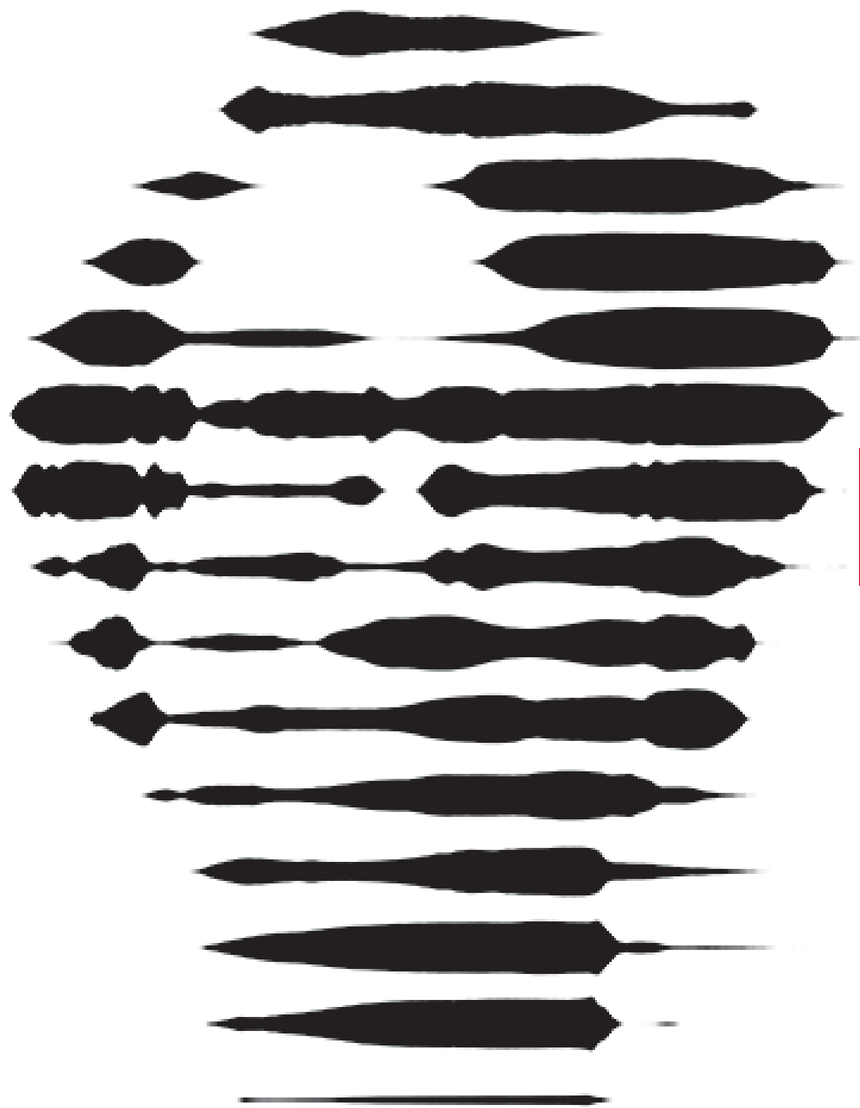


4th International Workshop on Plan9

Styx Caching via Journal Callbacks
V. Srinivas, N.W. Filardo

Complex GUIs in Plan 9
J. Amoson

Using Currying and Process-Private System Calls to Break the One-Microsecond System Call Barrier
R.G. Minnich, J. Floren, J. Mckie

Living Under a Poison Tree
E. Quanstrom

A New Boot Process
I.M.S. Souza

ATA au Naturel and SD Refresh
E. Quanstrom

Plan 9's USB
F.J. Ballesteros

Measuring Kernel Throughput on Blue Gene/P
R.G. Minnich, J. Floren

October 21-23 2009

Papers

Styx Caching via Journal Callbacks 5
N.W. Filardo, V. Srinivas

Building complex GUIs in Plan 9 15
Jonas Amoson

Using Currying and process-private system calls to break the one-microsecond system call barrier 23
R.G. Minnich, J. Floren, J. Mckie

Living under a Poison Tree 33
E. Quanstrom

A New Boot Process for Plan 9 39
I. Souza

SD Refresh 45
E. Quanstrom

ATA au Naturel 51
E. Quanstrom

Plan 9's Universal Serial Bus 53
Francisco J. Ballesteros

Measuring kernel throughput on Blue Gene/P with the Plan 9 research operating system 73
R.G. Minnich, J. Floren, A. Nyrhinen

Works in Progress

Btfs — a Bittorrent Client 85
M. Lonjaret

Dynamic resource configuration and control for an autonomous robotic vehicle 89
S.D. Johnson, B. Himebaugh and A. Kulkarni

Two Enhancements for Native Inferno 93
B.L. Stuart

Ircfs and Wm/irc 97
M. Lukkien

KNX Implementation for Plan 9 101
G.G. Múzquiz, E.S. Salvador, F.J. Ballesteros

A File System for Laptops 105
B.L. Stuart

Levitation Across the River Styx 109
J. Sickel

How To Make a Lumpy RNG 113
M.A. Covington

Mail as (real) files 119
F.J. Ballesteros

Inferno for the Sheevaplug 123
M. Lukkien

Ssh 2 Client 127
M. Lukkien

File Indexing and Searching for Plan 9 131
F.J. Ballesteros

Torrent 139
M. Lukkien

Styx Caching via Journal Callbacks

Venkatesh Srinivas
Nathaniel Wesley Filardo

me@acm.jhu.edu, nwf@cs.jhu.edu

Association for Computing Machinery
Johns Hopkins University
Baltimore, MD

ABSTRACT

Styx is a network protocol used in the Plan 9 and Inferno distributed operating systems. This protocol provides a common language for communication within the above-mentioned system. Styx is a simple client-driven, message-oriented protocol. This protocol performs poorly on high-latency links, independent of bandwidth, and has no provisions for caching or server-initiated notifications. Previous attempts at hiding latency have either required replacing Styx or accepting dramatically weaker coherency guarantees.

This paper introduces Journal Callbacks (JC), a mechanism for server-initiated notifications in client-driven protocols, such as Styx. Journal Callbacks allows for these notifications without modification to the underlying protocol.

We implemented a cache for Styx using Journal Callbacks. We attempted to hide latency by caching server responses; notifications are used for invalidation events. Notably, our cache and notification scheme does not alter the Styx protocol; instead, it runs as a side protocol on top of the existing stream. We present data from several benchmarks, showing our cache reduces effective latency comparably to the Plan 9 cfs cache.

1. Introduction

Styx [1] is a very simple resource abstraction protocol. It describes a set of named hierarchical trees of named objects; leaf entries in the tree are named ‘‘files’’; other entries are termed ‘‘directories’’. All objects store a fixed set of metadata. Additionally, files each provide an optionally seekable single stream of bytes. Some Styx servers support creating or removing subsets of their exported objects.

Styx allows more than one outstanding request through a client-controlled "tag". Responses may be uniquely paired to their requesting message since the server will simply copy the tag back. There is a mechanism for request cancellation by tag, but this facility is rarely used.

A Styx connection uses client-specified integers named "fids" to represent live handles to objects. A fid, naming the root of the server’s hierarchy, is derived from an initial attach message. Fids may be walked to traverse the exported file tree, stat-ed or wstat-ed to read or to write metadata, or Opened for subsequent Read and Write requests. Create operations take a fid naming the parent directory as well as the name

of the object to be created. Once a fid, Open or not, has finished its purpose, it is Clunked and its identifier is safely available for reuse.

Styx uniquely identifies every version of every object (typically "file") on a server by an entity called a QID. QIDs expose some minimal "type" information, a "path" identifier (essentially object identifier), and a "version" field. Typically, versions are incremented whenever a mutation request (i.e. write, wstat, create, or remove) is successful.

Opened fids *track* the current version of any server-side object. That is, if there are two fids, either from one or two clients, naming a given object, and one fid is used for a Write, then a subsequent Read on either fid will reflect the changes made.

Prior work [2] has demonstrated that operations over Styx can be dominated by latency of the link, which indicates that large performance gains may be had by reducing the number of RPCs that cross the wire. There are a number of ways one might go about this:

1. Redefining the protocol to need fewer RPCs,
2. Intercepting client RPCs and answering from cache before they may go over the wire, or
3. Altering the behavior of clients to eliminate superfluous RPCs.

Previous work falls into the first and second categories. This work is also of the second variety though we believe we are the first to investigate and find opportunities of the third class.

2. Related Work

2.1. cfs(4)

cfs(4) is an on-disk cache intended for use by Plan 9 terminals. It copies data from read messages into an on-disk cache. For subsequent read operations, if the data are already present, cfs responds with cached data. Once per open, cfs will stat an object on the server to check for validity of cached contents. cfs does not cache directory contents nor, by extension, does it attempt to hide any latency for walk operations. Every Walk, Stat, and Read on a directory is simply passed through to the server. Open messages become Stat followed by a open when the cached contents are shown to be stale.

cfs by design eliminates the ‘‘tracking’’ feature of fids described above; that is, opened fids passing through a cfs instance will continue to expose whatever data is in cache, not what is present on the server. It is therefore possible, since cfs does not do readahead or whole-file caching, to see a file’s stream in a state that corresponds to no server version and have to re-open the file to resynchronize.

2.2. Octopus’s Op

Op [2] is a revision to the Styx protocol which batches together operations on the wire to minimize the impact of latency. Ofs, the program which does Styx-to-Op intermediation, may optionally act as a cache. When so doing, it assumes data is unaltered for a brief period of time known as "coherency window" before it will act like cfs and check the remote server for validity.

2.3. Network File System

NFS [3] is a protocol in the UNIX world superficially similar to Styx. At least one modern NFSv3 client provides ‘‘close-to-open’’ cache coherency, similar to that found in cfs(4). When a client has a file open, it is assumed that the client’s cache matches the

authoritative copy and that no other client is making changes. When the client closes the file, all still dirty cache contents are written back to the server and the client's kernel issues a GETATTR. If on the next open call, the GETATTR request returns the same value, the cache contents are assumed valid. Concurrent writes, even concurrent appends, are not sensibly supported.

2.4. Andrew File System

The Andrew File System [4] also provides client-side caching. Clients are given time-limited promises of notification should an opened file change. Clients may extend these by actively reregistering their interest with the server. In Coda, an AFS descendent, these callbacks are able to range in granularity from files to entire AFS volumes; see5 . AFS assumes only one concurrent writer and generally writes back to the server only when a file is closed or when the cache overflows; therefore, servers do not call for writeback and there is no inter-client cache coherency.

2.5. Common Internet File System

Microsoft's CIFS [6] supports caching using both "opportunistic" and explicit (byte range) locking strategies. CIFS servers will notify clients of invalidations to open files; we are unclear if this extends to opened, cached files that are not currently open. CIFS allows caches to buffer writes and release them only on server notification of opportunistic lock breaks. Other clients are stalled while the server waits for the owning client to write back. The lock taking operations and notifications are built in to the underlying RPC protocol.

3. Design and Implementation

The next few sections describe the core ideas of JC cfs and then how JC cfs is put together, first the cache controller, then the client-side cache.

3.1. The Design

As mentioned, Styx uses QIDs to uniquely identify every version of every object on the server. The evolution of a Styx file server's state could be described by an append-only log of every QID modified (that is, whose version field changed) or removed. (Such a *journal* need not include creation events; we may assume that the toplevel directory simply exists by offset zero and all subsequent creation events will modify the containing directory.) Each client-side operation can be thought of as having some index into this log; conversely, each offset corresponds to zero or more read operations and exactly one write operaton. If a cache were to read this file and watch for appends, it would know when, subject to network latency, to invalidate cached data and reread from the server.

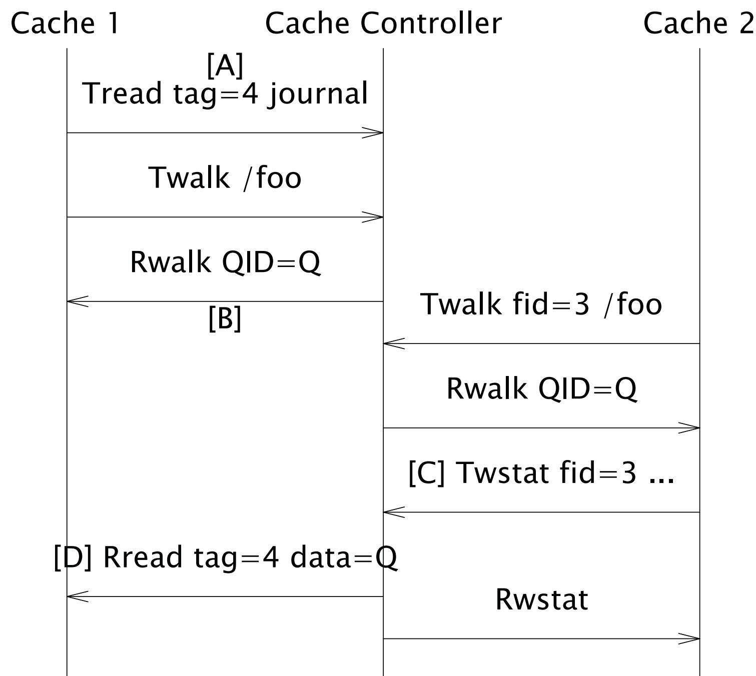
There are four agents in the Journal Callback design: the client, the server, the client-side cache, and the (server-side) cache controller. The client, often the kernel's `devmnt` , and Styx server (e.g. fossil) remain unmodified. The cache and controller act as Styx intermediators: that is, they have two Styx connections and respond to events from each. The cache and cache controller communicate using their own Styx messages over the wire. Additionally, the cache and cache controller are each free to initiate requests of the server. To avoid changing the Styx hierarchy as viewed from the client, we create a parallel hierarchy, using the *aname* feature. This design decision allows us great flexibility going forward.

The cache controller encapsulates all inter-cache information management. It serves to inform one cache when another has successfully carried out a mutation of server state. The cache controller is assumed to sit between all caches and the server. That is, while typically a server is permitted to handle clients directly, in the JC scheme we assume that the server's sole client is the cache controller.

Our cache controller filters the global QID journal to be specific to each connecting cache (which are identified by UUIDs). Every QID reported to the cache is considered cached. Further, it attempts to maintain knowledge of which server data have been seen by the cache even after the cache disconnects; it is possible to indicate to the cache that it has been gone too long and that it must assume that all cached data is out of date.

The cache mediates between a client and the cache controller. It will return cached contents – file, stat, and directory data – quickly when present and believed to be up-to-date. Caches are free to adopt a number of behaviors, including cfs-like behavior or simply blocking client requests, when the journal indicates that they are not synchronized with the server.

An example trace of a cache (1) implicitly registering interest in a file `/foo` and another cache (2) causing the cached contents to become invalid can be seen below. At point A, Cache 1 believes itself to be fully up to date and issues one more read against the journal which blocks. By point B, the cache controller has registered Cache 1's interest – that is, the potential to have cached – the QID Q. The second cache's Twstat operation at point C is forwarded to the server, and if the response is an Rwstat (rather than a Rerror), the cache controller wakes Cache 1 by answering the blocked read (point D) and will forward the success to Cache 2.



For performance and security reasons, one might wish to integrate the cache controller and server. We have not done so largely for ease of implementation and to avoid tethering ourselves to a particular server.

We have not yet implemented a mechanism – such as an append-only write-only file located beside the journals – for caches to notify the controller that a QID has been flushed. Such a mechanism would reduce cache controller memory and unnecessary notifications.

3.1.1. Asynchronous Notifications in Styx

We cannot claim that the core idea, of using a synthetic file to deliver events, is new. From the outset, one of us used the documented behavior of usb(4) audio devices,

> When all values from **audioctl** have been read, a zero-sized buffer is returned (the usual end-of-file indication). A new read will then block until one of the settings changes and then report its new value.

as precedent. We are unaware of any prior implementation of the scheme for caching, however.

3.2. Implementation

As mentioned, both the cache and the controller act as Styx intermediators. Since Styx offers only a single namespace for each of message tags and fids, both of these programs maintain mapping tables so that they can safely rewrite incoming and outgoing requests to avoid collision and, in the case of the controller, can map responses back to the appropriate cache.

The cache and cache controller are implemented in the Limbo programming language for the Inferno operating system; they total approximately 2400 lines of code. The client-side cache is approximately 600 lines, the server approximately 900, and the remaining 800 dedicated to plumbing - mapping structures for QIDs, Fids, and File structures, and infrastructure (module loading, argument parsing).

3.2.1. The Cache Controller

The cache controller is a constructed from a set of concurrent processes synchronizing through message passing. On receiving a connection from a client, the cache controller starts a number of processes to handle the per-client state: *remoteproc, tmsgfd2chan, rmsgfd2chan,* and *sjournalproc.*

Remoteproc exports a path in its namespace as the main filesystem to its client; to do this, it constructs a pipe and spawns an asynchronous exportfs kernel process. It then constructs two processes, *rmsgfd2chan* and *tmsgfd2chan* , to convert reads and writes on the pipe and client file descriptors into Limbo channel messages.

The *remoteproc* accepts Styx T-messages from its client and R-messages from the exportfs kernel process. It dispatches messages to the correct destination, based on a message's fid. Messages destined to the main file system are forwarded along the pipe to the exportfs process; most messages destined to the cache control file system are handled internally and receive a synchronous response. Read requests on journal files, however, are handled by starting a process, *handle_async_cc_read* , which returns data from the journal when it is available.

When a client cache first starts up and connects to a cache controller, it attaches to the cache-control file system and attempts to open its journal file, identified by a UUID. If its journal file does not exist, it attempts to create and then open it. In the cache controller, creating a journal starts another process, sjournalproc, which mediates sending and buffering invalidation messages from the cache controller to the client. Sjournalproc listens on two per-journal Limbo channels; one receives cache invalidation events, the other receives response channels from *handle_async_cc_read* Handle_async_cc_read provides sjournalproc with a channel in response to a client cache read on its journal. If the journal has any outstanding updates, it sends them along the new channel and drops its references to both the events and the return channel. Otherwise, the reply along the return channel is withheld until events are available. In this way, journals can continue to enqueue events even when a client is not attached.

3.2.2. The Client

The client-side cache acts as a Styx server on its standard input/output, for its client, and as a Styx client to the cache controller. The cache is constructed from a set of concurrent processes, similar to the cache controller. These processes, *tmsgfd2chan, rmsgfd2chan, msg2wire, journalproc,* and *scfs,* forward local Styx messages, return responses, and generate journal messages. They also maintain the read and stat cache data structures. *Tmsgfd2chan* and *rmsgfd2chan* are as in the cache controller – they convert Styx messages to Limbo channel message. *Msg2wire* provides synchronization for the connection to the cache controller, so that the main process *scfs* and the journal process *sjournalproc* do not interfere.

Scfs is the cache main process. It receives Styx T-messages from a client, typically Inferno's devmnt, rewrites the FIDs, and forwards most of those messages to the remote cache controller it is connected to. For TStat and TRead messages, it looks up the live file structure by its FID; the per-FID structure points to a per-QID structure, which holds a copy of the file's directory entry and a reference to its read cache. So long as the directory entry and read cache are present, they are used to serve reads entirely locally.

Journalproc is the main journal process; on starting, it attaches to the cache control *aname* and attempts to open its journal via a UUID; if the journal is not present, it creates it. *Journalproc* then enters a state machine, issuing Reads to the journal file and waiting for replies. On receiving a reply, it extracts the encoded QID, looks up the per-QID structure via a hash table, and invalidates both the directory entry and read cache contents for that file, thus keeping the read cache current.

4. Results

As a test workload, as well as a mechanism for ensuring that our code worked, we use our own build process as a benchmark. This involves running `mk` and the `limbo` compiler, reading system headers and our source files, and generating our `dis` executables. The total number and distribution of RPCs for the `mk` workload is provided in the Execution Characteristics section. We hope it provides a typical, read-heavy workload.

Measurements were taken in a few environments:

1. a trans-Pacific link, from a client in Baltimore to a server in Tokyo, Japan. Typical latency in this link was roughly 180 ms.
2. a trans-continental U.S. link, from a client in Baltimore to a server in San Francisco. Typical latency here was roughly 90 ms.
3. with the client and server both on the same machine.

RPC traces were captured with `mount-S` and a tool to capture Styx protocol traces, `statlisten.` For execution workloads, timing data, as reported by `time`, as shown is the average of three runs.

All jccfs measurements were taken in Inferno 4e on a Linux 2.6 host. All measurements of cfs were taken on a native Plan 9 CPU Server.

Results are not directly comparable between the Plan 9 and Inferno systems – the Plan 9 system hardware was different than that of the Inferno systems and the Plan 9 client issues a different number and distribution of RPCs to our cache. RPC counts and percent wall-clock time reduction are perhaps the most useful statistics.

