **Topen, Twrite** "rdlock <start> <length> <pid> <client>"**, Tclunk** – or wrlock, or unlock
- **GetLock** → **Twalk** ",ctl,getlock:rdlock+<start>+<length>+<pid>+<client>" "name", **Topen, Tread, Tclunk** – or wrlock
- **Fsync** → **Twstat** – (changing nothing with **Twstat** is a sync operation already available in 9P)
- **Readdir** → **Tread** – results affected by **Topen** modifiers. With ",nofollow" DT_ values are encoded in existing mode bits: DMAPPEND combined with a 0444 for symlinks. Any of the three read bits can be turned off yielding 8 DT_ values. With the DT_ bits hidden in the directory **Tread**, **Readdir** doesn't have to do any **Tstats** to get the d_type byte.
- **Xattrwalk**, **Xattrcreate** – the adapter synthesises fids, and reads/writes the ,ctl file.

6. Implementation Status

My code is very much a prototype, working just well enough to satisfy me that it could work properly with better engineering.

Working with 9P2000.L has involved some shenanigans because in many cases I need to walk a fid to provide modifiers, but sometimes the fid I have is already a file. You can't walk backward from a file or forward... So the adapter keeps multiple server fids for each 9P2000.L fid: one for the file, one a step back from the file and also keeps the filename within the directory. You can walk forward from a directory, but only if it is not open, so we also have to keep a separate fid if it is open (because someone might GetAttr the open file or directory). That's sometimes three server fids per client. Probably with FUSE I could get by with two fids: one to walk from and one to use when the fid is open. 9P2000 is probably unnecessarily restrictive about Twalks, and I think it's probably an unintended hold-over from old9P's Tclone, but it is what it is.

At time of writing:

The client and server are mostly functionally complete, but it's not difficult to cause segmentation faults at the moment.

- POSIX locking is coded, but not tested.
- I've not done anything about authorization.
- I haven't done anything about blocking devices.
- They are also single-threaded.

7. Examples

On Linux:

```
srvname$ 9pld
```

```
client$ 9l9pd srvname 564 /mnt
```

mounts the server file system on /mnt.

```
term% cat foo
hello
term%
```

You can see the Linux properties of a file “foo” like this:

```
term% cat ,ctl/foo
dev 45831
inode 265868
mode 0100644
nlink 1
uid 1001
gid 1001
rdev 0
size 6
blksize 4096
blocks 8
atime 1739199962
mtime 1739199962
ctime 1739466812
atimensec 460383798
mtimensec 470383712
ctimensec 926076796
name foo
term%
```

To make a symbolic link, do this:

```
term% echo symlink bar ../../foo >,ctl/,
term%
```

Symbolic links are followed by default:

```
term% cat bar
hello
term%
```

To see a symbolic link target, do this:

```
term% cat ,nofollow/bar; echo
../../bin
term%
```

To make an extended attribute, do this:

```
term% echo Xuser.teapot chocolate
>,ctl/foo
term% grep teapot ,ctl/foo
Xuser.teapot chocolate
term%
```

8. References

[Hens] Eric Van Hensbergen, Ron Minnich, Grave Robbers from Outer Space: Using 9P2000 Under Linux, USENIX, 2005.

[Killian] Killian, T.J. "Processes as Files." In USENIX Summer Conference Proceedings. Salt Lake City, Utah, 1984.

[Pike] Rob Pike et al, “Plan9 from Bell Labs”, proc UKUUG Conf, London, UK, 1990

9. See Also

https://github.com/chaos/diod/blob/master/protocol.md

https://www.kernel.org/doc/html/latest/filesystems/9p.html