Total RPC counts, for un- and cold-cache behavior

Host	Job	Walk	Clunk	Stat	Read	Write	Open	Create+Rem	TOTAL
Inferno									
(uncached)	mk all	151	83	7	99	49	62	14	465
(uncached)	re-mk	5	3	0	6	0	3	0	17
(uncached)	mk nuke	28	5	0	12	0	5	14	64
(jccfs)	mk all	151	83	7	139 *	48	62	14	504
(jccfs)	re-mk	5	3	0	3	0	3	0	14
(jccfs)	mk nuke	18	5	0	18 **	0	5	14	60
Plan 9									
(uncached)	mk all	151	83	14	97	21	55	14	435
(uncached)	re-mk	5	3	0	4	0	3	0	15
(uncached)	mk nuke	26	5	2	8	0	5	14	60
(cfs)	mk all	151	83	14	43	21	55	14	381
(cfs)	re-mk	5	3	0	4	0	3	0	15
(cfs)	mk nuke	26	5	2	8	0	5	14	60

*: 139 TReads were issued; 96 were asynchronous and to the cache aname; 43 were to the main aname **: 18 TReads were issused; 7 were asynchronous and to the cache aname; 11 were to the main aname

mk all times vs latency

Latency	System	Uncached	Cold	Hot
14 ms	Inferno/jccfs	7.8 s	7.5 s (3.8%)	6.8 s (13%)
90 ms	Inferno/jccfs	49.5 s	43.6 s (12%)	38.5 s (22%)
180 ms	Inferno/jccfs	92.7 s	80.3 s (13%)	73.1 s (21%)
180 ms	Plan 9/cfs(4)	103.1 s	79.4 s (23%)	72.2 s (30%)

re-mk times vs latency

Latency	System	Uncached	Cold	Hot
90 ms	Inferno/jccfs	2.8 s	-	1.8 s (36%)
180 ms	Inferno/jccfs	5.3 s	3.3 s (38%)	2.9 s (45%)
180 ms	Plan9/cfs(4)	3.9 s	2.3 s (41%)	2.3 s (41%)

mk nuke times vs latency

Latency	System	Uncached	Cold	Hot
180 ms	Inferno/jccfs	12.8 s	11.9 s (7.0%)	6.6 s (48%)
180 ms	Plan9/cfs(4)	11.4 s	11.1 s (2.6%)	6.1 s (46%)

We demonstrate a percentage wall clock time reduction roughly in line with cfs(4), though we pay a little bit for our (unoptimized; see future work) journals.

5. Discussion and Conclusions

Our implementation of Journal Callbacks and cache for Styx exhibits similar performance gains in absolute time to the Plan 9 cfs cache. Also looking at the timing measurements in the previous section, as latency rises, both cfs and jccfs scale similarly - their mechanism of operation is very different, however. Cfs reduces the total number of read requests. JC Cfs, while increasing the number of RPCs, serves both Stat and Read requests from its cache.

Compared to cfs or an uncached client, jccfs increases the total number of TRead requests. Why does jccfs exhibit any performance gain over an uncached client, then? The answer is that reads to the cache control aname are not synchronous with RPCs to the main aname. When we described Styx as 'performing poorly' on high-latency links, one the reasons is that Styx clients synchronously wait for RPC responses before sending future messages, making poor use of a network's Bandwidth-Delay Product. The added cost of journal reads and responses make use of this available capacity to enable server-initiated notifications.

An initial implementation of Journal Cachebacks for caching appears to be approximately as effective as the open-to-close coherency cache, cfs. We have shown that our technique is viable and that even a minimal implementation offers real-world performance gains while maintaining the Styx protocol.

6. Future Work

There are a number of avenues of further investigation which merit attention. First, while we have shown that our scheme works, we have not yet shown at least one of its conjectured major strengths. Second, our journal metadata could be exposed and/or enhanced to great effect. Third, we have found, through testing and observation, a few opportunities to improve the behavior of Inferno's Styx client. Fourth, we feel we are in a good position to give well-motivated suggestions for a hypothetical next-generation Styx.

6.1. On-Disk Caching

As a proof of concept, our cache does quite well; however, it is unable to demonstrate a serious advantage of the JC scheme over that of cfs(4): a cache which has been offline may quickly catch up. We expect an on-disk, terminal-side jccfs cache to dramatically reduce the number of RPCs generated at startup relative to cfs(4). Our cache controller already ensures that journals are maintained and populated even after the cache has hung up or closed its journal; the on-disk cache simply would have taken more time than we had.

6.2. Exposing Notifications to Programs

The journal callback mechanism and controller, if present on a given mount, could be made to provide a file or file system monitoring API, similar to Linux's `inotify`[7] Unlike `inotify`, however, a JC-based API would work over networks (NFSv3 and prior do not seem to be sufficiently capable; NFSv4 is) and could be presented as just another kernel virtual server with ctl file, requiring no additional syscalls.

6.3. Exposing More Information to Caches

The current data stream in the journal file contains only QIDs. This is tragically little information, allowing a cache the sole action of invalidation of cached contents, even if it was the reason for the mutation. Every Twrite in fact invalidates the entire file's content as well as the containing directory. This is hardly ideal, so we would like to report "re-QID operations" to caches emitting mutation operations. Similarly, we could report "small" changes to directories (version increment and other stat changes, insertions, deletions) and possibly even files.

It will be easy to accomidate these features by extending our (fortunately not yet standardized and documented) on-the-wire protocol to embed additional data records after

each QID in the journal. We note that as long as these data are well framed (by using, e.g., a TLV format), the protocol is extensible without having to update all caches and controllers in lockstep: a cache encountering a metadata field it does not understand may simply revert to invalidating all data corresponding to the QID, as if there were no ancillary fields.

6.4. Changes to `devmnt` or Additional Latency Reductions

We have observed Inferno's `devmnt` to generate unnecessary RPCs, such as

; pwd	; cd .
Tmsg.Walk(1,45,26,nil)	Tmsg.Walk(1,46,45,nil)
Rmsg.Walk(1,array[] of {})	Rmsg.Walk(1,array[] of {})
Tmsg.Open(1,26,0)	
Rmsg.Open(1,Qid(16r8,73,16r80),8192)	
Tmsg.Clunk(1,26)	Tmsg.Clunk(1,46)
Rmsg.Clunk(1)	Rmsg.Clunk(1)

For the `pwd` case, a single Tstat request would have sufficed. In the `cd.` case, no messages should have been sent at all. Whether these traces are due to bugs or deliberate simplification of the logic in `devmnt` is unclear. We note, however, that for correctness we can not avoid sending Topen messages over the wire, and so the impact of an intermediator may be lower than the impact of an optimal rewrite of `devmnt`.

We do not currently, but are in a good position to, synthesize Rwalk messages to entries in cache and only instantiate them on demand. Walks that merely clone may be thought of as always in cache. This would eliminate both RPCs in the cd case. For ordinary file objects, we could also merge open requests to keep only one such fid open over the wire.

Further, Tclunk messages may Rerror but the result is not meaningful if so: the fid is no longer valid and the semantics of close() are that it cannot fail when given a real, open fd. Since the kernel knows the openness state of an fd, there is no need for the kernel to wait a full RTT when it emits a Tclunk. We have shown remarkable improvements in performance (as might be expected from the RPC count table, above) by merely generating Rclunk messages in the cache.

6.5. Recommendations for a Next-edition Styx

Based on the implementation effort and measurements done above, we have some small recommendations to make the protocol more amenable to intermediation:

6.5.1. Standardize Identification of Synthetic Files

Currently, Styx does not have a standard mechanism for discriminating between synthetic and real files. By real files, we mean Styx objects which are expected to read back what was written and, in the case of single open or successful exclusive open, report the same contents for each read through start to finish. On the other hand, things like control files, named pipes, and network connections, are "synthetic" – that is, we are not surprised, and do not assume another concurrent user, when Tread at a fixed

---

Directories do not permit seeking and are generally considered "real". For non-QTDIR objects, we may further require that all reads at the same offset, again with only one active user, return the same contents.

offset returns different data or an error.

This lack of differentiation means that currently care must be taken as to what portions of the namespace (equivalently, what servers) are subject to (caching) intermediation. For example, cfs intermediating an imported /net would almost surely render the /net so imported inoperable.

A previous proposal, to add a bit to the type field of the QID to indicate the syntheticness (or realness) of an object was previously put forward [8] and met with some resistance. There is an undocumented convention that synthetic files have a QID version of zero. We propose that this be a requirement of correct servers. With objects so labeled, our cache and controller will be able to correctly know when to simply pass requests through to the server.

6.5.2. Standardize or Report Mutation Effects on QIDs

Currently, in response to a mutation, our cache controller must walk a fid to the server's file and re-collect the stat information, despite already knowing the mutated object(s) *state.* In particular, this walk is necessary to recover the version field, which is entirely under server control. We therefore suggest that a next-generation Styx standardize on the already traditional behavior of merely incrementing the version on every mutation event. Should this standardized behavior be seen as unacceptable, we suggeset instead that all mutating Styx RPCs be adjusted to return the QID of objects mutated – Rcreate must contain two QIDs, all others just one.

We note in passing that there has been some discussion of making version fields propagate to root. If this behavior were ever adopted, our cache and cache controller could certainly be made to work with it, but since it is not likely to be standardized we propose that servers doing anything atypical with the version field be forced to declare so in their Rversion messages, so that (our) intermediators may either adopt the propagation of updates to root behavior or fall back to more pessimistic but safe behaviors.

7. Acknowledgements

We would like to acknowledge David Eckhardt for getting the idea of journals planted in one of our heads many years ago.

1. Rob Pike and Dennis Ritchie, *The Styx Architecture for Distributed Systems*.
2. Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano, and Spyros Lalis, ‘‘Op: Styx batching for High Latency Links,’’ *IWP9 2007*.
3. *Network File System*, http://nfs.sourceforge.net/.
4. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, ‘‘Andrew: A distributed personal computing environment.,’’ *Commun. ACM* **29**(3), pp. 184-201 (Mar. 1986).
5. L. Mummert and M. Satyanarayanan, *Variable Granularity Cache Coherence*.
6. *Common Internet File System*, http://msdn.microsoft.com/en-us/library/aa365233(VS.85).aspx.
7. Robert Love, *Kernel Korner – Intro to inotify*, http://www.linuxjournal.com/article/8478.
8. Francisco J Ballesteros, *[9fans] QTCTL?*, http://9fans.net/archive/2007/10/539.

Building complex GUIs in Plan 9

Jonas Amoson
jonas.amoson@hv.se

University West
461 86 Trollhättan, Sweden

ABSTRACT

How can non-trivial graphical user interfaces be designed in Plan 9 without them losing their minimalistic style? Different toolkits are discussed, and a proposal for a tabbed toolbar is suggested as a way to add functionality without cluttering the interface and avoiding the use of pop-up dialog boxes. A hypothetical port of the GUI in LyX is used as an example.

1. Introduction

The user interface of Plan 9 sometimes confuses newcomers from other systems, as they are used to grouping user interfaces into being either text-based or graphical, and Plan 9 is neither or both. It is graphical from the ground up and the mouse needs to be used all the time, despite the fact that most programs are text based command line tools and that the system is configured by editing configuration files.

The next discovery is that the graphical programs look quite different from their cousins on other popular GUIs on X11 or Windows. They are very minimalistic and do without well-known graphical components such as dialog boxes and buttons with icons. One reason for this might be an aspiration for a very clean interface where traditional GUI widgets do not fit in. Another reason could be that all the graphical applications written so far, being comparatively simple, have not needed much help from menus and buttons in their interfaces.

This paper tries to shed light on the design of graphical interfaces for Plan 9 applications focusing on interactive components such as menus, toolbars and dialog boxes. How does one design an application GUI in Plan 9 that has many functions but still blends well with the minimalistic look of the system? Existing Plan 9 applications has been examined in order to rough out a "look-and-feel" style guide in comparison with traditional WIMP (Windows, Icon, Mouse, Pointer) systems.

Toolkit libraries are the spine of graphical applications, largely defining their behaviour. The graphics libraries libdraw, libpanel, libframe and libcontrol are presented and discussed in the context of what graphical components they provide a GUI programmer.

To exemplify, a hypothetical port of the GUI for the word processor LyX is discussed, and a proposal for a tabbed toolbar is suggested in order to keep the interface clean and to avoid pop-up windows.

Some conclusions are drawn from this work, but the main purpose of the paper is to inspire discussions on how to design GUIs for applications with somewhat more complex GUI needs, without losing the minimalistic style of Plan 9.

2. GUI philosophy

This is an attempt to outline a *style guide* for graphical programs in Plan 9 by observing the current pool of applications in the distribution and also to some extent in the contrib directory. The first observation is:

> **Keep the graphical interface as minimalistic as possible, with as few functions as possible.**

This embodies the principle of “The fewer the functions, the easier to use” [1]. If some functionality is supplied by another program, do not include it in a new program too. This rule “one tool for each task” or rule of modularity [2] is easier to achieve on the command line than in a graphical interface, but the *plumbing* mechanism [3] in Plan 9 might be a way to let the user access functionality of other tools, such as Postscript previewer, without including extra code for previewing in the application.

There is no need to make everything configurable. Users will accept a well chosen default, and it makes programs less complex [1].

> **Maximise the usage of the space in the working window for actual content.**

The windowing system has no title bars, and most applications have no menus or toolbars that are visible all the time, but use context menus activated by button 2 and 3 on the mouse.

> **A graphical program should stay within the borders of the window it was started in.**

As a graphical program “takes over” the screen area of the window from which it was executed, the user does not get surprised by windows popping up on the screen, and resizing is fully controlled by the user [1]. As an exception to this rule some programs, mostly games, automatically change the size of the window to a size that fits the application. The changed size persists when the user quits the game.

Popping up secondary windows (such as dialog boxes) is common in GUIs of other systems, but almost unseen among plan 9 applications. Instead some programs like *acme(1)* split the window into multiple frames, using *tiling* [4, p. 348].

Some programs use scroll–bars if the content of a window (or a frame) doesn’t fit, whereas some applications simply show as much as fits, eventually forcing the user to resize the window to be able to access the hidden parts.

> **Icons (small stylistic pictures) are usually not used on buttons or toolbars, instead ordinary text is used.**

Graphical images are not evil, but text occupies less space, and is often easier to understand [5, p. 168]. An example is the editable toolbar in *acme* where shortcuts to commands can be added just by typing.

3. Toolkits

Libraries of GUI components are often called *toolkits* and are constructed to ease application development, so that the programmer does not have to write lots of low–level graphics code in realising a GUI. Toolkits influence the look–and–feel of an application, so programs will probably look better together, if they were developed using the same toolkit.

All programs that use the graphical subsystem of Plan 9 will use libdraw, the basic graphics library described in *draw(2).* Libdraw does not provide much help with widgets, but together with *event(2)* it is easy to make “mouse button popup–menus”, the most commonly used interaction component of graphical applications in Plan 9.

3.1. libpanel

The panel library was developed by Tom Duff in order to write the Mothra web browser [6]. Panel offers traditional GUI components like pull down menus, input boxes, radio buttons, etc. and has a 3D−look−and−feel†, found in most GUIs today.

Although the look and feel of panel is minimalistic compared to many other 3D toolkits, it does not blend so well with the style of other plan 9 applications. Libpanel is not included in the standard distribution (4th edition) of Plan 9.

3.2. libcontrol

The *control(2)* library developed by Rob Pike and Sape Mullender provides a set of interactive controls (widgets) using the thread library, *thread(2).* Each control has its own thread, and it is possible to send it messages and to receive events from it. Libcontrol is, to my understanding, the preferred toolkit for new applications with complex graphical user interfaces.

3.3. libframe

Another library *frame(2)* provides "frames of editable text in a single font" and is used by applications like *acme* and *rio(1).*

3.4. Toolkit features and use

Table 1 shows the GUI features provided by each of the above described toolkits and also an estimate of how frequently the libraries are used by applications in /sys/src.

Table 1: Toolkit summary.

Component	draw	frame	panel	control
Button			x	x
Context menu	x		x	x
Pulldown menu			x	
Slider		x	x	
Text panel		x	x	x
Number of apps in /sys/src	43	5	–	3

4. Case study − LyX

To practically investigate GUI design options for Plan 9, a port of the graphical interface of LyX, the LaTeX frontend, will be discussed.

> "*LyX is a document processor that encourages an approach to writing based on the structure of your documents (WYSIWYM) and not simply their appearance (WYSIWYG).*" [http://www.lyx.org]

LyX was chosen because it has a featureful GUI that simplifies the editing task compared to using a markup language and a standard text editor. It is also a good example of a GUI frontend for a text based application.

When porting an application, it is easier to break with the style of the target system, than in a fresh design that are built bottom−up. Considerations should be made to follow the style of the target system, or if needed, carefully extend the style in order to embrace new types of applications.

† 3D in the sense that shadowed borders give the impression of a raised or depressed button.

4.1. GUI description

To get a grip of the functionality that has to be ported, we start with an overview over the current graphical interface of LyX. Figure 1 shows a screenshot of LyX running under Linux, with a document loaded into the editing buffer. The interface consists top-down of:

- Pull down menu-bar (File, Edit, View, Insert, ...)
- Icon toolbar with buttons for commonly used functions
- Writing area
- Status bar

The writing area lets the user edit text in different fonts, but also handles *insets* such as figures, tables and equations. The writing area will not be further discussed in this paper. The toolbar with buttons also features a pull-down menu for selecting paragraph types in the document (standard, section, subsection, quotation, ...).

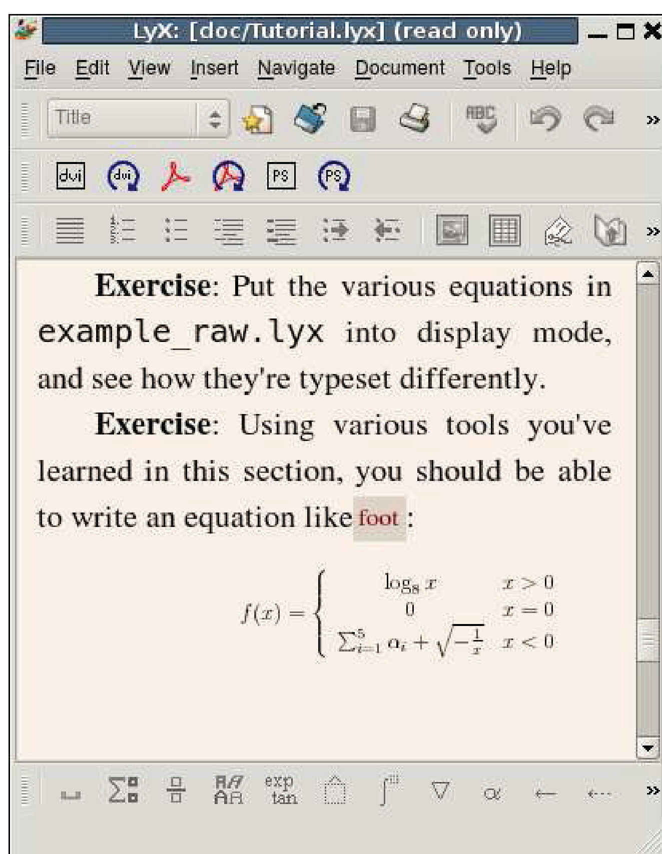


Figure 1: Screenshot of LyX running under Linux.

4.2. User interaction

Some of the pull down menus in LyX open dialog boxes (or pop-up windows) with further options. Figure 2 shows the box for controlling the layout of paragraphs. Settings for the document as whole (meta-data) is configured in a dialog box with a tree graph with sections for each types of configuration.

Most of the pop-up windows in LyX are “stay on top” but *modeless*†, enabling the

user with a large screen to keep the windows open while editing the document. Other menu items and menu buttons take direct action in the document, some by inserting a formula box in the document, that is then further filled out by the user.

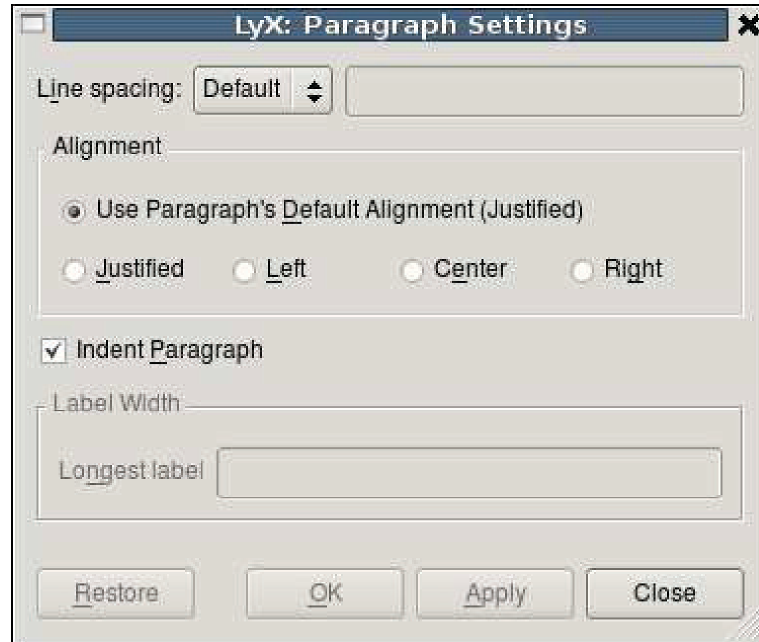


Figure 2: Paragraph settings in LyX.

4.3. Choice of toolkit

There are many possible routes to take in translating a GUI to the Plan 9 environment. One obvious way would be to create an interface as identical as possible to its original form, but the risk is then high that it will feel ill-fitted in its new environment.

For a translation function-by-function, the Panel library would probably be a first choice, as it has the widest support for traditional GUI concepts like pull-down menus and other 3D widgets, see table 1. But as mentioned above, using the Control library would make the application fit better with the overall style of Plan 9, and the discussion that follows assumes a port based on libcontrol.

4.4. Menus

The graphical interface in LyX is full of menus and buttons, whereas the typical plan 9 program is not, so how does one proceed? Some functionality can be fulfilled by other tools, and can therefore be left out; this includes Postscript and PDF-export, dialog boxes for opening and saving files as well as printing.

Menu items that cannot easily be omitted, without compromising the usefulness of LyX compared to hand editing the LaTeX code, include setting headings as well as insertion of figures and tables.

Context menus (activated by pressing mouse button 2 or 3) could be used for some functionality, but putting too many menu items, or even sub-level items, in them should be discouraged [1]. Many users of Plan 9 have also become used to the mouse

---

† Modeless as opposed to modal pop-up windows that locks the main window until the user has closed the dialog box [4, p.355]. Modeless windows that are open all the time are often called *tool boxes*.

chording mechanism in *acme* and would probably wish for similar functionality in other applications for heavy text editing, reserving the mouse buttons for that purpose.

4.5. The toolbar

A toolbar can be seen as a portion of the graphical interface reserved for commands and buttons, in contrast to the main working area, in our case the writing area in LyX. The text editor *sam(1)* has a window for entering written commands and many frame based applications such as *acme* and the web browser *abaco(1)* have toolbars with text buttons.

The toolbar is accessed using the mouse, which would be in line with Plan 9 style, as the system is quite mouse intensive compared to other graphical systems, especially as keyboard shortcuts are not used much. One drawback with the toolbar is that it easily gets messy if it is filled with different buttons and options, another negative aspect is that it wastes available screen space. A proposed toolbar solution for a port of the LyX GUI will be discussed later.

4.6. Dialog boxes

Dialog boxes are quite common in traditional graphical interfaces found on Unix, Windows or Mac OS, and they are used mostly when the application needs to ask the user something more in detail, like where to save a file and in what format. Another use for dialog boxes is for setting preferences, especially in systems where text-based configuration files are avoided at all cost. Dialog boxes have the advantage, compared to toolbars, of not occupying screen area when they are not activated.

As dialog boxes are uncommon in graphical programs designed for Plan 9 it might be desirable to do without them in a port of the LyX GUI. One way could be to sub-frame the working window whenever a dialogue has to be made with the user, the frame is again removed as soon as the user is finished with it. The good thing about this solution is that the application stays within its allocated window area, but the main drawback is that the editing window has to shrink its size temporarily, which may seem even more annoying to the user than the opening of secondary pop-up windows. Another suggestion is to reserve a fixed sub-frame for dialogue purposes, avoiding pop-ups and shrinking frames.

4.7. Dynamic dialog toolbar

If space is to be withheld for a toolbar or for user dialogues, the use of it ought to be as efficient as possible. One idea is to use a menu-tabbed toolbar much like the *Ribbon* in Microsoft Office 2007 [7]. A crude mock-up of how it could look is shown in figure 3.

The idea with the tabbed toolbar is that the user first selects one of the tabs, say *paragraph settings* which causes the toolbar to be filled with controls associated with paragraphs. Reaching a function now requires an extra click on the tab, but only if the user wasn't already editing the paragraph settings for some other paragraph in the document.

5. Conclusions

An outspoken goal of Plan 9 is to keep the GUI simple and clean. This can be accomplished by reducing functionality or dividing the tasks into multiple programs, thereby eliminating the need for more complex UI structures such as dialog boxes and button toolbars. When this cannot be done without losing too much of the usefulness of having a GUI for the application, new directions have to be found.

For GUIs with lots of functionality, as in the word processor LyX, a dynamic dialog toolbar is proposed as an efficient use of screen space in an attempt to reduce the need for annoying pop-up windows.

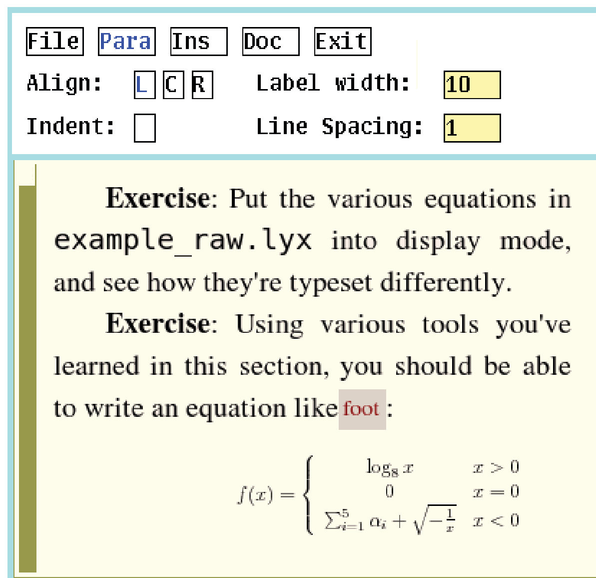


Figure 3: Mock-up of LyX with a tabbed toolbar.

6. References

[1] Rob Pike, “Window Systems Should Be Transparent”, *Computing Systems,* Vol. 1, 3, pp. 279—296, Summer 1988.

[2] Eric S. Raymond, *The Art of Unix Programming,* Addison-Wesley, 2004.

[3] Rob Pike, “Plumbing and Other Utilities”, *Proceedings of the 2000 USENIX Technical Conference,* pp. 159—170, San Diego, 2000.

[4] Wilbert Galitz, *The Essential Guide to User Interface Design,* Wiley Computer Publishing, 2002.

[5] Jeff Raskin, *The Humane Interface, New Directions for Designing Interactive Systems,* Addison-Wesley, 2000.

[6] Tom Duff, “A Quick Introduction to the Panel Library”, available online at `http://plan9.bell-labs.com/sources/extra/mothra/libpanel/panel.pdf`

[7] Microsoft Corp., “The Office 2007 Ribbon Overview”, published online at `http://office.microsoft.com`

Using Currying and process-private system calls to break the one-microsecond system call barrier

Ronald G. Minnich[*,†] and John Floren and Jim Mckie [‡]

January, 2009

Abstract

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. In contrast to 20 years of tradition in High Performance Computing (HPC), we require that programs access network interfaces via the kernel, rather than the more traditional (for HPC) OS bypass.

In this paper we discuss our research in modifying Plan 9 to support sub-microsecond "bits to the wire" (BTW) performance. Rather than taking the traditional approach of radical optimization of the operating system at every level, we apply a mathematical technique known as Currying, or pre-evaluation of functions with constant parameters; and add a new capability to Plan 9, namely, process-private system calls. Currying provides a technique for creating new functions in the kernel; process-private system calls allow us to link those new functions to individual processes.

1 Introduction

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. Our research goals in this work are aimed at rethinking how HPC systems software is structured. One of our goals is to re-examine and, if possible, remove the use of OS bypass in HPC systems.

OS bypass is a software technique in which the application, not the operating sytem kernel, controls the network interface. The kernel driver is disabled, or, in some cases, removed; the functions of the driver are replaced by an application or library. All HPC systems in the "Top 50", and in fact most HPC systems in the Top 500, use OS bypass. As the name implies, the OS is completely

[*]

[†]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration under contract DEAC0494AL85000. SAND- 2009-5156C.

[‡]Bell Laboratories, Murray Hill, NJ, USA

bypassed; packets move only at the direction of the application. This mode of operation is a lot like the very earliest days of computers, where not a bit of I/O moved unless the application directly tickled a bit of hardware. It involves the application (or libraries) in the lowest possible level of hardware manipulation, and even requires application libraries to replicate much of the operating systems capabilities in networking, but the gains are seen as worth the cost.

One of the questions we wish to answer: is OS bypass still needed, or might it be an anachronism driven by outdated ideas about the cost of using the kernel for I/O? The answer depends on measurement. There is not much doubt about the kernel's ability to move data at the maximum rate the network will support; most of the questions have concerned the amount of time it takes to get a message from the application to the network hardware. So-called short message performance is crucial to many applications.

HPC network software performance is frequently characterized in terms of "bits to the wire" (BTW) and "ping-pong latency". Bits to The Wire is a measure of how long it takes, from the time an application initiates network I/O, for the bits to appear on the physical wire. Ping-pong latency is time it take a program to send a very small packet (ideally, one bit) from one node to another, and get a response (usually also a bit). These numbers are important as they greatly impact the performance of collectives (such as a global sum), and collectives in turn can dominate application performance [2] [4] In an ideal world, ping-pong latency is four times the "bits to the wire" number. Some vendors claim to have hit the magical 1 microsecond ping-pong number, but a more typical number is 2-3 microseconds, with a measured BTW number of 700 nanoseconds. However, these numbers always require dedicated hosts, devices controlled by the application directly, no other network activity, and very tight polling loops. The HPC systems are turned into dedicated network benchmark devices.

A problem with OS bypass is that the HPC network becomes a single-user device. Because one application owns the network, that network becomes unusable to any other program. This exclusivity requires, in turn, that all HPC systems be provisioned with several networks, increasing cost and decreasing reliability. While the reduction in reliability it not obvious, one must consider that the two networks are not redundant; they are both needed for the application to run. A failure in either network aborts the application.

By providing the network to programs as a kernel device, rather than a set of raw registers, we are making HPC usable to more than just specialized programs. For instance, the global barrier on the Blue Gene systems is normally only available to programs that link in the (huge) Deep Computing Messaging Facility (DCMF) library or the MPI libraries[1], which in turn link in the DCMF. Any program which wishes to use the HPC network must be written as an MPI application. This requirement leads to some real problems: what if we want the shell to use the HPC network? Shells are not MPI applications; it makes

[1]MPI libraries are typically much larger than the Plan 9 kernel; indeed, the configure script for OpenMPI is larger than the Plan 9 kernel

no sense whatsoever to turn the shell into an MPI application, as it has uses outside of MPI, such as starting MPI applications!

On Plan 9 we make the global barrier available as a kernel device, with a simple read/write interface, so it is even accessible to shell scripts. For example, to synchronize all our boot-time scripts, we can simply put `echo 1 > /dev/gib0barrier` in the script. The network hardware becomes accessible to any program that can open a file, not just specialized HPC programs.

Making network resources available as kernel-based files makes them more accessible to all programs. Seperating the implementation from the usage reduces the chance that simple application bugs will lock up the network. Interrupts, errors, resources conflicts, and sharing can be managed by the kernel. That is why it is there in the first place. The only reason to use OS bypass is the presumed cost of asking the kernel to perform network I/O.

One might think that the Plan 9 drivers, in order to equal the performance of OS bypass, need to impose a very low overhead – in fact, no overhead at all: how can a code path that goes through the kernel possibly equal an inlined write to a register? The problem with this thinking, we have come to realize, is the fact that complexity is conserved. It is true that the OS has been removed. But the need for thread safety and safe access to shared resources can not be removed: the support has to go *somewhere*. That somewhere is the runtime library, in user mode.

Hence, while it is true that OS bypass has zero overhead in theory, it can have very high overhead in fact. Programs that use OS bypass always use a library; the library is usually threaded, with a full complement of locks (and lockiing bugs and race conditions); OS functions are now in a library. In the end, we have merely to offer lower overhead than the library.

There are security problems with OS bypass as well. To make OS bypass work, the kernel must provide interfaces that to some extent break the security model. On Blue Gene/P, for example, DMA engines are made available to programs that allow them to overwrite arbitrary parts of memory. On Linux HPC clusters, Infiniband and other I/O devices are mapped in with mmap, and users can activate DMAs that can overwrite parts of kernel memory. Indeed, in spite of the IOMMUs which are supposed to protect memory from badly behaved user programs, there have been recent BIOS bugs that allowed users of virtual network interfaces to roam freely over memory above the 4 gigabyte boundary. Mmap and direct network access are really a means to an end; the end is low latency bits to the wire, not direct user access. It is so long since the community has addressed the real issue that means have become confused with ends.

2 Related work

The most common way to provide low latency device I/O to programs is to let the programs take over the device. This technique is most commonly used on graphics devices. Graphics devices are inherently single-user devices, with

multiplexing provided by programs such as the X server. Network interfaces, by contrast, are usually designed with multiple users in mind. Direct access requires that the network be dedicated to one program. Multi-program access is simply impossible with standard networks.

Trying to achieve high performance while preserving multiuser access to a device has been achieved in only a few ways. In the HPC world, the most common is to virtualize the network device, such that a single network device appears to be 16 or 32 or more network devices. The device requires either a complex hardware design or a microprocessor running a real-time operating system, as in Infiniband interfaces: thus, the complex, microprocessor-based interfaces do bypass the main OS, but don't bypass the on-card OS. These devices are usually used in the context of virtual machines. Device virtualization requires hardware changes at every level of the system, including the addition of a so-called iommu [1].

An older idea is to dynamically generate code as it is needed. For example, the code to read a certain file can be generated on the fly, bypassing the layers of software stack. The most known implementaiton of this idea is found in Synthesis [3]. While the approach is intriguing, it has not proven to be practical, and the system itself was not widely used.

The remaining way to achieve higher performance is by rigorous optimization of the kernel. Programmers create hints to the compiler, in every source file, about the expected behaviour of a branch; locks are removed; the compiler flags are endlessly tweaked. In the end, this work results in slightly higher throughput, but the latency – "bits to the wire" – time changes little if at all. It is still too slow. Recent experiences shows that very high levels of optimization can introduce security holes, as was seen when a version of GCC optimized out all pointer comparisons to NULL.

Surprisingly, there appears to have been little other work in the area. The mainline users of operating systems do not care; they consider 1 millisecond BTW to be fine. Those who do care use OS bypass. Hence the current lack of innovation in the field: the problems are considered to be solved.

The status quo is unacceptable for a number of reasons. Virtualized device hardware increases costs at every level in the I/O path. Device virtualization adds a great deal of complexity, which results in bugs and security holes that are not easily found. The libraries which use these devices have taken on many of the attributes of an operating system, with threading, cache- and page-aligned resource allocation, and failure and interrupt management. Multiple applications using multiple virtual network interfaces end up doing the same work, with the same libraries, resulting in increased memory cost, higher power consumption, and a general waste of resources all around. In the end, the applications can not do as good a job as the kernel, as they are not running in priveleged mode. Applications and libraries do not have access to virtual to physical page mappings, for example, and as a result they can not optimize memory layout as the kernel code.

3 Our Approach

Our approach is a modification of the Synthesis approach. We do create curried functions with optimized I/O paths, but we do not generate code on the fly; curried functions are written ahead of time and compiled with the kernel, and only for some drivers, not all. The decision on whether to provide curried functions is determined by the driver writer.

At run time, if access to the curried function is requested by a program, the kernel pre-evaluates and pre-validates arguments and sets up the parameters for the driver-provided curried function. The curried function is made available to the user program as a private system call, i.e. the process structure for that one program is extended to hold the new system call number and parameters for the system call. Thus, instead of actually synthesizing code at runtime, we augment the process structure so as to connect individual user processes to curried functions which are already written.

We have achieved sub-microsecond system call performance with these two changes. The impact of the changes on the kernel code is quite minor.

We will first digress into the nature of Curry functions, describe our changes to the kernel and, finally discuss the performance improvements we have seen.

3.1 Currying

The technique we are using is well known in mathematical circles, and is called currying. We will illustrate it by an example.

Given a function of two variables, $f(x, y) = y/x$, one may create a new function, $g(x)$, if y is known, such that $g(x) = f(x, y)$. For example, if y is known to be 2, the function g might be $g(x) = f(x, 2)$.

We are interested in applying this idea to two key system calls: read and write. Each takes a file descriptor, a pointer, a length, and an offset. In the case of the Plan 9 kernel, we had used a kernel trace device and observed the behavior of programs. Most programs:

- Used less than 32 distinct pages when passing data to system calls
- Opened a few files and used them for the life of the program
- Did very small I/O operations

We also learned that the bulk of the time for basic device I/O with very small write sizes – the type of operation common to collective operations – was taken up in two functions: the one that validated an open file descriptor, and the one that validated an I/O address.

The application of currying was obvious: given a program which is calling a kernel function read or write function: $f(fd, address, size)$, with the same file descriptor and same address, we ought to be able to make a new function: $g(size) = f(fd, address, size)$, or even $g() = f(fd, address, size)$.

Tracing indicated that we could greatly reduce the overhead. Even on an 800 Mhz. Power PC, we could potentially get to 700 nanoseconds. This compares

very favorably with the 125 ns it takes the hardware to actually perform the global barrier.

3.2 Connecting curry support to user processes

The integration of curried code into the kernel is a problem. Dynamic code generation looks more like a security hole than a solution.

Instead, we extended the kernel in a few key ways:

- extend the process structure to contain a private system call array, used for fastpath system calls
- extend the system call code to use the private system call array when it is passed an out-of-range system call number
- extend the driver to accept a fastpath command, with parameters, and to create the curried system call
- extend the driver to provide the curried function. The function takes no arguments, and uses pre-validated arguments from the private system call entry structure

4 Implementation of private system calls on Plan 9 BG/P

To test the potential speeds of using private system calls, a system was implemented to allow fast writes to the barrier network, specifically for global OR operations, which are provided through `/dev/gib0intr`. The barrier network is particularly attractive due to its extreme simplicity: the write for a global OR requires that we write to a Device Control Register, a single instruction, which in turn controls a wire connected to the CPU. Thus, it was easy to implement an optimized path to the write on a per-process basis.

The modifications described here were made to a branch of the Plan 9 BG/P kernel. This kernel differed from the one being used by other Plan 9 BG/P developers only in that its portable `incref` and `decref` functions had been redefined to be architecture-specific, a simple change to allow faster performance through processor-specific customizations. In other words, we are comparing our curried function support to an already-optimized kernel.

First, the data structure for holding fast system call data was defined in the `/sys/src/9k/port/portdat.h` file (from this point on, kernel files will be assumed to reside under `/sys/src/9k/`, thus `port/portdat.h`).

In the same file, the `proc` struct was modified to include the following declarations:

```
/* Array of private system calls */
Fastcall *fc;
```

```
/* Our special fast system call struct */
struct Fastcall {
    /* The system call number */
    int scnum;
    /* A communications endpoint */
    Chan *c;
    /* The handler function */
    long (*fun)(Chan*, void*, long);
    void *buf;
    long n;
};
```

Figure 1: Fast system call struct

```
int cfd, gdf, scnum=256;
char area[1], cmd[256];
gfd = open("/dev/gib", ORDWR);
cfd = open("/dev/gib0ctl", OWRITE);
cmd = smprint("fastwrite %d %d 0x%p %d", scnum, fd, area, sizeof(area));
write(cfd, cmd, strlen(cmd));
close(cfd);
docall(scnum);
```

Figure 2: Sample code to set up a fastpath systemcall

```
/* # private system calls */
int fcount;
```

Programs are required to provide a system call number, a file descriptor, pointer, and length. It may seem odd that the program must provide a system call number. However, we did not see an obvious way to return the system call number if the system chose it. We also realized that it is more consistent with the rest of the system to have the client choose an identifier. That is how 9P works: clients choose the file identifier when a file is accessed. Note that, because the Plan 9 system call interface has only two functions which can do I/O, the Fastcall structure we defined above covers all possible I/O operations. The contrast with modern Unix systems is dramatic.

Next, we modified the Blue Gene barrier device, `bgp/devgib.c`, to accept `fastwrite` as a command when written to `/dev/gib0ctl`. When the command is written, the kernel allocates a new `Fastcall` in the `fc` array, using a user-provided system call number and a channel pointing to the barrier network, then sets `(*fun)` to point to the `gibfastwrite` function and finally increments `fcount`. The code to set up the fast path is shown in Figure 2.

Following the `write`, `scnum` contains a number for a private system call to write to the barrier network. From there, a simple assembly function (here

```
TEXT docall(SB), 1, $0
    SYSCALL
    RETURN
```

Figure 3: User-defined system call code for Power PC

called `docall`) may be used to perform the actual private system call. The code is shown in Figure 3.

When a system call interrupt is generated, the kernel typically checks if the system call number matches one of the standard calls; if there is a match, it calls the appropriate handler, otherwise it gives an error. However, the kernel now also checks the user process's `fc` array and calls the given `(*fun)` function call if a matching private call exists. In the case of the barrier device, it calls `gibfastwrite`, which writes '1' to the Device Control Register. The fastcall avoids several layers of generic code and argument checking, allowing for a far faster write.

5 Results

In order to test the private system call, we wrote a short C program to request a fast write for the barrier. It performs the fastpath setup as shown above. Then, it calls it calls the private system call. The private system call is executed many times and timed to find an average cost per call. As a baseline, the traditional write call was also tested using a similar procedure.

We achived our goal of sub-microsecond bits to the wire. With the traditional write path, it took approximately 3,000 cycles per write. Since the BG/P uses 850 MHz PowerPC processors, this means a normal write takes approximately 3.529 microseconds. However, when using the private system calls, it only takes around 620 cycles to do a write, or 0.729 microseconds. The overall speedup is 4.83. The result is a potential ping-pong performance of slightly under 3 microseconds, which is competitive wth the best OS bypass performance.

6 Conclusions and Future Work

Runtime systems for supercomputers have been stuck in a box for 20 years. The penalty for using the operating system was so high that programmers developed OS bypass software to get around the OS. The result was the creation of OS software above the operating system boundary. Operating systems have been recreated as user libraries. Frequently, the performance of OS bypass is cited without taking into account the high overhead of these user-level operating systems.

This paper shows an alternative to the false choice of slow operating systems paths or fast user-level operating systems paths. It is possible to use a general-

purpose operating system for I/O and still achieve high performance.

We have managed the write side of the fastcall path. What remains is to improve the read side. The read side may include an interrupt, which complicates the issue a bit. We are going to need to provide a similar reduction in overhead for interrupts.

We have started to look at curried pipes. Initial performance is not very good, because the overhead of the Plan 9 kernel queues is so high. It is probably time to re-examine the structure of that code in the kernel, and provide a faster path for short blocks of data.

Our goal, in the end, is to show that IPC from a program to a user level file server can be competitive with in-kernel file servers. Achieving this goal would help improve the performance of file servers on Plan 9.

7 Acknowledgements

This work is support by the DOE Office of Advanced Scientific Computing Research. IBM has provided support from 2006 to the present. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] Muli Ben-Yehuda, Jimi Xenidis, and Michal Ostrowski. Price of safety: Evaluating iommu performance. June 2007.

[2] Fabrizio Petrini, Darren J. Kerbyson, and Scott 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*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1:11–32, 1988.

[4] Y. Tanaka, K. Kubota, M. Sato, and S. Sekiguchi. A comparison of data-parallel collective communication performance and its application. *High Performance Computing and Grid in Asia Pacific Region, International Conference on*, 0:137, 1997.

Living under a Poison Tree

Erik Quanstrom
quanstro@coraid.com

ABSTRACT

Within the last year, the transition to the *nupas* [1] mail system has been completed at Coraid. In that time, the number of messages stored and the size of inboxes has increased by an order of magnitude. Yet the amount of storage required on the WORM has been reduced by an order of magnitude and the amount of core required has been reduced by an order and a half. Most of these gains have been realized through the caching strategies made possible by the *mdir*(6) format. Much smaller but significant optimizations and bug fixes in the last year have enabled the current level of performance.

Introduction

The introduction of *nupas* has been largely successful. Most heavy users access the system through IMAP. The system is accessed daily through Apple Mail, Firefox, Opera and Outlook. Last year at this time, both mail servers ran out of memory on a daily basis. With many very large inboxes, nearly 1GB of data per day was added to the WORM. Opening large mailboxes was understandably accompanied by a delay on the order of minutes. Today, though 14 new users have been added, less than 5% of available memory is used by *upas*, and the elimination one of the two mail servers is being considered. Only 100MB of data per day is added to the WORM. Logging now accounts for more of the dump than email. The largest mailboxes can now be opened in a few seconds. This is summarized in Figure 1.

date	total messages	total MB	largest box messages	largest box MB	dump blocks	core MB
200808	12491	1143	975	249	123000	5386
200908	157075	8307	15587	790	14500	482

Figure 1

While this improvement would not have been possible without the move to caching facilitated by moving to the *mdir* format, some surprising secondary limitations were found and some bugs were found due to faulty assumptions.

Hash Handling

The initial roll out of *nupas* was somewhat disappointing. The memory savings was less than expected. Several users were using a few hundred megabytes of core each. Part of this was explained by email clients such as Apple Mail keeping several connections open at once and the increase in IMAP-capable mobile devices. Even so, it was typical for users with large inboxes to have 50MB instances of *upas/fs* immediately upon opening the inbox. *Leak*(1) was unable to complete a scan of these processes. It was determined that linearly increasing the allocation for two large memory blocks in an interleaved fashion with other small allocations was causing pathological memory fragmentation. Switching to exponential allocation fixed *leak*, but revealed that there was no memory leak.

In addition, opening large mailboxes was taking an inordinate amount of cpu. Profiling indicated that the file hash table handling accounted for the bulk of startup time. Since the scheme for handling files was to keep them in a hash table. Each file needed a hash entry. Since each message part has 30 standard files for the message body, header, subject and so on, a message with n subparts will require $30n+1$ hash entries. Each one of these entries is allocated with *malloc*(2). A 5000 message mailbox would have a minimum of 150000 hash entries, but likely at least double that number. Since there were 1227 buckets, the load factor α on the hash table, or the average number of entries per hash chain, would be at least 122 [2], [3].

Given that the load factor is so large for such a small mailbox, and since it seems that building the hash table is expensive, it may be worth analyzing how long we expect building this table to take. Interestingly this topic does not appear to be explicitly discussed in [3]. If we consider adding a single element to a hash table, it's not hard to see that computing the bucket by hashing a predetermined set of strings with a fixed value is bounded by a constant time. Since we do need to guard against duplicates, adding an element to the hash chain, will take $O(1+\alpha_t/2)$ time [4]. Since

$$\sum_{t=0 \to \alpha} 1 + \frac{\alpha_t}{2} = \frac{(1+\alpha)^2}{4},$$

we can expect that loading the hash table will take $O(\alpha^2)$ time.

There are two potential solutions to this problem. Either replace the current lookup with one of sub-linear time and/or reduce the number of hash entries. The latter seemed like the best first approach as this could also address excessive memory use. And, given the poor big-O performance of our algorithm, any reduction of nodes will result in quadratic speedups.

The hash entries for the 30 standard files that populate each message directory were replaced with a single dummy entry entry "xxx." The file portion of the QID was stripped out. Since all message directories are numeric and all of the standard files begin with a letter, we simply lookup the dummy entry when asked to lookup file starting with a letter. If the dummy entry is found, the given name is translated to a file id by linear search of a static table. The file id is added back to the returned QID. This change reduces the number of allocated hash entries from $30n+1$ to $n+1$. Likewise the new load factor $\alpha' = \alpha/30$ and the time to build the hash table will be $O(\alpha'^2/4) = O((\alpha/30)^2) = O(\alpha^2/900)$. While this approach fails to address the quadratic behavior, it does address the memory use and provides three orders' of magnitude headroom.

It is important to note in this analysis that IMAP clients such as Apple Mail open each mail box every minute or so to check for new messages. If there are none the mail box is immediately closed. Thus the time it takes to open a mailbox is one of the most important benchmarks in our system. Also, due to the frequent mailbox scanning without closing the IMAP connection, reducing memory fragmentation is vitally important. Based on the experience with *leak*, a few small items (e.g. mime types) that were sure to be reused were freed when mail boxes are closed. This reduced the total long term memory use for *upas/fs* driven by Apple Mail by an order of magnitude.

Last year's *upas/fs* was benchmarked against this year's for a 15200–message mailbox. One further significant change has been made: the number of hash buckets has been increased to 1999. Both the time and memory usage to start and to run *nedmail*(1) are listed. The results are summarized in Figure 2.

date	start time (s)	start core (MB)	ned time (s)	ned core (MB)
200808	72	67	233	133
200908	0.40	17	3.3	17

Figure 2

While memory use is still somewhat disappointing, start time has improved by two orders of magnitude. Since memory use is predicted to be linearly related to the number of messages, it is envisioned that this issue will not need to be revisited until 30000–message mailboxes become commonplace, when it is expected that the $O(\alpha^2)$ of hash addition will again begin to be important.

Index Scanning

The scanning of the mailbox index relied on the order of messages in a mailbox being stable. A missing or extra message near the beginning of the list of messages (assumed to be in date–of–delivery order) could result in a large number of messages being deleted from the index and the mailbox. Initially it was assumed that this case was not important, since non–stable sorting would indicate a bug in the particular mail box code. Unfortunately a few bugs were found that deleted mail. It also proved a difficult problem to tackle for *mdir* mailboxes since they are sorted by date from the UNIX from line, since the order of delivery to the *mdir* is not available. Directory order is not stable when deletions are possible, since deletions on the file server simply mark a slot free and new files fill the first free directory slot. Yet new messages that are older than some (or even all) existing mail may be added.

The solution employed was to keep an AVL tree keyed on the SHA1 checksum of each message. This allowed to matching of existing message structures to index entries to be robust in the face of ordering problems or races between the index and mailbox. The AVL tree was also employed to detect duplicate messages. Duplicate messages and other rejected messages are now silently dropped rather than deleted.

Avoiding Mail Box Scans

For very large mailboxes, scanning the mailbox can be fairly resource intensive. Since mailboxes tend to be open many times, it is desirable to avoid duplicating this effort. To accomplish this, each mail box type may save a line of mailbox–specific information

to the index. If the mail box is older than the index, then the index is read without consulting the underlying mailbox. For example *mdir* mailboxes save the QID of the *mdir* directory to the index. If the QID matches the QID saved in the index, the index is considered “newer” than the *mdir*.

When the mail box scan is avoided, the difference for our 15200-message mailbox is 0.4s for the cached case versus 3.3s for the cached case.

Mdir Scanning; *Upas/fs* Scanning

The decision to sort *mdir* mailboxes by date continues to be problematic. Since new messages must be given a higher message number than older message ids, it is possible for messages to be out of numeric order. This requires all *upas/fs* clients to be aware that relative message ids may not be stable as they are with a tradition mbox format. It would be possible to declare the order of messages in the index to be the the order of messages in a mailbox. However the same problem would arise if the index is regenerated for any reason. This would seem to be an unwise dependency.

Sorting has been a particular problem for *imap4d* due to the quixotic definition of the imap UID and sequence numbers. IMAP uses two different numbers to as handles on a message. The UID is an almost permanent identifier assigned in increasing order. Sequence numbers are only valid during a session and take on values from $1 - n$, where n is the number of messages in the mail box. The difficult requirement is that if $UID_n < UID_m$ then one must have $seq_n < seq_m$. This puts the sorting of IMAP (by UID) in conflict with the sorting of *upas/fs* (by date). Rather than trying to maintain two conflicting sorted indexes, the same AVL tree solution used for index scanning was employed. Other clients such as *nedmail*(1) simply read the entire message directory and resort.

Further Work

Clearly there is a lot of room for further work. Much of the further work mentioned in [1] remains undone. There is ongoing work to multithread the the file server interface of *upas/fs*. It seems that replacing the current hashing strategy needs to be in the medium-term plans. The AVL tree strategy seems fruitful and should be reused by *mdir* and *nedmail* to avoid the need to read and sort the mail directory as a whole. Sorting of mailboxes continues to be a sticky wicket. IMAP search and listing performance needs some reevaluation.

Credits

This work wouldn’t have been possible without the patience of my coworkers at Coraid. In particular Brantley Coile and Ryan Thomas deserve special mention for suffering through many buggy versions. In addition Sape Mullender has reported many important bugs and included invaluable *snapfs*(4) debugging snapshots. Finally Gabriel Diaz Lopez De la llave has provided a lot of valuable feedback through his GSOC work related to adding multithreading to *upas/fs*.

Abbreviated References

[1] E. Quanstrom “Scaling Upas”, proceedings of the International Workshop on Plan 9, October, 2008.

[2] E. Quanstrom “(n)upas update”, email to the 9fans list, May 21, 2009, `http://9fans.net/archive/2009/05/106`.

[3] D. E. Knuth, *The Art of Computer Programming*, vol 3., 2d ed., 1998.

[4] G. H. Gonnet, “Expected Length of the Longest Probe Sequence in Hash Code Searching”, Department of Computer Science, University of Waterloo, CS–RR–78–46, 1978.

A new boot process for Plan 9 from Bell Labs

Iruatã Souza
iru.muzgo@gmail.com

ABSTRACT

We describe a new way of booting Plan 9 in which no special bootstrap program is involved; rather, the bootstraping is done by just another Plan 9 kernel. In this way, we can take advantage of the full range of devices, networks, and file systems supported by the operating system.

1. Motivation

The Plan 9 4th Edition boot process has been the target of critics from the Plan 9 users and its replacement has been discussed for years [1]. We tried to synthesize these discussions and ideas in the form of a rewrite of the relevant boot routines.

2. Introduction

Plan 9 boots on PCs with help from an auxiliary kernel called *9load(8).* Such a kernel does not pretend to be general on its purpose and does not share its source code tree with the other Plan 9 kernels. This approach has shown limitations and in order to have these limitations addressed, fundamental changes needed to be made; the current status and future directions of the new Plan 9 boot process is hereafter discussed.

3. Plan 9 on PC

In order for this paper to be self-contained, this section presents the minimum required information about Plan 9 running on computers of the x86 architecture.

3.1. Storage

We assume the storage medium to be some form of local disk. Such a disk can be divided into slices/partitions, in which case it is said to be partitioned.

The slice reserved for Plan 9 (or the whole unpartitioned disk) is itself divided into Plan 9 partitions. A standard Plan 9 installation taking the whole disk and using *fossil(4)* as the root file sytem layouts the disk as follows (Begin and End in sector units):

The code implementing the solution was initially written as part of the *Google Summer of Code* 2009 program.

Name	Start	End	Description
9fat	63	–	Plan 9 kernels and boot configuration
nvram	–	–	non–volatile ram for PCs
fossil	–	–	*fossil(4)*
swap	–	–	swap area

3.2. Boot

After the Power On Self Test, the BIOS loads sector zero of the disk into physical memory address 0x7C00 and jumps to that location. If the disk is partitioned, that sector will contain the Master Boot Record (MBR). The MBR searches the master partition table for the active partition, loads that partition's Partition Boot Sector (PBS) into 0x7C00, and jumps there. If the disk is not partitioned, sector zero is the PBS itself.

3.3. PBS

The partition boot sector starts by jumping over its Boot Parameter Block (BPB) [2]. Its BPB ID field is edited by *format(8)* to contain the starting sector of the root directory of the FAT filesystem to where it is being installed. That directory is searched for a file named '9LOAD   '. If the desired file is found, the PBS calls BIOS interrupt 13 [4] to read the file's contents to memory 0x1000 physical and far jumps there.

PBS is hardcoded as a FAT boot sector and only understands FAT filesystems. Then it can only be used if a FAT is present at the beginning of the Plan 9 disk slice. Because of the x86 segmentation used by the PBS and the BIOS interrupt, *9load(8)* has a size limit of approximately 1MB.

3.4. 9load(8)

9load(8) is the PC bootstrap program: an auxiliary kernel with its own source code tree. Its main purpose is running a Plan 9 kernel. For this purpose to be met, *9load(8)* needs to a) enable 32-bit protected mode, b) load boot configuration, and c) find and load a kernel.

Plan9.ini(8) is the boot configuration file for PCs. *9load(8)* probes storage media searching for files plan9.ini and plan9/plan9.ini. When a file is found, it reads at most 100 configuration lines in the form name=value, storing them in memory at CONFADDR (0x1200) in order for the loaded kernel to read.

The bootfile line in *plan9.ini(8)* may contain the kernel path; if no such line is found, a prompt is presented for the user to type the desired path. The kernel must be either in *a.out(6)* or ELF format and can be gzip compressed. *9load(8)* reads the kernel's text segment to virtual 0xF0100000 and the data segment to the first page-aligned address after the text segment's end. Everything in place, *9load(8)* jumps to the kernel entry point at virtual 0xF0100020.

The way *9load(8)* works shows some limitations. Except for bootfile, the only way to set boot configuration is by using *plan9.ini(8).* We recognize that having a permanent configuration is a valuable feature, but forcing the user to edit a file everytime she wants to experiment with boot parameters does not seem to be optimal. In addition, the bootstrap program requires *plan9.ini(8)* and the kernel to be in a FAT partition.

A minimal device driver and filesystem infrastructure is needed for *9load(8)* to do its job. Since its source code is not the same as the other kernels, the exiting structure of the later can not be enjoyed by the former. This leds to duplicated effort, where drivers need be ported from kernels to *9load(8)* if kernel supported hardware is to be used in the boot process.

3.5. Stock Kernels

Plan 9 kernels assume to some degree that they are bootstrapped by *9load(8).* In any setup, at least 32-bit protected mode is expected to be turned on in the processor.

If *draw(3)* is going to be used, the kernel will rely on *9load(8)* to have setup part of VGA configuration; and if APM is needed, again the kernel will take it for granted from the bootstrap program. Even that *sd(3)* needs to have an in-memory table of partitions it does not parse disks for that information, expecting that *9load(8)* has done the parsing and stored the table in a CONFADDR line.

3.6. boot(8)

Besides being linked in the kernel image itself, *boot(8)* is mounted at /boot/boot and is the first user program to run. It connects to the file server specified by the user (via *plan9.ini(8)* or prompt) and mounts it as the namespace root. It then spawns a new process and run *init(8).*

Boot(8) has limitations akin to *9load(8)'s.* It does not fully enjoy the features provided by the system. As an example, the namespace root, if local, must be *kfs(4)* or *fossil(4);* if another filesystem is needed, routines particular to *boot(8)* must be written even if the supporting programs already exists in Plan 9.

4. 9null

In order to address the mentioned limitations, we wrote a new PBS, modified *boot(8)* and created a new kernel configuration. To the effort under which this work was done we gave the name 9null.

Initially the effort was only to remove *9load(8)* and boot a kernel directly. Russ Cox did solve part of the problem by writing a minimal bootstrap program that would load a kernel linked with itself [1]. His solution still left us a) the need for *plan9.ini(8),* b) the need for the kernel and *plan9.ini(8)* to be on 9fat, and c) the need for the local root to be either *kfs(4)* or *fossil(4).* We solved a) and c) with modifications to *boot(8),* and b) with a new PBS.

4.1. pbs32.s

Our new PBS is pbs32.s. Its first task is to make the switch to 32-bit protected mode so that it can address the whole address space. As a side-effect, the kernel size limit which existed in the old PBS vanishes.

The sectors of the Plan9 slice on disk are layouted as follows:

```
Sector Description
0      Partition Boot Sector
1      Plan 9 partition table
2..k   Space reserved for kernel (and possibly configuration)
k..n   data
```

Pbs32.s parses the master partition table, if any, to find the first sector of the Plan 9 slice; if the table is empty, Plan 9 sector zero coincides with sector 0 of the disk. The PBS then loops using ATA commands to read disk sectors and checking them for the *a.out(6)* signature; if such a sector is found, the file is read to physical memory 0x00100000 with proper alignment guaranteed. At last, a jump is made to the kernel entry point at 0x00100020 physical.

Such approach poses the need for the kernel to be contiguously placed on disk. On the other hand, since it only knows about disk sectors, it is filesystem agnostic and the space reserved for kernel storage does not even need to have a filesystem at all.

4.2. 9pcload

9pcload is the kernel created to be loaded directly by pbs32.s. It consists of minor additions to the pcf configuration so as to help the modifications made in *boot(8).* The biggest difference is that *rc(1)* was added to /boot.

4.3. Boot(8) modifications

Boot(8) was modified in three places. 9pcload, in cooperation with /sys/src/9/boot/mkboot, sets a global variable pcload to 1 that allows *boot(8)* to know if it is booted by pcload.

In the case 9pcload started *boot(8), plan9.ini(8)* is scanned, loaded, and the configuration added to #ec for the next kernel to use; *boot(8)* asks for a kernel to be booted – instead of the root file server – and use *reboot(8)* to reboot into it. If '!' is given as the kernel, *rc(1)* is started and the boot process may be carried manually. For the other kernels, *boot(8)* behaves as described in 3.6.

5. Open questions and work in progress

In order for 9null to fully replace the current boot process, there are some questions that need to be addressed and work that need to be finished.

Pbs32.s uses ATA commands to read sectors from disk, so using it on floopies is not supported. Since a great part of the machines comes with no floppy drive anymore, that does not seem to be a real problem. In any case, the old PBS can still be used when booting from floppy.

The new PBS does not support compressed kernels, though this only seems to be a problem when booting from floppies. Since those are not supported, it can be regarded as a non–issue.

Boot(8) is being rewritten to be minimal enough so as to let *rc(1)* carry the boot process. This raises the possibility for *boot(8)* to enjoy the full range of Plan 9 services.

At last, we have not put the effort to solve the problem of PXE booting but understand that, in order to preserve Plan 9 principles, methods of booting remotely must be supported. We believe that it is even paramount that the new boot process be as agnostic as possible to the location of its kernel and root.

6. Conclusions

We have managed to write a new boot process for Plan 9 in which a kernel is directly loaded by the early, sometimes called first stage, bootloader. Even not ready for production use yet, it shows advantages over the old boot process, being at the same time simpler and more general.

Our hope then is that this work will allow more users to boot Plan 9.

7. References

[1] R. Cox, *9fans archives*, `http://9fans.net/archive/2005/12/81`, 2005

[2] Microsoft® Corporation, *Microsoft Extensible Firmware Initiative FAT32 File System Specification*, `http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx`, Version 1.03, December 6, 2000

[3] *Plan 9 Manuals*, current edition published online at `http://plan9.bell-labs.com/sys/man`

[4] Intel® *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume 3A: System Programming Guide, Part 1 and Volume 3B: System Programming Guide, Part 2

Devsd Refresh

Erik Quanstrom
quanstro@quanstro.net

ABSTRACT

The Plan 9 *sd*(3) interface provides uniform access for disk-like devices for Plan 9. Its interfaces have proven fairly robust. With no incompatible changes, features like full 48-bit (ATA) and 64-bit (SCSI), SGPIO[1]/SES-2[2] enclosure management lights, and centralized SCSI CDB to ATA translation have been straightforward to add. Nonetheless, *sd* assumes fixed rather than hot-swappable drives and uses a intricate drive lettering scheme that may require some client-visible changes in the future.

Introduction

Although *sd*(3) stands for storage device, the name belies its SCSI roots. Nonetheless, it has proved quite robust in the face of changing requirements and easily extensible. The addition of several new hot-pluggable SATA and combined mode SATA/SAS controllers has stretched the model a bit. New drives and controllers already can break the 32-bit LBA barrier. Certainly this will be common in the future. Revisions to the ATA standard have introduced tricky new power-saving modes. We will outline the changes that were necessary to support these new drivers, the changes to the drivers themselves and what needs to be addressed in the future.

Raw ATA Support

While Plan 9 allows user-level commands to send raw SCSI commands directly to devices through the `raw` file, there is no such facility for ATA. Drives using the ATA command set get by by emulating a small set of SCSI commands. As outlined in the *ATA au Naturel*[3] paper, support for raw ATA commands was added without disturbing existing SCSI functionality. This required an extra *ataio* function and added about 20 lines of code to the generic device.

Full LBA Support

Full LBA support has been limited at several points to 32 bits. This limits drive size to 2TB with 512-byte sectors. With drive sizes currently at 2TB and chipset-level RAID that can combine drives, this is becoming a serious limitation. While *sdaoe*(3) sidesteps many of these issues, the other drivers have not been addressed.

These 32-bit limitations are in part due to the translation to SCSI in `pc/sdscsi.c`. The current ATA drivers all translate IO requests into SCSI CDBs and then translate them from SCSI to the original LBA and number of sectors. So SCSI limitations also limit ATA. Finally, ATA drivers are unable to translate `READ/WRITE (16)`, required for LBAs

larger than 32 bits, back into LBA format.

The initial step, generating the 16-byte IO commands when necessary was taken several years ago. However, drivers using `scsionline` were still limited to 32-bits due to the use of `GET CAPACITY (10)`. It was straightforward to detect overflow and issue a 16-byte command. As we consider the drivers, the translation from CDB to LBA in the various ATA drivers (*sdaoe* excepted) do not translate `READ/WRITE (16)`. Rather than fix them all individually, a new function *sdfakescsirw* was added which converts a CDB back into an LBA, number of sectors and if it is a read or write request. This fixed the 32-bit limitation and removed a number of duplicate (and not entirely compatible) copies of the same code.

For many drivers, it does not make sense to translate from LBA to CDB and back again. The *sdiahci* driver was modified to take advantage of this. A new *ahcibio* function was written that uses the LBA and sector count directly. In the case of ATAPI devices only, a SCSI CDB is generated using *scsibio*. For *sdaoe* the new *aoebio* is even simplier. Since the AoE driver does not support ATAPI, a CDB is never generated.

Unfortunately, a *rio* function must still be provided to support basic *scuzz(8)* functionality. This limits the amount of code that can be eliminated.

Lights

SES-2 and SGPIO are standards defining how to interact with a drive enclosure. Features vary widely by enclosure and HBA. It is possible to control up to three lights per drive (a green activity light, an amber locate light and a red fail light). In addition, a number of sensor readings may be available.

Rather than add add-hoc functionality to *sd*, it was decided to pass SGPIO/SES-2 commands through via control files. It would have been possible to simply reuse the `sd`*XX*`/ctl` file, but this would have made detecting and parsing the status of the lights cumbersome. Instead, a facility to add extra control files to a device was added. A single function was added to register a new control file.

For SGPIO/SES-2 lights, a `led` control file was added. Reading this file returns the current slot state. This is one of normal, rebuild, locate, spare, or fail. Writing a new state to the `led` file sets that slot to the given state. For example

```
chula# cat led
normal
chula# echo fail>led
```

sets the red fail light on red and turns on the backplane alarm. To reset the alarm,

```
chula# echo normal>led
chula# cat led
normal
```

It is envisioned that this functionality could be harnessed by *fs*(3) or *smart*(8) to automaticly announce the state of drives.

Since SGPIO/SES-2 also support temperature and fan sensors, additional control files could be added to support them.

Autosense

There are two ways for SCSI commands to return data. The method used depends on the HBA and the SCSI transport protocol. Early SCSI standards required the REQUEST SENSE to return sense (error) information. Autosense returns the sense data along with the response in a transport-dependent manner. When autosense is used, unsolicited REQUEST SENSE commands will return no sense, as sense data is returned only once. SAM-3[4] makes autosense mandatory. However the protocol provided by the raw file is not autosensing; it requires an explicit REQUEST SENSE. To get around this, *sd* uses the flag SDnosense to instruct the driver to act as if autosense had been disabled. Since this forces an additional burden on most modern drivers, the technique used by *sdmylex* of saving sense data until a REQUEST SENSE command is issued was moved to *sd*. Conversion to the new system remains unfinished as no working machines capable of driving available and supported cards were available for testing. The old sense-data saving code will simply be unused.

Bits

A few odds and ends were also changed. According to the *sd* manual page, the 10th unit attached to controller 0 should be named sd0a. Unfortunately, it was named sd010. In this case, the manual was considered authoritative and the code was "fixed."

It had also been impossible for anyone other than *eve* to read the sdctl file. Since the list of of controllers is not a security concern, the ability for *eve* to grant looser permissions was added in line with other *sd*-generated files. In addition, the error messages were sharpened so that setting unsettable information with *wstat* is flagged as an error.

There was also a vendor-specific MODE SENSE response which could be used to return the IDENTIFY (PACKET) DEVICE data. It appeared that this wormhole was unused, except by *scuzz*(8). It was removed since this data is accessible more conventionally via *atazz*(8).

The sdctl file is now generated so that there is exactly one line per drive letter. Formerly, a line was generated only if the driver provided an appropriate *rtopctl* function. *Sd* now supplies a basic entry for drivers lacking this function. This simplifies unit enumeration by spelling out each controller prefix. For each prefix, only 16 units need to be probed.

Drivers

The following drivers have received significant work, or are new: *sdaoe*, *sdata*, *sdiahci*, *sdloop*, *sdorion*, *sdmv50xx*. The *sdloop* (loopback) and *sdorion* drivers are new. All were converted to use libfis. Print statements were audited so drivers properly identify themselves in console messages.

The *Sdata* driver now uses the standard *sdsense*, *sdfakescsi* and *sdfakescsirw* functions. Additionally, a bug that caused Intel ICH south bridges to hang the system on boot was fixed.

The AHCI driver was updated to revision 1.3 of the specification[5] and power management support was added. Support for sector sizes other than 512 bytes was added. Disabled ports, staggered spinup, device discovery ATAPI devices have all been fixed. PATA bridged drives such as SATA Disk on Modules (DOMs) are now supported. Several AMD SB6xx bugs were fixed and support for VIA, nVidia, and JMicron devices was added.

Further Work

While all of the changes to *sd* could use further polish, there are two areas that have been ignored: hot-pluggable devices and drive enumeration. Hot-pluggable devices are a rough fit in *sd* because they may appear at any time and disappear at any time. The parallel SCSI drivers only create units where a drive is detected at boot time. The IDE driver takes the same approach. Since it is assumed that drives are not hot-pluggable, this makes sense. (As long as the IDE driver is actually presenting PATA drives, that is.) For SATA and SAS, this approach doesn't work. Either device directories must be created dynamicly, or a directory for each port must be created on boot. The later approach was taken, though this may need revisiting in the face of SAS expanders and SATA port multipliers, which may have tens of thousands of ports.

Another consequence of hot-pluggable drives is that it's hard to know how long to wait at boot for drives to appear. In a large system with many drives, it's not uncommon for drives to straggle in several minutes after the operating system gains control. Clearly it would be unacceptable to wait two minutes for each drive. But if that drive contains your root filesystem, you need to wait. The current kludge is for *9load* to use a heuristic algorithm to spin up the drives it sees. This seems to be working well in practice, but does not appear to be a satisfactory solution.

The *verify* and *online* model for bringing a unit online seem at odds with hot-plug devices. An experimental version of *sd* was built that combined the initial *pnp*, *enable* and *verify* functions into *pnp*. This allows drive discovery to take place in parallel and reduces the amount of state the controller needs to carry. The *online* function was replaced with *ready*. This function returns 1 if the drive is ready for access. The process of bringing the drive online is done asynchronously by a background process. As this is already a requirement for hot-pluggable drives, this requires no addition code. While this approach has worked well and simplified the code, it still doesn't answer the question of when to wait for the drive to be ready and for how long.

Currently *sd* picks meaningful drive letters. On a pc, `sdC0` and `sdD0` are, as expected, the primary IDE master and the secondary IDE master. However, if these same drives are not given the IDE legacy IO ports, they would be labeled `sdE0` and `sdF0`. Other ATA drives start with drive letter 'E.' SCSI drives start with the first controller being named `sd0`. The new Orion driver can control either SATA or SAS drives. The only solution in the current scheme is to start with a new primary letter — 'a' was chosen. My primary development machine has drive letters a, C—F, and I. This is a somewhat unwieldy situation. It may make sense to enumerate units sequentially. So instead of having units `sda0` and `sdE0` one would have `sd00` and `sd01`. If it is worth preserving the ability to associate drive letters with controllers, sdctl can continue to provide the mapping for interested applications. Dynamicly configured drives could read their assigned letters back.

Abbreviated References

[1] Serial GPIO, published online at
`ftp://ftp.seagate.com/sff/SFF-8485.PDF`.

[2] SCSI Enclosure Services - 2 (SES-2), published online at
`http://www.t10.org/ftp/t10/drafts/ses2/ses2r19a.pdf`

[3] E. Quanstrom "ATA au Naturel", proceedings of the International Workshop on Plan 9, October, 2009.

[4] SCSI Architectural Model - 3 (SAM-3), published online at

http://www.t10.org/cgi-bin/ac.pl?t=f&f=sam3r14.pdf
[5] Advanced Host Controller Interface (AHCI) 1.3, published online at
http://download.intel.com/technology/serialata/pdf/rev1_3.pdf

ATA au Naturel

Erik Quanstrom
quanstro@quanstro.net

ABSTRACT

The Plan 9 *sd*(3) interface allows raw commands to be sent. Traditionally, only SCSI CDBs could be sent in this manner. For devices that respond to ATA/ATAPI commands, a small set of SCSI CDBs have been translated into an ATA equivalent. This approach works very well. However, there are ATA commands such as SMART which do not have direct translations. I describe how ATA/ATAPI commands were supported without disturbing existing functionality.

Introduction

In writing new *sd*(3) drivers for plan 9, it has been necessary to copy laundry list of special commands that were needed with previous drivers. The set of commands supported by each device driver varies, and they are typically executed by writing a magic string into the driver's `ctl` file. This requires code duplicated for each driver, and covers few commands. Coverage depends on the driver. It is not possible for the control interface to return output, making some commands impossible to implement. While a work around has been to change the contents of the control file, this solution is extremely unwieldy even for simple commands such as `IDENTIFY DEVICE`.

Considerations

Currently, all *sd* devices respond to a small subset of SCSI commands through the raw interface and the normal read/write interface uses SCSI command blocks. SCSI devices, of course, respond natively while ATA devices emulate these commands with the help *sd*. This means that *scuzz*(8) can get surprisingly far with ATA devices, and ATAPI (*sic.*) devices work quite well. Although a new implementation might not use this approach, replacing the interface did not appear cost effective and would lead to maximum incompatibilities, while this interface is experimental. This means that the raw interface will need a method of signaling an ATA command rather than a SCSI CDB.

An unattractive wart of the ATA command set is there are seven protocols and two command sizes. While each command has a specific size (either 28-bit LBA or 48-bit LLBA) and is associated with a particular protocol (PIO, DMA, PACKET, etc.), this information is available only by table lookup. While this information may not always be necessary for simple SATA-based controllers, for the IDE controllers, it is required. PIO commands are required and use a different set of registers than DMA commands. Queued DMA commands and ATAPI commands are submitted differently still. Finally, the data direction is implied by the command. Having these three extra pieces of information in addition to the command seems necessary.

A final bit of extra-command information that may be useful is a timeout. While *alarm*(2) timeouts work with many drivers, it would be an added convenience to be able to specify a timeout along with the command. This seems a good idea in principle, since some ATA commands should return within milli- or microseconds, others may take hours to complete. On the other hand, the existing SCSI interface does not support it and changing its kernel-to-user space format would be quite invasive. Timeouts were left for a later date.

Protocol and Data Format

The existing protocol for SCSI commands suits ATA as well. We simply write the command block to the raw device. Then we either write or read the data. Finally the status block is read. What remains is choosing a data format for ATA commands.

The T10 Committee has defined a SCSI-to-ATA translation scheme called SAT[4]. This provides a standard set of translations between common SCSI commands and ATA commands. It specifies the ATA protocol and some other sideband information. It is particularly useful for common commands such as `READ (12)` or `READ CAPACITY (12)`. Unfortunately, our purpose is to address the uncommon commands. For those, special commands `ATA PASSTHROUGH (12)` and `(16)` exist. Unfortunately several commands we are interested in, such as those that set transfer modes are not allowed by the standard. This is not a major obstacle. We could simply ignore the standard. But this goes against the general reasons for using an established standard: interoperability. Finally, it should be mentioned that SAT format adds yet another intermediate format of variable size which would require translation to a usable format for all the existing Plan 9 drivers. If we're not hewing to a standard, we should build or choose for convenience.

ATA-8 and ACS-2 also specify an abstract register layout. The size of the command block varies based on the "size" (either 28- or 48-bits) of the command and only context differentiates a command from a response. The SATA specification defines host-to-drive communications. The formats of transactions are called Frame Information Structures (FISes). Typically drivers fill out the command FISes directly and have direct access to the Device-to-Host Register (D2H) FISes that return the resulting ATA register settings. The command FISes are also called Host-to-Device (H2D) Register FISes. Using this structure has several advantages. It is directly usable by many of the existing SATA drivers. All SATA commands are the same size and are tagged as commands. Normal responses are also all of the same size and are tagged as responses. Unfortunately, the ATA protocol is not specified. Nevertheless, SATA FISes seem to handle most of our needs and are quite convenient; they can be used directly by two of the three current SATA drivers.

Implementation

Raw ATA commands are formatted as a ATA escape byte, an encoded ATA protocol `proto` and the FIS. Typically this would be a H2D FIS, but this is not a requirement. The escape byte 0xff, which is not and, according to the current specification, will never be a valid SCSI command, was chosen. The protocol encoding `proto` and other FIS construction details are specified in `/sys/include/fis.h`. The `proto` encodes the ATA protocol, the command "size" and data direction. The "atazz" command format is pictured in Figure 1.

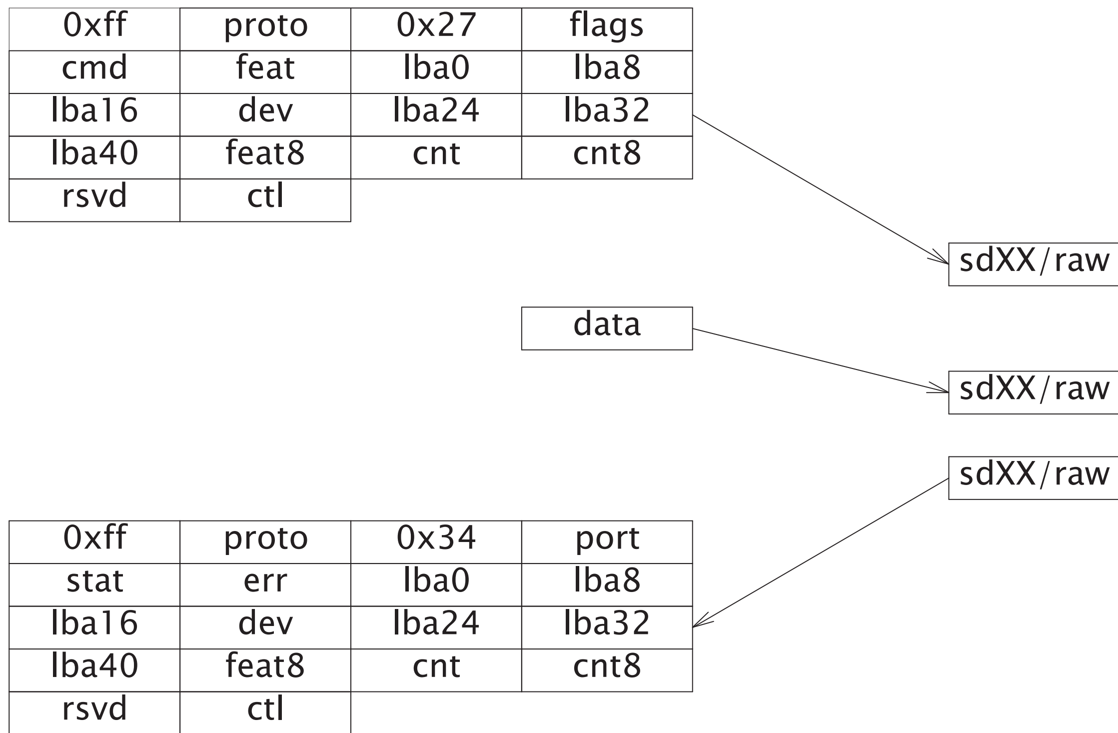


Figure 1

Raw ATA replies are formatted as a one-byte sd status code followed by the reply FIS. The usual read/write register substitutions are applied; ioport replaces flags, status replaces cmd, error replaces feature.

Important commands such as `SMART RETURN STATUS` return no data. In this case, the protocol is run as usual. The client performs a 0-byte read to fulfill data transfer step. The status is in the D2H FIS returned as the status. The vendor ATA command 0xf0 is used to return the device signature FIS as there is no universal in-band way to do this without side effects. When talking only to ATA drives, it is possible to first issue a `IDENTIFY PACKET DEVICE` and then a `IDENTIFY DEVICE` command, inferring the device type from the successful command. However, it would not be possible to enumerate the devices behind a port multiplier using this technique.

Kernel changes and Libfis

Very few changes were made to devsd to accommodate ATA commands. the `SDreq` structure adds `proto` and `ataproto` fields. To avoid disturbing existing SCSI functionality and to allow drivers which support SCSI and ATA commands in parallel, an additional `ataio` callback was added to `SDifc` with the same signature as the existing `rio` callback. About twenty lines of code were added to `port/devsd.c` to recognize raw ATA commands and call the driver's `ataio` function.

To assist in generating the FISes to communicate with devices, `libfis` was written. It contains functions to identify and enumerate the important features of a drive, to format H2D FISes And finally, functions for *sd* and *sd* -devices to build D2H FISes to capture the device signature.

All ATA device drivers for the 386 architecture have been modified to accept raw ATA commands. Due to consolidation of FIS handling, the AHCI driver lost 175 lines of code, additional non-atazz-related functionality notwithstanding. The IDE driver remained

exactly the same size. Quite a bit more code could be removed if the driver were reorganized. The mv50xx driver gained 153 lines of code. Development versions of the Marvell Orion driver lost over 500 lines while `libfis` is only about the same line count.

Since FIS formats were used to convey commands from user space, `libfis` has been equally useful for user space applications. This is because the *atazz* interface can be thought of as an idealized HBA. Conversely, the hardware driver does not need to know anything about the command it is issuing beyond the ATA protocol.

Atazz

As an example and debugging tool, the *atazz*(8) command was written. *Atazz* is an analog to *scuzz*(8); they can be thought of as a driver for a virtual interface provided by *sd* combined with a disk console. ATA commands are spelled out verbosely as in ACS-2. Arbitrary ATA commands may be submitted, but the controller or driver may not support all of them. Here is a sample transcript:

```
az> probe
/dev/sda0   976773168; 512  50000f001b206489
/dev/sdC1   0; 0    0
/dev/sdD0   1023120; 512    0
/dev/sdE0   976773168; 512  50014ee2014f5b5a
/dev/sdF7   976773168; 512  5000cca214c3a6d3
az> open /dev/sdF0
az> smart enable operations
az> smart return status
normal
az> rfis
00
34405000004fc2a00000000000000000
```

In the example, the `probe` command is a special command that uses `#S/sdctl` to enumerate the controllers in the system. For each controller, the `sd` vendor command `0xf0` (GET SIGNATURE) is issued. If this command is successful, the number of sectors, sector size and WWN are gathered and and listed. The `/dev/sdC1` device reports 0 sectors and 0 sector size because it is a DVD-RW with no media. The `open` command is another special command that issues the same commands a SATA driver would issue to gather the information about the drive. The final two commands enable SMART and return the SMART status. The smart status is returned in a D2H FIS. This result is parsed the result is printed as either “normal,” or “threshold exceeded” (the drive predicts imminent failure).

As a further real-world example, a drive from my file server failed after a power outage. The simple diagnostic SMART RETURN STATUS returned an uninformative “threshold exceeded.” We can run some more in-depth tests. In this case we will need to make up for the fact that *atazz* does not know every option to every command. We will set the `lba0` register by hand:

```
az> smart lba0 1 execute off-line immediate # short data collection
az> smart read data
col status: 00 never started
exe status: 89 failed: shipping damage, 90% left
time left: 10507s
shrt poll: 176m
ext poll: 19m
az>
```

Here we see that the drive claims that it was damaged in shipping and the damage occurred in the first 10% of the drive. Since we know the drive had been working before the power outage, and the original symptom was excessive UREs (Unrecoverable Read Errors) followed by write failures, and finally a threshold exceeded condition, it is reasonable to assume that the head may have crashed.

Stand Alone Applications

There are several obvious stand-alone applications for this functionality: a drive firmware upgrade utility, a drive scrubber that bypasses the drive cache and a SMART monitor.

Since SCSI also supports a basic SMART-like interface through the `SEND DIAGNOSTIC` and `RECEIVE DIAGNOSTIC RESULTS` commands, *disk/smart*(8) gives a chance to test both raw ATA and SCSI commands in the same application.

Disk/smart uses the usual techniques for gathering a list of devices or uses the devices given. Then it issues a raw ATA request for the device signature. If that fails, it is assumed that the drive is SCSI, and a raw SCSI request is issued. In both cases, *disk/smart* is able to reliably determine if SMART is supported and can be enabled.

If successful, each device is probed every 5 minutes and failures are logged. A one shot mode is also available:

```
chula# disk/smart -atv
sda0: normal
sda1: normal
sda2: normal
sda3: threshold exceeded
sdE1: normal
sdF7: normal
```

Drives `sda0`, `sda1` are SCSI and the remainder are ATA. Note that other drives on the same controller are ATA. Recalling that `sdC0` was previously listed, we can check to see why no results were reported by `sdC0`:

```
chula# for(i in a3 C0)
    echo identify device |
        atazz /dev/sd$i >[2]/dev/null |
        grep '^flags'
flags   lba llba smart power nop sct
flags   lba
```

So we see that `sdC0` simply does not support the SMART feature set.

Further Work

While the raw ATA interface has been used extensively from user space and has allowed the removal of quirky functionality, device setup has not yet been addressed. For example, both the Orion and AHCI drivers have an initialization routine similar to the following

```
newdrive(Drive *d)
{
	setfissig(d, getsig(d));
	if(identify(d) != 0)
		return SDeio;
	setpowermode(d);
	if(settxmode(d, d->udma) != 0)
		return SDeio;
	return SDok;
}
```

However in preparing this document, it was discovered that one sets the power mode before setting the transfer mode and the other does the opposite. It is not clear that this particular difference is a problem, but over time, such differences will be the source of bugs. Neither the IDE nor the Marvell 50xx drivers sets the power mode at all. Worse, none is capable of properly addressing drives with features such as PUIS (Power Up In Standby) enabled. To addresses this problem all four of the ATA drivers would need to be changed.

Rather than maintaining a number of mutually out-of-date drivers, it would be advantageous to build an ATA analog of `pc/sdscsi.c` using the raw ATA interface to submit ATA commands. There are some difficulties that make such a change a bit more than trivial. Since current model for hot-pluggable devices is not compatible with the top-down approach currently taken by *sd* this would need to be addressed. It does not seem that this would be difficult. Interface resets after failed commands should also be addressed.

Source

The current source including all the pc drivers and applications are available in the following *contrib*(1) packages on *sources*:
`quanstro/fis`,
`quanstro/sd`,
`quanstro/atazz`, and
`quanstro/smart`.

The following manual pages are included:
fis(2), *sd*(3), *sdahci*(3), *sdaoe*(3), *sdloop*(3), *sdorion*(3), *atazz*(8), and *smart*(8).

Abbreviated References

[1]*sd*(1), published online at
`http://plan9.bell-labs.com/magic/man2html/3/sd`

[2]*scuzz*(8), published online at
`http://plan9.bell-labs.com/magic/man2html/8/scuzz`

[3]T13 *ATA/ATAPI Command Set - 2*, revision 1, January 21, 2009, formerly published online at `http://www.t13.org`.

[4]T10 *SCSI/ATA Translation - 2 (SAT-2)*, revision 7, February 18, 2007, formerly published online at `http://www.t10.org`.

Plan 9's Universal Serial Bus

Francisco J. Ballesteros
nemo@lsub.org

ABSTRACT

The Universal Serial Bus is a complex and, therefore, a popular bus on personal computers and other devices. Many devices including disks, keyboards, mice, and network cards are attached to computers using it. The bus is a tree with hubs as nodes and devices as leafs and uses polling from the root of the tree, which is the bus controller. It permits hot plugging and removal of devices at any time. This paper describes the system software used on Plan 9 to drive the bus and its devices. How to use the software is not described here, but in the Plan 9 user's manual.

1. Introduction

The Universal Serial Bus (or USB) [1] is a standard bus developed by a set of corporations including Intel, Compaq, Microsoft, Digital, IBM, and Northern Telecom. It started in 1994 with version 1 supporting two transfer modes: Low speed transfers at 1.5 Mb/s and full speed transfers at 12 Mb/s. Version 2 was introduced on 2000. It defined, so-called, high speed transfers capable of 12 Mb/s. This is the version in use today and is what the software described on this paper can drive. Version 3 of the standard was released in 2008 introducing 5 Gb/s transfers (called superspeed transfers). However, as of today, there is no hardware in the market supporting this version of the standard.

The bus structure is certainly complex, when compared to other buses, mostly because of the requirements on the software. It is a made of a tree of devices with a host controller at the root, hubs implementing branches, and devices attached to leaves of the tree (to USB ports). There can be up to 127 devices attached to the bus including hubs.

USB devices are standarized into classes of devices, further divided into subclasses, and sets of devices speaking a particular protocol. Together, class, subclass, and protocol identify a device type, codified as a number known as a ''CSP''. In practice some devices belong to a ''vendor specific class'' that may contain any type of device, rendering the CSP useless. In this case vendor and product identifiers are the only choice to determine which one is the device at hand.

A device may be in fact a combination of different devices packaged together. For example, keyboard and mouse combos are packaged into a single device attached to a single USB port. In part because of this, in part to try to simplify the interaction of the software with devices, a device includes different addressable entities called *endpoints*, grouped into *interfaces*. Each interface is an administrative entity that has its own CSP and includes one or more endpoints. In our example, a keyboard and mouse combo may provide two interfaces (one for the mouse and one for the keyboard).

Initially, all devices include a zero endpoint used for configuration purposes. The setup endpoint is available as long as the device is attached. Other endpoints may be configured later by the software, as dictated by the device. These are grouped into

interfaces, which correspond to a function performed by the device. Each interface has also an associated CSP that identifies its type. In few words, an endpoint is an artifact used for I/O and has an associated CSP indicating its purpose.

As of today, the Plan 9 USB software supports version 2 of the bus including several drivers for disks, keyboards, mice, serial lines, ethernet cards, and KNX devices. It is likely that more drivers will be added in the future. The software is responsible for enumerating devices on the bus, configuring them, and providing interfaces to use them and perform actual I/O.

The enumeration process consists on detecting devices attached to the bus and assigning addresses to them. A newly attached device uses a well-known configuration address to permit the software performing the enumeration to reach the new device. A consequence is that only one device can be attached and configured at a time. Once the new device has been given an address, another port may be permitted to attach another device, which starts using the configuration address. Enumeration has to take into account that hot-plugging is supported by the bus so that devices may be attached and removed at any time.

Device configuration may not be trivial for some devices. This means that it is better to keep as much of the USB software as feasible outside of the kernel. At least, the part responsible for configuring devices. Configuration is generic in principle, because devices include data to describe themselves. However, for many devices it is necessary to perform specific configuration tasks, which may be complex as well. Once a device is configured, one or several data pipes (endpoints) are available for use to operate on the device.

For efficiency reasons it is desirable to keep the mechanism used for I/O within the kernel. The idea is that after a device driver has configured a device the kernel provides the actual mechanism for performing I/O and programs may perform `read` and `write` system calls to obtain and to send data. For some devices, which require following specific interaction protocols, this may not be possible and a driver must sit between the application and the I/O mechanism provided by the kernel.

Device combos have implications for device drivers. A single device driver must be responsible for the entire device, but several drivers may be required to deal with the different functions of the device. That is the case of `kb`, which handles mouse and keyboard combos and thus corresponds to two different drivers.

Last but not least, several USB devices may be required for use during the boot process. For example, keyboards and mice. This implies including them into the kernel as boot files. Embedding all these drivers into the kernel as separate programs replicates multiple times almost the same code for the C library and the 9P library, among other utilities. Some mechanism is necessary to avoid this duplication.

These are the requirements on the software to drive USB. In what follows we describe this software after a brief overview of the USB hardware and its protocol.

2. USB hardware and protocol

The purpose of USB is to perform transfers between the host controllers and devices attached to the bus (see figure 1). All details are described in the specification [1]. Here we introduce only the most relevant ones for understanding the rest of this paper. As the Plan 9 software does, we also deviate from the standard in the description, for simplicity.

There are three different controllers in use today. Two of them implement USB version 1: the Universal Host Controller Interface, or UHCI [4], and the Open Host Controller Interface, or OHCI [2]. The former was developed by Intel and the later was developed by Compaq, Microsoft, and National Semiconductor. Both are in use on current hardware.

The main difference between UHCI and OHCI is that the former does not provide any indication whatsoever regarding which transfer is responsible for an interrupt, while the later includes a more sensible interface to the software.

USB version 2 is implemented by the Enhanced Host Controller Interface or EHCI [3] (which like UHCI does not provide any mean to know which transfer is responsible for an interrupt). The EHCI controller is usually packaged along with one or more version 1 controllers, called companion controllers. This means that supporting USB 2.0 requires drivers for all three controllers. In version 3 an Extensible Host Controller Interface, or XHCI, includes a companion version 2 controller (therefore it can be expected that driving the four controllers will be necessary to drive USB 3.0). Its specification is not yet available.

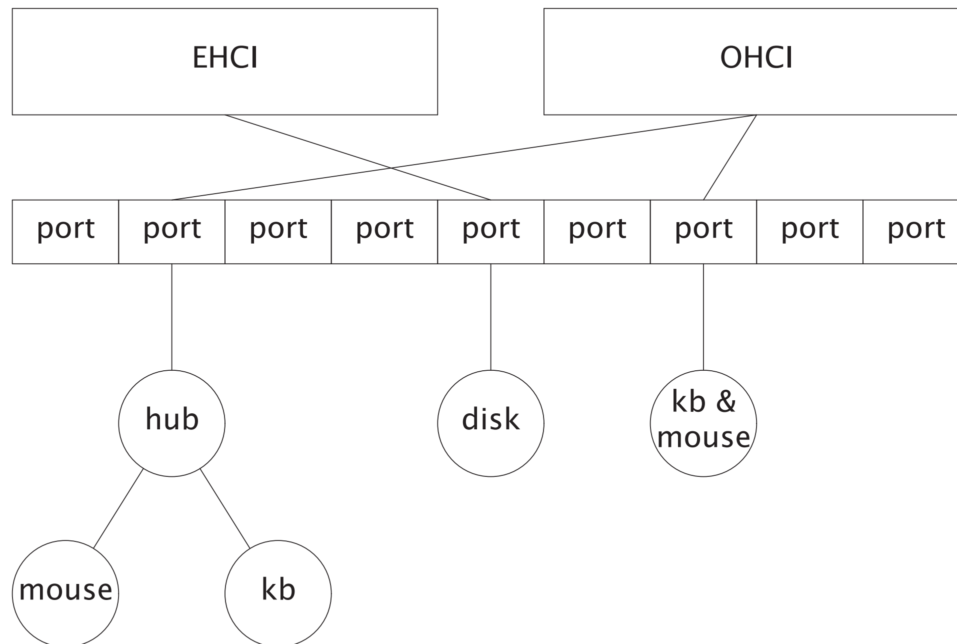


Figure 1. An example Universal Serial Bus.

Packaged with the controllers are included a series of USB ports, where devices and hubs may be attached (Hubs attach to a port and provide extra ports on the bus.) In theory the root of the device tree is a hub, called a root hub. In practice the root hub must be implemented by software. Most other systems do so. We tried not to do it to avoid a significant amount of software.

On a 2.0 USB system ports on the root hub (that is, included in the host controller) may be independently routed to either the 2.0 controller (EHCI) or to a companion USB 1.0 controller. This is done depending on the device speed. Full and low speed devices are routed to the companion controller and high speed devices are kept routed to the EHCI. For example, in the figure 1 a 1.0 hub is attached to the 2nd port. The companion OHCI is responsible for this USB 1.0 subset of the USB 2.0 bus.

Data transfers are implemented by the host controller, which is able to poll devices on the bus using TDMA and a communication protocol spoken on the USB wires. This happens both for input and for output. There are four types of transfers used to talk to endpoints, but it must be noted that an endpoint is capable of only one transfer type:

- *Control transfers* are RPCs to the device. Among other things, they permit configuring the device and recovering from soft errors.
- *Bulk transfers* are unidirectional transfers intended to send or receive a significant amount of data (e.g., 512 Kbytes).

- *Interrupt transfers* are not really interrupts, but unidirectional and small data transfers (e.g., 8 bytes). They are asynchronous in nature but are not so in practice (devices are polled). Ironically, to achieve the asynchronous character of the transfer the device has to be polled often, meaning that these transfers are considered synchronous by USB controllers.
- *Isochronous transfers* are unidirectional transfers that must be performed on a timely basis. For example, audio output. They sacrifice error checking in favor of timeliness.

There are forbidden combinations, but in general, for each transfer type we have full, low, and high speed variants of the transfer type. Full and low speed ones are supported by all three controllers; High speed ones are supported only by EHCI. Bulk, interrupt, and isochronous transfers correspond one to one to transfers understood by the controller. Control transfers do not.

A control transfer is actually a sequence of two or more transfers. First, a *setup* transfer asks the device to perform a control request, perhaps requiring a data transfer from the host to the device or from the device to the host. Second, if the request requires exchanging data between the controller and the device, a second transfer exchanges data (this may be more than one transfer if not all data fits in a USB packet). Third, an empty data transfer from the device to the controller reports the status for the entire control transfer.

USB devices must understand a set of standard control requests, described in chapter 9 of [1]. However, many devices implement non-standard requests (or perhaps standarized requests that are specific of the device class). The most popular control requests are those that retrieve *descriptors* from devices. These are standarized binary descriptions for devices and device features. Some descriptors have to be implemented by all devices while others are supported or not depending on the device. Obtaining device descriptors is necessary to learn how many endpoints there are, what their types are, and how to build a request to activate them and enable I/O.

On the bus a transfer requires multiple packets. Chapter 8 of [1] describes the protocol. Most details are uninteresting but the addressing and the data acknowledge mechanism are important for the software.

Bus addresses are made out of a device address and an endpoint address. That is, the addressable entities are the endpoints and not the devices. Device addresses are assigned by the software. Endpoint addresses are dictated by the hardware.

All devices have initially (after attachment to the bus) an endpoint with address zero, known as the device's setup endpoint. It is always an endpoint using control transfers and thus it is also known as the control endpoint. Its main purpose is to configure the device.

Depending on the type of device other endpoints will be available once the device has been configured. For example, a mouse is likely to have an interrupt input endpoint to report mouse events, separate from the control endpoint. In the same way, a disk drive usually has two bulk endpoints (one to write data to the device and one to read data from it) although it may have a single input/output bulk endpoint. Addressing is important here because information supplied by the device may specify that a given endpoint address is the one to use for a particular task. Therefore, endpoint addresses must be exposed to the USB software.

Data may be lost during transfers. As an acknowledge mechanism to recover from this situation, successive packets sent from the controller to a given endpoint use one out of two different values alternatively (the same happens on the other direction, from the device to the controller). This is called the *data toggle* bit and toggles between two values called *data0* and *data1*. How this is codified varies from one controller to another (and from one transfer type to another). Also, some high speed transfers use a

different acknowledgment mechanism to be able to send multiple transfers on a single bus time frame.

The important point for drivers is that due to errors the controller and the device may be out of sync. At this point the device will simply refuse any transfer with the wrong toggle. To continue operating the device, data toggles must be synchronized again. Errors causing a cease of device I/O are called device *stalls*. Recovering from a stall is called *unstalling* the device. But note that what stalls is really an endpoint and not necessarily the entire device.

3. Plan 9's USB overview

The Plan 9 USB software is organized as depicted in figure 2. The kernel includes the *usb*(3) device driver, known as `#u`, in charge of providing I/O to endpoints present on the bus. A suite of user programs provide for everything else. One program, `usbd`, is in charge of bus enumeration. Remaining programs are USB device drivers. Both `usbd` and device drivers employ a library, called the USB library, for common tasks and data structures.

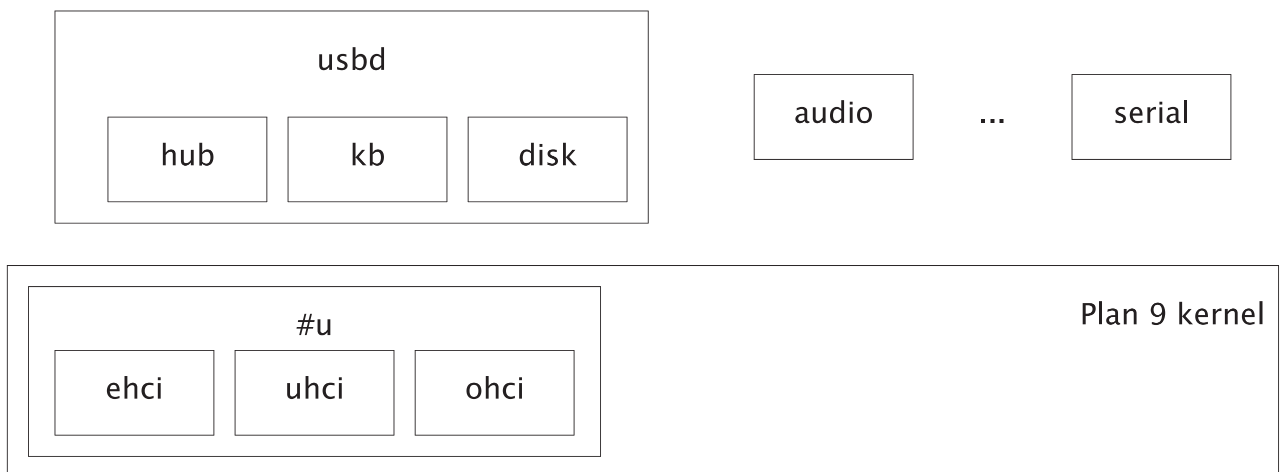


Figure 2. Organization of the USB software.

As figure 2 shows, drivers may be embedded into `usbd` (like `kb` and `disk` in the figure) or kept as separate programs (like `audio` and `serial` in the figure.) The `hub` driver is built into `usbd` and may not be started separately, because hubs play an important role in bus enumeration.

The `#u` device is responsible for initializing the host controllers and abstracting I/O mechanisms used on the bus. Its external interface deviates from the standard in an attempt to simplify things for users and device drivers. The only objects supplied by `#u` are *endpoints*. An endpoint represents a communication channel to a device in the bus (actually, to a bus address).

Figure 3 shows an example file tree provided by `#u`. Each endpoint is represented by a directory that includes two files: `data` and `ctl`, similar to a network connection. The former is used to perform actual I/O and the later is used to issue control requests for the endpoint (not to be confused with USB control transfers). Endpoints are named `ep`*N*`.`*M*, where *N* is the device address and *M* is the endpoint number. Endpoints with name `ep`*N*`.0` are therefore control endpoints.

At boot time `#u` provides one endpoint per root hub. Of course there is no such thing as root hubs, but the user program `usbd` uses these initial endpoints to enumerate the bus. Requests sent through them to query status of ports and to enable them are intercepted by the software and implemented by relying on the host controller interface.

Apart from this feature, there is no software implementation for USB root hubs on Plan 9.

For each new device discovered on the bus `usbd` creates another endpoint in `#u` for its *setup* endpoint. This is used by `usbd` to perform part of the configuration for the device and by the device driver to complete the configuration and, perhaps, to issue other requests to the device. As a result of the configuration, other endpoints are created for the device.

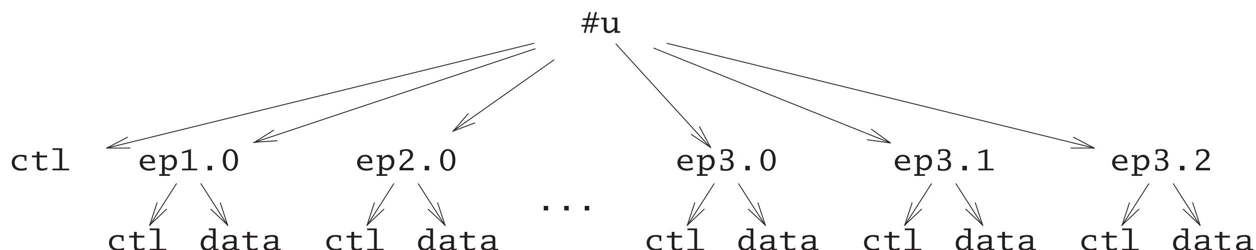


Figure 3: Example USB file tree.

3.1. Kernel drivers

Usb(3) is the driver responsible for providing I/O access to the bus, by providing the endpoint interface described above. Internally, it is built out of a generic device driver that provides the file interface and three different controller drivers that are linked into the former. The interface of the generic part to the rest of the kernel is similar to that in other devices, a `Devtab` structure. The interface between the generic device driver and the three controller-specific drivers deserves some comments.

In general, all three controllers perform the same tasks. They provide a register interface to query and change the status of USB ports on the root hub and use a memory mapped interface with data structures used to perform I/O for all supported transfer types. These data structures may be fancy and include binary trees.

Although much of the code could be shared between controllers (as done in other systems) the particularities of each controller get in the way. For example, how to indicate which data toggle to use for a transfer depends on the transfer type and also on the controller used. Who controls the toggles (the hardware or the software) also depends on the transfer type and on the controller. And the same can be said of many other details. Factoring out the shared code between different controllers would make it necessary to dive often into controller specific code (and data).

Instead of doing so, the Plan 9 `#u` device driver contains only a portable representation for endpoints to support the interface provided to user programs. It is up to each controller to look into the portable representation to configure controller specific data structures to match the portable representation. But for intercepting requests intended for root hubs, all the implementation for I/O is kept on the controller specific part of the code.

This is the portable representation for an endpoint, as kept in `#u`. All endpoints are kept in a global array and the index for each one is kept in the `idx` field. Most of the fields correspond to configuration information retrieved from the device by a user level program and supplied later to the kernel. Other files are used for bookkeeping and to synchronize access to the endpoint.

```
struct Ep
{
	Ref;				/* one per fid (and per dev ep for ep0s) */

	/* const once inited. */
	int	idx;			/* index in global eps array */
	int	nb;			/* endpoint number in device */
	Hci*	hp;			/* HCI it belongs to */
	Udev*	dev;			/* device for the endpoint */
	Ep*	ep0;			/* control endpoint for its device */

	/* configuration */
	QLock;				/* protect fields below */
	char*	name;			/* for ep file names at #u/ */
	int	inuse;			/* endpoint is open */
	int	mode;			/* OREAD, OWRITE, or ORDWR */
	int	clrhalt;		/* true if halt was cleared on ep. */
	int	debug;			/* per endpoint debug flag */
	char*	info;			/* for humans to read */
	long	maxpkt;			/* maximum packet size */
	int	ttype;			/* transfer type */
	ulong	load;			/* in µs, for a transfer of maxpkt bytes */
	void*	aux;			/* for controller specific info */
	int	rhrepl;			/* fake root hub replies */
	int	toggle[2];		/* saved toggles (while ep is not in use) */
	long	pollival;		/* poll interval ([µ]frames; intr/iso) */
	long	hz;			/* poll frequency (iso) */
	long	samplesz;		/* sample size (iso) */
	int	ntds;			/* nb. of Tds per µframe */
};
```

The setup endpoint for a device is used to talk to the device and therefore represents the entire device. There is no separate abstraction to represent a USB device. Instead, all endpoints keep in the ep0 field a link to the control endpoint representing the device. A shared data structure maintains per-device information and can be found linked at the dev field of any endpoint. It is declared as follows.

```
struct Udev
{
	int	nb;			/* USB device number */
	int	state;			/* state for the device */
	int	ishub;			/* hubs can allocate devices */
	int	isroot;			/* is a root hub */
	int	speed;			/* Full/Low/High/No -speed */
	int	hub;			/* dev number for the parent hub */
	int	port;			/* port number in the parent hub */
	Ep*	eps[Ndeveps];		/* end points for this device (cached) */
};
```

A device is mostly an address for the device, kept in nb, and a series of endpoints, kept in eps. Each endpoint has its own address, which matches the index in the per-device endpoint array.

Provided an endpoint, the driver for a controller is responsible for performing I/O on it. To do so, and to provide access to the ports in the controller, the driver must implement the following interface. Currently there are three implementations for EHCI, UHCI, and OHCI.

```
struct Hciimpl
{
        void      *aux;                          /* for controller info */
        void      (*init)(Hci*);                 /* init. controller */
        void      (*dump)(Hci*);                 /* debug */
        void      (*interrupt)(Ureg*, void*);    /* service interrupt */
        void      (*epopen)(Ep*);                /* prepare ep. for I/O */
        void      (*epclose)(Ep*);               /* terminate I/O on ep. */
        long      (*epread)(Ep*,void*,long);     /* transmit data for ep */
        long      (*epwrite)(Ep*,void*,long);    /* receive data for ep */
        char*     (*seprintep)(char*,char*,Ep*); /* debug */
        int       (*portenable)(Hci*, int, int); /* enable/disable port */
        int       (*portreset)(Hci*, int, int);  /* set/clear port reset */
        int       (*portstatus)(Hci*, int);      /* get port status */
        void      (*debug)(Hci*, int);           /* set/clear debug flag */
};
```

`Init` prepares the controller for operation. `Dump`, `seprintep`, and `debug` are used to control debug diagnostics. `Interrupt` is obviously the interrupt handler for the controller. All other functions are the interesting ones. They implement I/O according for endpoints described by the (portable) data structure shown above and are described in the following sections.

3.2. Hub ports

Ports on root hubs are handled by `portenable`, `portreset`, and `portstatus`. The common `usbread` and `usbwrite` functions intercept requests directed to root hubs to query or adjust the status for their ports. Instead of sending messages through the USB bus, `usbread` and `usbwrite` rely on port handling functions provided by the controller driver. By doing so, `usbd` may remain (mostly) unaware of the difference between root hubs and other hubs.

Instead of performing a full software emulation for root hubs, `#u` includes just a few USB requests (those calling the functions described here). `Usbd` tries not to use other hub features to avoid the need for a full emulation. However, some features are required to configure hubs for operation and thus are used on actual (non-root) hubs. This introduces into `usbd` a few places where the code takes different paths for root and secondary hubs, but the alternative would be to implement a full emulation of root hubs.

Other details necessary in practice to operate on root hubs (e.g., port power configuration) are dealt with in the controller initialization code. All other USB software remains unaware of them.

3.3. Input/Output

The suite provided by `epopen`, `epclose`, `epread`, and `epwrite` provides I/O through USB endpoints. The first two functions prepare the endpoint data structures for I/O and release them, respectively. To avoid resource consumption, an endpoint is kept open only while necessary (while the endpoint data file is open). At all other times, the controller driver keeps no state at all for the endpoint.

During `epopen` the generic description of the endpoint provided by the `Ep` data structure is consulted to configure the actual data structures used by the hardware. Therefore, there is no configuration interface between the generic and the controller specific software (other than the agreed-upon endpoint data structure).

One problem introduced by this scheme is that the physical device may retain configuration, such as protocol data toggles, between successive opens of the same endpoint. But closing an endpoint and opening it again must continue I/O from where it was

left at. This issue is addressed by arranging for epclose to save this information in the portable description of the endpoint, and by making epopen consult the saved state. The Ep structure has an aux field for the controller driver to use. But note that unless the endpoint is in use this field would be null, therefore, saved state is kept within Ep itself.

Epread and Epwrite perform input and output from the bus. In general, they switch on the endpoint transfer type and prepare a transfer of that type, but that is not the case for control transfers.

USB control transfers may require different transfers depending on the request, as said before. To make all control transfers look similar for the user, the interface provided to the user is a single write with the request (and any data sent to the device), perhaps followed by a single read to retrieve data requested by the previous write. The entire transfer is performed during the call to write. This permits retrieving the error status, which is sent by the device at the end of the data transfer, when the request is made. Any data retrieved from the device is kept in memory and given to the user if read is called next (before another write).

Processes using the endpoint must in any case coordinate their requests and thus, performing control transfers in this way does not introduce more race conditions than those existing in the user code.

As an aid, processes not coordinating are kept apart by the DMEXCL permission enforced by #u for endpoint data files. For those who care to look before using, the endpoint control file reports whether the endpoint is in use or not.

Epread and epwrite are responsible for timing out unresponsive devices, raising an error in that case. Arguably, they time out only control and bulk requests. In general these requests complete soon after being issued; Interrupt and isochronous requests do not. Timing out these requests in the kernel in a controlled way helps canceling requests only when the device seems to be not responding. User timeouts might occur with bad timing and confuse the device, which may be simply sending negative acknowledgements while busy doing other things.

Recently we have found some devices where bulk requests may not complete until further device activity. This suggests that only control requests should be timed out even though most devices need timeouts on bulk requests to operate properly with unresponsive devices.

Device drivers like audio and printers may exit after configuring their devices. Using the endpoint data files suffices for them at this point. The problem is that Plan 9 software expects to find these files on conventional names at /dev. As an aid, #u implements a control request capable of giving a second name (at #u) to an endpoint data file.

4. Bus enumeration and hot plugging

Bus enumeration is performed by a single process executing in usbd. This process starts by looking at endpoints existing at boot time. As said before a control endpoint is used to represent a device. The kernel device creates one such endpoint per root hub. From there on, usbd asks for the status for each port in the hub. When a device is connected to a port usbd asks the kernel to create a new endpoint for its control endpoint and, using it, retrieves device descriptors and performs several configuration requests. After configuration, if the device is known to be a hub usbd adds the endpoint to the list of hubs to poll, and polls it. Otherwise, if instructed to do so by its configuration file, usbd starts a new process to execute the device driver for the attached device.

Only one process must use a device at a time. Therefore it is important for usbd and the device driver to coordinate regarding access to USB devices. This problem has been avoided by preventing both usbd and a device driver to own the same device. Figure 4 depicts the typical scenario. In the figure time flows top-down and most of the time will be spent in the small dotted segments, which represent the device while in use. The figure shows only the attachment/detachment process.

Initially, the device is attached to the bus. After that, a port poll done by usbd will see that a device is connected to a port. After configuring the port, by issuing a reset signal and enabling the port, usbd allocates an endpoint (by issuing a control request to the kernel). This becomes the control endpoint for the device and represents the entire device from now on.

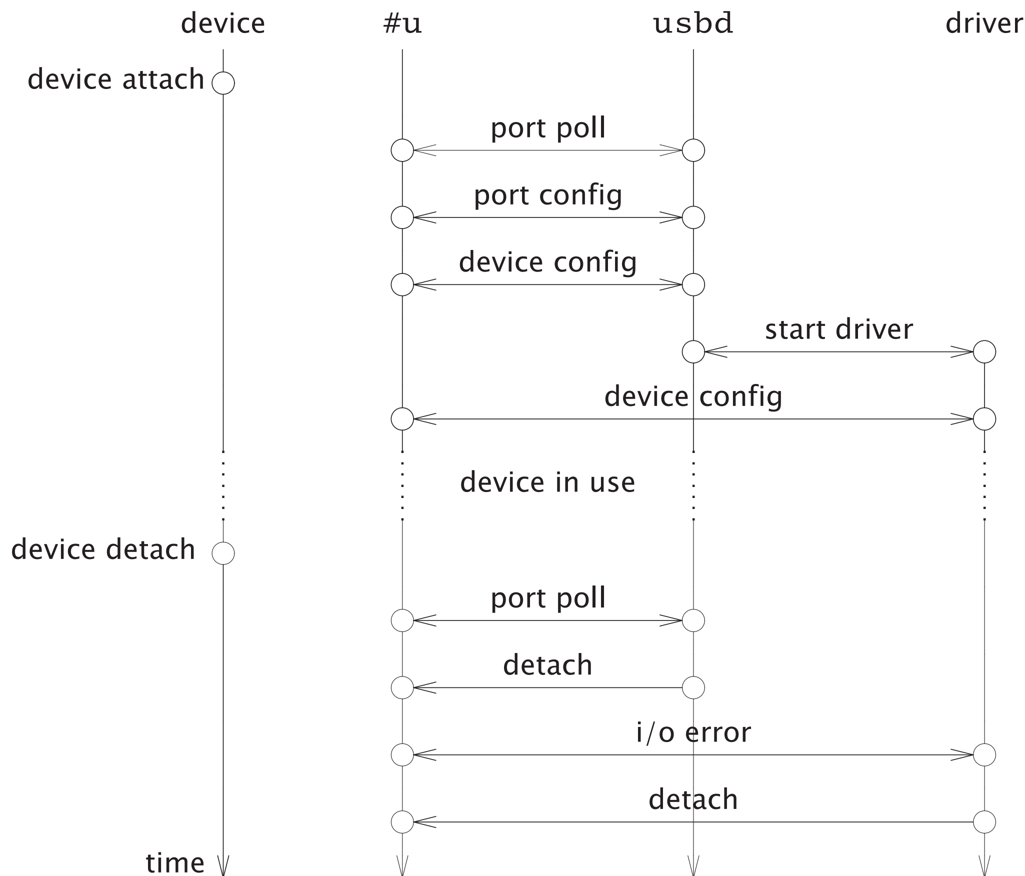


Figure 4. Device attachment and detachment. Time flows down.

Then, usbd configures the device (assigns an address, reads device descriptors reporting the type of device, and activates a configuration for the device). As part of the configuration usbd supplies to the kernel some information about the device according to its descriptors. Most of this information is not used by kernel. It is intended to provide enough information for drivers and users to locate devices, and to learn the relevant data about an endpoint of interest. CSPs, vendor, and product identifiers, and device strings are handled this way. As an example, this is the information supplied for an endpoint:

```
; cat /dev/usb/ep3.0/ctl
enabled control rw speed full maxpkt 64 ival 0 samplesz 0 hz 0 hub 1 port 3 busy
storage csp 0x500608 vid 0x951 did 0x1613 Kingston 'DT 101 II'
;
```

Once the device is configured, `usbd` is able to start a new process to execute the device driver if instructed to do so. Either case, `usbd` will *not* touch the device from now on. The driver started is free to open as many endpoints as needed, and to create new endpoints for the device, should it require that, without interference from `usbd`.

At some point the physical device may be detached. Now one of two things will happen. Either `usbd` notices that first, as the result of polling the port used by the device; or the device driver learns it first because of I/O errors while using the device. The process noticing the event first, whichever it is, issues a `detach` control request to the kernel to notify the detachment of the device.

Upon detachment, no further I/O is allowed and all requests (but for a few control requests) return a ''device is detached'' error diagnostic. Thus, if `usbd` notices first it causes the detachment of the device in the kernel. Any outstanding I/O request is canceled and the driver is notified that the device is detached. If the device driver notices first then it will detach the device and exit on its own (and at some point `usbd` will see the port as detached).

Another device may be plugged into the port before the next poll done by `usbd`. This can be detected because the port will not be enabled (although it will have a device connected). In this case we pretend that the port suffered two events: a detach followed by an attach. By doing it this way we can avoid the need for using interrupt endpoints provided by most hubs (real non-root hubs) to report status changes for ports. This saves the software to process such endpoints and also saves their software emulation in the case of root hubs.

Device drivers providing a file system interface are very important regarding hot-plugging of devices. These drivers must not simply exit when a device is detached. When the device driver is embedded into `usbd` it may be that no process is currently executing the driver's code, because the file system implementation is shared in this case. Thus, upon detachment the file interface is removed from the file tree and no further action is taken.

When the device driver runs as a separate process it should stay running but respond with I/O errors to any further I/O attempt. However, the file tree must be kept alive and responding to permit any client to continue working, if only to exit cleanly. Note that in other case, should the device be bound at `/dev`, the namespace would become broken whenever `/dev` is used. When the user notices that the device is no longer working the file system will be unmounted and at that point the driver exits.

5. USB device drivers

A USB device driver on Plan 9 is a user program responsible for configuring and operating a USB device. When `usbd` notices a new device on the bus it inspects its static configuration to see if a device driver must be started. Should that be the case, `usbd` creates a new process to execute the driver (either by calling `exec` for external drivers or by calling the driver's `init` function for those linked within `usbd`).

External or standalone drivers must take the burden of locating the devices to drive. By convention a driver will manage all devices known that are not managed by an already executing driver. All the information needed for this purpose (including if the device is in use by another driver) is available by reading the control files of existing control endpoints. That is, files named `/dev/usb/ep0.*/ctl`.

In practice, a device driver may be started to serve an interface (as part of a driver for the entire device). Thus device drivers must pay attention not only to the device CSP but also to CSPs for device interfaces. On Plan 9 all this information is retrieved by reading control files and most drivers may forget this detail.

Many things done by USB drivers are done by several (when not all) drivers. The corresponding code is kept in a USB library, which is also used by usbd. The main data structure in the library is Dev. It represents an endpoint provided by #u.

After a driver (or usbd) identifies a device of interest (i.e., an endpoint) it calls the library to create a Dev data structure for it, and spawns a new process using it as an argument, to handle the device. For drivers built into usbd it is usbd who cares of what has been said so far. For other drivers convenience routines are present in the USB library that can do the job.

The driver entry point is not main, but a function called init that receives a Dev data structure representing the device control endpoint. It is done this way to permit the same code to be used both for the embedded version of the driver and for a standalone version executed as an independent program.

The Dev structure refers to an endpoint directory at #u and includes file descriptors for the endpoint control and data files:

```
struct Dev
{
        Ref;
        char*     dir;                /* path for the endpoint dir */
        int       id;                 /* usb id for device or ep. number */
        int       dfd;                /* descriptor for the data file */
        int       cfd;                /* descriptor for the control file */
        int       maxpkt;             /* cached from usb description */
        Ref       nerrs;              /* number of errors in requests */
        Usbdev*   usb;                /* USB description */
        void*     aux;                /* for the device driver */
        void      (*free)(void*);     /* idem. to release aux */
};
```

Initially only the control file is open. Usbd keeps the data file (for the control endpoint) open while configuring the device but closes all descriptors before starting the device driver. The device driver keeps both the control and the data files open most of the time.

This data structure is reference counted to permit an endpoint to go only when no part of the driver is using it. This is important specially for drivers providing a file system interface, which might have outstanding requests while trying to shutdown the device.

All the relevant USB device descriptors must be read from the device to determine which interfaces are present and which endpoints should be used for I/O. This information is gathered by the USB library and placed into a Usbdev structure pointed to by Dev.usb. Therefore, few device drivers require reading descriptors themselves.

A driver must inspect the USB configuration information to locate interfaces with CSPs that correspond to the functionality provided, and also to locate the endpoints to be used for I/O. Despite help from the USB library all drivers must do this before allocating other endpoints on the device.

In some cases this is not enough. Some devices have descriptors that are specific for them, and are not parsed by the USB library. The library places such unpacked descriptors within the Usbdev structure, and the driver is responsible for parsing them if necessary. Chapter 9 of [1] is a good reference for common USB descriptors. Specification documents for particular device classes (or for particular devices) usually describe all device-specific descriptors necessary to drive the hardware.

In many cases drivers may remain unaware of most standard control requests, because the library has functions that do most of the work (configuring the device and unstalling endpoints upon errors). Device specific requests on the other hand must

always be issued by the driver, who knows when and how they must be done. The only thing the library can do is to provide a general purpose `usbcmd` utility function and some symbols to help building such requests.

5.1. Embedded drivers

For most drivers there is almost no difference between the embedded version and the standalone one. As said above, either `usbd` or a `main` function for the driver takes care of locating devices of interest and calling the driver's `init` function for each device found. A side effect of preparing a driver for embedding is that its `main` function may be borrowed almost entirely from another driver.

All other code is kept into a USB driver library that is linked to `usbd` (and also to the standalone version of the driver). This library contains drivers and is not to be confused with the USB library that is a convenience for writing drivers.

An important point for embedded drivers is that they must be careful not to rely on static storage and must be programmed to be reentrant. This is necessary because several instances of the driver may be created at different times to handle different devices, and they should rely just on the `Dev` structure for the control endpoint to operate the device (Of course further endpoints and other data structures can be created but none of them may be static storage).

Reference counting is also important for embedded drivers. A driver should go when the `Dev` reference supplied to its entry point goes down to zero. Initially, the driver's `init` function is given a `Dev` that counts one reference. When the reference goes away the endpoint is closed and the device released. If this happens due to an error, the driver issues a `detach` control request to the endpoint before releasing the reference, causing the endpoint to be collected when the last reference goes away. Drivers may install their own auxiliary structure and a free routine into `Dev`, as an aid to the implementation and also to support clean exits.

5.2. File system interface

The file interface provided by some USB drivers is important because it is the conventional interface for the corresponding devices on Plan 9. But it is also important because it is a key piece of the hot plugging mechanism.

If a device is removed and the file system interface for its driver simply responds with I/O errors to any further request, or aborts, `/dev` may become broken in the name space where this happens.

As devices come and go it is important to be able to see the file interface for devices adjust accordingly. For example, a directory named `/dev/sdU3.0`, corresponding to a disk for the device number 3, is expected to be there (albeit in a detached state) upon disk disconnection; as long as there is someone using the file interface. Only actual data transfers are reported as failed with I/O errors. Directories still work. When the last user of this file is gone, the directory silently disappears. Otherwise, inserting and removing a disk several times would lead to multiple (unused) directories and device numbers would become large numbers hard to remember.

The venerable 9P library is not used by USB software because of the requirements mentioned before. Instead, a small and specialized USB file system library is included in the standard USB library.

The interface for a file server is defined by the following data structure:

```
struct Usbfs
{
	char		name[Namesz];
	uvlong		qid;
	Dev*		dev;
	void*		aux;

	int		(*walk)(Usbfs *fs, Fid *f, char *name);
	void		(*clone)(Usbfs *fs, Fid *of, Fid *nf);
	void		(*clunk)(Usbfs *fs, Fid *f);
	int		(*open)(Usbfs *fs, Fid *f, int mode);
	long		(*read)(Usbfs *fs, Fid *f, void *data, long count, vlong offset);
	long		(*write)(Usbfs *fs, Fid*f, void *data, long count, vlong offset);
	int		(*stat)(Usbfs *fs, Qid q, Dir *d);
	void		(*end)(Usbfs *fs);
};
```

It can be seen how it includes both a `name` for the file tree and a `Dev` reference, pointing to the device behind the file system interface. Only operations named in `Usbfs` have to be implemented (some of them may left null if not of interest). They work as expected given their names.

Of these operations, `open`, `read`, and `write` are executed concurrently by the library using different processes. Thus an implementation may block if necessary without blocking the entire interface for the device (or worse, the entire set of drivers linked into `usbd`). Remaining operations must terminate promptly and are supported by a single, per client, process started by the library.

Despite concurrency there is no flush mechanism. Upon a flush the ongoing operation is left alone (only its result is ignored). This is not a problem in the case of USB drivers. Some flushes happen after device errors (e.g., because an application or a user interrupts a request). They find the device in a detached state as explained before. As a side effect, the detach operation interrupts any further I/O, making flush unnecessary in this case. Other flushes are legitimate interrupt requests issued by the user, and we simple let the outstanding operations complete and be ignored when they do.

The file system interface is designed to stack file systems. The library provides a file system implementation called `usbdir` that implements a single directory and accepts calls to `usbadd` and `usbdel` functions to plug other file systems into this directory. The `name` field in `Usbfs` is used as the name of the file tree when stacked into this directory, and the `qid` field is used to multiplex the Qid space among different file trees. This is the mechanism used to provide the same file interface, some times within an entire tree provided by `usbd`, some times as a solitary tree provided by the device.

6. Status and future work

The software described in this paper is operational. However, there are several issues that must be addressed. Input Isochronous streams for OHCI controllers are not yet implemented. As said before, bulk transfers should perhaps not be timed out by the kernel. The ethernet driver seems to have problems and must be fixed. Also, due to hardware unavailability, version 3.0 of the bus is not supported. In general, the source must be reviewed to undergo some cleaning.

In the future these issues will be addressed and more drivers will be written for other devices present on the bus.

There is plenty of room for performance tuning and optimization. The only optimization introduced so far is using embedded buffers for transfer descriptors. Data structures used to maintain controller specific transfer descriptors include a few bytes of

data storage. For transfers small enough (which are common) no data buffer is allocated and the embedded buffer is used instead.

7. Acknowledgments

The USB software described here is the result of a collective and secret effort. We are grateful to the anonymous authors of the previous USB software (Charles Forsyth and others). We are grateful to Plan 9 developers for the system and to 9fans for testing the software. Geoff Collyer proved to be invaluable to make bugs show up, so they could be debugged. Gorka Guardiola implemented the USB serial device driver and support for USB KNX devices. Cinap Lenrek implemented the old ethernet driver and fixed the new one to make it work. Russ Cox suggested the mechanism to provide named files and avoid keeping a driver running on simple cases. Enrique Soriano added support for a USB toy, a demon with wings that I expect to use as a mail biff. Erik Quanstrom hunted a bug in the USB disk. Thank you all.

References

1. Compaq, Hewlett–Packard, Intel, Lucent, Microsoft, NEC and Philips, USB 2.0 Specification, 2000.
2. Compaq, Microsoft and N. Semiconductor, OpenHCI – Open Host Controller Interface Specification for USB, 1995.
3. Intel, Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0, 2002.
4. Intel, Universal Host Controller Interface Design Guide, Revision 1.1, 1996.

Using Currying and process-private system calls to break the one-microsecond system call barrier

Ronald G. Minnich[*][†] and John Floren and Jim Mckie [‡]

January, 2009

Abstract

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. In contrast to 20 years of tradition in High Performance Computing (HPC), we require that programs access network interfaces via the kernel, rather than the more traditional (for HPC) OS bypass.

In this paper we discuss our research in modifying Plan 9 to support sub-microsecond "bits to the wire" (BTW) performance. Rather than taking the traditional approach of radical optimization of the operating system at every level, we apply a mathematical technique known as Currying, or pre-evaluation of functions with constant parameters; and add a new capability to Plan 9, namely, process-private system calls. Currying provides a technique for creating new functions in the kernel; process-private system calls allow us to link those new functions to individual processes.

1 Introduction

We have ported the Plan 9 research operating system to the IBM Blue Gene/L and /P series machines. Our research goals in this work are aimed at rethinking how HPC systems software is structured. One of our goals is to re-examine and, if possible, remove the use of OS bypass in HPC systems.

OS bypass is a software technique in which the application, not the operating sytem kernel, controls the network interface. The kernel driver is disabled, or, in some cases, removed; the functions of the driver are replaced by an application or library. All HPC systems in the "Top 50", and in fact most HPC systems in the Top 500, use OS bypass. As the name implies, the OS is completely

---

[*]

[†]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration under contract DEAC0494AL85000. SAND- 2009-5156C.

[‡]Bell Laboratories, Murray Hill, NJ, USA

bypassed; packets move only at the direction of the application. This mode of operation is a lot like the very earliest days of computers, where not a bit of I/O moved unless the application directly tickled a bit of hardware. It involves the application (or libraries) in the lowest possible level of hardware manipulation, and even requires application libraries to replicate much of the operating systems capabilities in networking, but the gains are seen as worth the cost.

One of the questions we wish to answer: is OS bypass still needed, or might it be an anachronism driven by outdated ideas about the cost of using the kernel for I/O? The answer depends on measurement. There is not much doubt about the kernel's ability to move data at the maximum rate the network will support; most of the questions have concerned the amount of time it takes to get a message from the application to the network hardware. So-called short message performance is crucial to many applications.

HPC network software performance is frequently characterized in terms of "bits to the wire" (BTW) and "ping-pong latency". Bits to The Wire is a measure of how long it takes, from the time an application initiates network I/O, for the bits to appear on the physical wire. Ping-pong latency is time it take a program to send a very small packet (ideally, one bit) from one node to another, and get a response (usually also a bit). These numbers are important as they greatly impact the performance of collectives (such as a global sum), and collectives in turn can dominate application performance [2] [4] In an ideal world, ping-pong latency is four times the "bits to the wire" number. Some vendors claim to have hit the magical 1 microsecond ping-pong number, but a more typical number is 2-3 microseconds, with a measured BTW number of 700 nanoseconds. However, these numbers always require dedicated hosts, devices controlled by the application directly, no other network activity, and very tight polling loops. The HPC systems are turned into dedicated network benchmark devices.

A problem with OS bypass is that the HPC network becomes a single-user device. Because one application owns the network, that network becomes unusable to any other program. This exclusivity requires, in turn, that all HPC systems be provisioned with several networks, increasing cost and decreasing reliability. While the reduction in reliability it not obvious, one must consider that the two networks are not redundant; they are both needed for the application to run. A failure in either network aborts the application.

By providing the network to programs as a kernel device, rather than a set of raw registers, we are making HPC usable to more than just specialized programs. For instance, the global barrier on the Blue Gene systems is normally only available to programs that link in the (huge) Deep Computing Messaging Facility (DCMF) library or the MPI libraries[1], which in turn link in the DCMF. Any program which wishes to use the HPC network must be written as an MPI application. This requirement leads to some real problems: what if we want the shell to use the HPC network? Shells are not MPI applications; it makes

[1]MPI libraries are typically much larger than the Plan 9 kernel; indeed, the configure script for OpenMPI is larger than the Plan 9 kernel

no sense whatsoever to turn the shell into an MPI application, as it has uses outside of MPI, such as starting MPI applications!

On Plan 9 we make the global barrier available as a kernel device, with a simple read/write interface, so it is even accessible to shell scripts. For example, to synchronize all our boot-time scripts, we can simply put `echo 1 > /dev/gib0barrier` in the script. The network hardware becomes accessible to any program that can open a file, not just specialized HPC programs.

Making network resources available as kernel-based files makes them more accessible to all programs. Seperating the implementation from the usage reduces the chance that simple application bugs will lock up the network. Interrupts, errors, resources conflicts, and sharing can be managed by the kernel. That is why it is there in the first place. The only reason to use OS bypass is the presumed cost of asking the kernel to perform network I/O.

One might think that the Plan 9 drivers, in order to equal the performance of OS bypass, need to impose a very low overhead – in fact, no overhead at all: how can a code path that goes through the kernel possibly equal an inlined write to a register? The problem with this thinking, we have come to realize, is the fact that complexity is conserved. It is true that the OS has been removed. But the need for thread safety and safe access to shared resources can not be removed: the support has to go *somewhere*. That somewhere is the runtime library, in user mode.

Hence, while it is true that OS bypass has zero overhead in theory, it can have very high overhead in fact. Programs that use OS bypass always use a library; the library is usually threaded, with a full complement of locks (and lockiing bugs and race conditions); OS functions are now in a library. In the end, we have merely to offer lower overhead than the library.

There are security problems with OS bypass as well. To make OS bypass work, the kernel must provide interfaces that to some extent break the security model. On Blue Gene/P, for example, DMA engines are made available to programs that allow them to overwrite arbitrary parts of memory. On Linux HPC clusters, Infiniband and other I/O devices are mapped in with mmap, and users can activate DMAs that can overwrite parts of kernel memory. Indeed, in spite of the IOMMUs which are supposed to protect memory from badly behaved user programs, there have been recent BIOS bugs that allowed users of virtual network interfaces to roam freely over memory above the 4 gigabyte boundary. Mmap and direct network access are really a means to an end; the end is low latency bits to the wire, not direct user access. It is so long since the community has addressed the real issue that means have become confused with ends.

2 Related work

The most common way to provide low latency device I/O to programs is to let the programs take over the device. This technique is most commonly used on graphics devices. Graphics devices are inherently single-user devices, with

multiplexing provided by programs such as the X server. Network interfaces, by contrast, are usually designed with multiple users in mind. Direct access requires that the network be dedicated to one program. Multi-program access is simply impossible with standard networks.

Trying to achieve high performance while preserving multiuser access to a device has been achieved in only a few ways. In the HPC world, the most common is to virtualize the network device, such that a single network device appears to be 16 or 32 or more network devices. The device requires either a complex hardware design or a microprocessor running a real-time operating system, as in Infiniband interfaces: thus, the complex, microprocessor-based interfaces do bypass the main OS, but don't bypass the on-card OS. These devices are usually used in the context of virtual machines. Device virtualization requires hardware changes at every level of the system, including the addition of a so-called iommu [1].

An older idea is to dynamically generate code as it is needed. For example, the code to read a certain file can be generated on the fly, bypassing the layers of software stack. The most known implementaiton of this idea is found in Synthesis [3]. While the approach is intriguing, it has not proven to be practical, and the system itself was not widely used.

The remaining way to achieve higher performance is by rigorous optimization of the kernel. Programmers create hints to the compiler, in every source file, about the expected behaviour of a branch; locks are removed; the compiler flags are endlessly tweaked. In the end, this work results in slightly higher throughput, but the latency – "bits to the wire" – time changes little if at all. It is still too slow. Recent experiences shows that very high levels of optimization can introduce security holes, as was seen when a version of GCC optimized out all pointer comparisons to NULL.

Surprisingly, there appears to have been little other work in the area. The mainline users of operating systems do not care; they consider 1 millisecond BTW to be fine. Those who do care use OS bypass. Hence the current lack of innovation in the field: the problems are considered to be solved.

The status quo is unacceptable for a number of reasons. Virtualized device hardware increases costs at every level in the I/O path. Device virtualization adds a great deal of complexity, which results in bugs and security holes that are not easily found. The libraries which use these devices have taken on many of the attributes of an operating system, with threading, cache- and page-aligned resource allocation, and failure and interrupt management. Multiple applications using multiple virtual network interfaces end up doing the same work, with the same libraries, resulting in increased memory cost, higher power consumption, and a general waste of resources all around. In the end, the applications can not do as good a job as the kernel, as they are not running in priveleged mode. Applications and libraries do not have access to virtual to physical page mappings, for example, and as a result they can not optimize memory layout as the kernel code.

3 Our Approach

Our approach is a modification of the Synthesis approach. We do create curried functions with optimized I/O paths, but we do not generate code on the fly; curried functions are written ahead of time and compiled with the kernel, and only for some drivers, not all. The decision on whether to provide curried functions is determined by the driver writer.

At run time, if access to the curried function is requested by a program, the kernel pre-evaluates and pre-validates arguments and sets up the parameters for the driver-provided curried function. The curried function is made available to the user program as a private system call, i.e. the process structure for that one program is extended to hold the new system call number and parameters for the system call. Thus, instead of actually synthesizing code at runtime, we augment the process structure so as to connect individual user processes to curried functions which are already written.

We have achieved sub-microsecond system call performance with these two changes. The impact of the changes on the kernel code is quite minor.

We will first digress into the nature of Curry functions, describe our changes to the kernel and, finally discuss the performance improvements we have seen.

3.1 Currying

The technique we are using is well known in mathematical circles, and is called currying. We will illustrate it by an example.

Given a function of two variables, $f(x, y) = y/x$, one may create a new function, $g(x)$, if y is known, such that $g(x) = f(x, y)$. For example, if y is known to be 2, the function g might be $g(x) = f(x, 2)$.

We are interested in applying this idea to two key system calls: read and write. Each takes a file descriptor, a pointer, a length, and an offset. In the case of the Plan 9 kernel, we had used a kernel trace device and observed the behavior of programs. Most programs:

- Used less than 32 distinct pages when passing data to system calls
- Opened a few files and used them for the life of the program
- Did very small I/O operations

We also learned that the bulk of the time for basic device I/O with very small write sizes – the type of operation common to collective operations – was taken up in two functions: the one that validated an open file descriptor, and the one that validated an I/O address.

The application of currying was obvious: given a program which is calling a kernel function read or write function: $f(fd, address, size)$, with the same file descriptor and same address, we ought to be able to make a new function: $g(size) = f(fd, address, size)$, or even $g() = f(fd, address, size)$.

Tracing indicated that we could greatly reduce the overhead. Even on an 800 Mhz. Power PC, we could potentially get to 700 nanoseconds. This compares

very favorably with the 125 ns it takes the hardware to actually perform the global barrier.

3.2 Connecting curry support to user processes

The integration of curried code into the kernel is a problem. Dynamic code generation looks more like a security hole than a solution.

Instead, we extended the kernel in a few key ways:

- extend the process structure to contain a private system call array, used for fastpath system calls
- extend the system call code to use the private system call array when it is passed an out-of-range system call number
- extend the driver to accept a fastpath command, with parameters, and to create the curried system call
- extend the driver to provide the curried function. The function takes no arguments, and uses pre-validated arguments from the private system call entry structure

4 Implementation of private system calls on Plan 9 BG/P

To test the potential speeds of using private system calls, a system was implemented to allow fast writes to the barrier network, specifically for global OR operations, which are provided through `/dev/gib0intr`. The barrier network is particularly attractive due to its extreme simplicity: the write for a global OR requires that we write to a Device Control Register, a single instruction, which in turn controls a wire connected to the CPU. Thus, it was easy to implement an optimized path to the write on a per-process basis.

The modifications described here were made to a branch of the Plan 9 BG/P kernel. This kernel differed from the one being used by other Plan 9 BG/P developers only in that its portable `incref` and `decref` functions had been redefined to be architecture-specific, a simple change to allow faster performance through processor-specific customizations. In other words, we are comparing our curried function support to an already-optimized kernel.

First, the data structure for holding fast system call data was defined in the `/sys/src/9k/port/portdat.h` file (from this point on, kernel files will be assumed to reside under `/sys/src/9k/`, thus `port/portdat.h`).

In the same file, the `proc` struct was modified to include the following declarations:

```
/* Array of private system calls */
Fastcall *fc;
```

```
/* Our special fast system call struct */
struct Fastcall {
    /* The system call number */
    int scnum;
    /* A communications endpoint */
    Chan *c;
    /* The handler function */
    long (*fun)(Chan*, void*, long);
    void *buf;
    long n;
};
```

Figure 1: Fast system call struct

```
int cfd, gdf, scnum=256;
char area[1], cmd[256];
gfd = open("/dev/gib", ORDWR);
cfd = open("/dev/gib0ctl", OWRITE);
cmd = smprint("fastwrite %d %d 0x%p %d", scnum, fd, area, sizeof(area));
write(cfd, cmd, strlen(cmd));
close(cfd);
docall(scnum);
```

Figure 2: Sample code to set up a fastpath systemcall

```
/* # private system calls */
int fcount;
```

Programs are required to provide a system call number, a file descriptor, pointer, and length. It may seem odd that the program must provide a system call number. However, we did not see an obvious way to return the system call number if the system chose it. We also realized that it is more consistent with the rest of the system to have the client choose an identifier. That is how 9P works: clients choose the file identifier when a file is accessed. Note that, because the Plan 9 system call interface has only two functions which can do I/O, the Fastcall structure we defined above covers all possible I/O operations. The contrast with modern Unix systems is dramatic.

Next, we modified the Blue Gene barrier device, `bgp/devgib.c`, to accept `fastwrite` as a command when written to `/dev/gib0ctl`. When the command is written, the kernel allocates a new `Fastcall` in the `fc` array, using a user-provided system call number and a channel pointing to the barrier network, then sets `(*fun)` to point to the `gibfastwrite` function and finally increments `fcount`. The code to set up the fast path is shown in Figure 2.

Following the `write`, `scnum` contains a number for a private system call to write to the barrier network. From there, a simple assembly function (here

```
TEXT docall(SB), 1, $0
    SYSCALL
    RETURN
```

Figure 3: User-defined system call code for Power PC

called `docall`) may be used to perform the actual private system call. The code is shown in Figure 3.

When a system call interrupt is generated, the kernel typically checks if the system call number matches one of the standard calls; if there is a match, it calls the appropriate handler, otherwise it gives an error. However, the kernel now also checks the user process's `fc` array and calls the given `(*fun)` function call if a matching private call exists. In the case of the barrier device, it calls `gibfastwrite`, which writes '1' to the Device Control Register. The fastcall avoids several layers of generic code and argument checking, allowing for a far faster write.

5 Results

In order to test the private system call, we wrote a short C program to request a fast write for the barrier. It performs the fastpath setup as shown above. Then, it calls it calls the private system call. The private system call is executed many times and timed to find an average cost per call. As a baseline, the traditional write call was also tested using a similar procedure.

We achived our goal of sub-microsecond bits to the wire. With the traditional write path, it took approximately 3,000 cycles per write. Since the BG/P uses 850 MHz PowerPC processors, this means a normal write takes approximately 3.529 microseconds. However, when using the private system calls, it only takes around 620 cycles to do a write, or 0.729 microseconds. The overall speedup is 4.83. The result is a potential ping-pong performance of slightly under 3 microseconds, which is competitive wth the best OS bypass performance.

6 Conclusions and Future Work

Runtime systems for supercomputers have been stuck in a box for 20 years. The penalty for using the operating system was so high that programmers developed OS bypass software to get around the OS. The result was the creation of OS software above the operating system boundary. Operating systems have been recreated as user libraries. Frequently, the performance of OS bypass is cited without taking into account the high overhead of these user-level operating systems.

This paper shows an alternative to the false choice of slow operating systems paths or fast user-level operating systems paths. It is possible to use a general-

purpose operating system for I/O and still achieve high performance.

We have managed the write side of the fastcall path. What remains is to improve the read side. The read side may include an interrupt, which complicates the issue a bit. We are going to need to provide a similar reduction in overhead for interrupts.

We have started to look at curried pipes. Initial performance is not very good, because the overhead of the Plan 9 kernel queues is so high. It is probably time to re-examine the structure of that code in the kernel, and provide a faster path for short blocks of data.

Our goal, in the end, is to show that IPC from a program to a user level file server can be competitive with in-kernel file servers. Achieving this goal would help improve the performance of file servers on Plan 9.

7 Acknowledgements

This work is support by the DOE Office of Advanced Scientific Computing Research. IBM has provided support from 2006 to the present. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

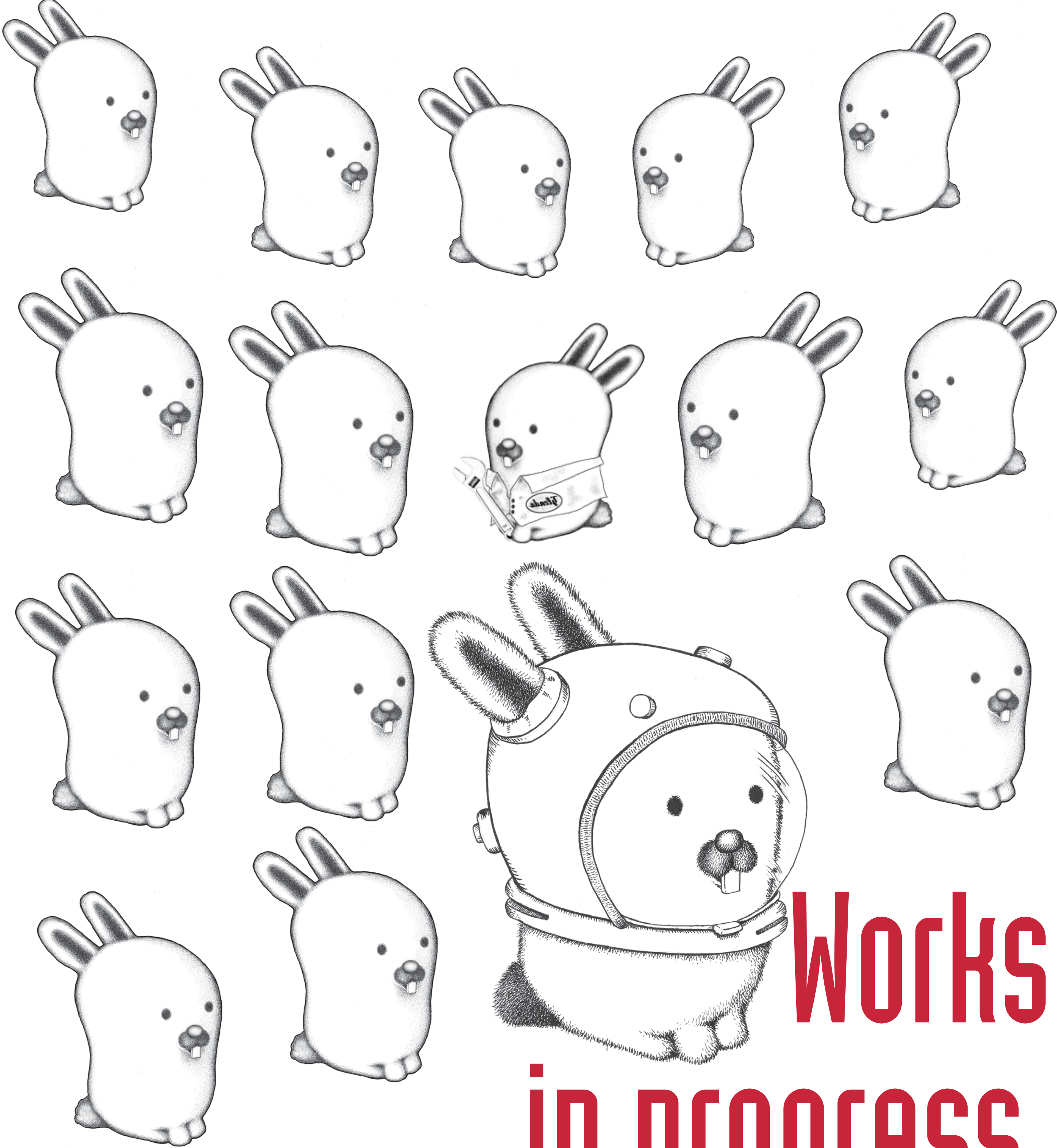
References

[1] Muli Ben-Yehuda, Jimi Xenidis, and Michal Ostrowski. Price of safety: Evaluating iommu performance. June 2007.

[2] Fabrizio Petrini, Darren J. Kerbyson, and Scott 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*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1:11–32, 1988.

[4] Y. Tanaka, K. Kubota, M. Sato, and S. Sekiguchi. A comparison of data-parallel collective communication performance and its application. *High Performance Computing and Grid in Asia Pacific Region, International Conference on*, 0:137, 1997.



Works
in progress.

iwp9.org

Btfs - a BitTorrent client

Mathieu Lonjaret
lejatorn@gmail.com

ABSTRACT

Btfs is an attempt at an implementation of the *BitTorrent* protocol. The primary goal is to design and program a BitTorrent client for *Plan 9 from Bell Labs* (referred to as *Plan 9* in the following) providing the basic functionalities found in most BitTorrent clients. The general design and user interface are elaborated with the Plan 9 file server model in mind.

1. Motivation

To start, there is no usable implementation of a BitTorrent client available on Plan 9. As it is a pretty popular protocol nowadays, it seems like a good idea to provide that possibility for a (hopefully) growing userbase. The distribution of the Plan 9 system could also benefit from it: there already are a few sites out of Bell Labs - already mirroring the Plan 9 iso - which could altogether be seeders for a torrent of the iso, hence relieving some of the load on the Bell Labs server.

Additionally, the BitTorrent protocol is both pretty simple and interesting as far as network protocols go, hence getting a lot of interest from various projects. Its distributed nature, the tracker component set aside, could lead to interesting developments on the Plan 9 platform. For example, the idea of a kind of coupling between BitTorrent and some venti servers to distribute the blocks was suggested.

2. Choices

2.1. Port or native

A port of the mainline BitTorrent client would probably be possible since Python has already been ported. However, there is no apparent obstacle to a native implementation following the Plan 9 file server approach. Appart from the obvious satisfaction of having a native program which integrates well with Plan 9, it will also give the opportunity for an acme program as an additional user interface later (akin to acme Mail). Finally, designing this project in its entirety is an educational opportunity to find out how well Plan 9 specific concepts like the file server approach and the csp model would fit in for a peer to peer program.

2.2. Language

As most of the Plan 9 code is written in C, this is the obvious choice. It allows for a better integration with the rest of the system and makes it easier if one wants to reuse pieces of code from other programs. However *limbo* fans should note there is another ongoing effort for a BitTorrent client (on *inferno*) being written in limbo.

2.3. HTTP queries

A BitTorrent tracker listens to http queries, thus one could write and add to the client the code needed to send such queries, or one can rely on already existing components such as hget or webfs. The latter was chosen, because there is no need for a rewrite when webfs is perfectly suited for this task, and again because it fits better with the file server way. However, queries to trackers can contain NULs, and webfs

---

See mjl’s contrib: http://plan9.bell-labs.com/sources/contrib/mjl/

currently does not allow that, so it needs to be slightly modified.

2.4. Threads

A BitTorrent client, like any peer to peer program, has to interact with a lot of other peers, and it is composed of several independant tasks, therefore it makes sense to distribute these tasks amongst several threads or procs. As threads are inherently safe to avoid race conditions, and since procs did not seem necessary, only threads are used for now.

The scheduling in use is what seemed to be the simplest: for each peer one thread is created and it gets all the pieces it can from this peer. The threads are synchronised with an *Alt* structure and each of them relinquishes control with *send(2)* at points in the execution where it would supposedly wait for network packets from the peer to come. An *Ioproc(2)* is used to avoid the situation where all the threads are blocked because one of them is waiting on a dial. This design is of course subject to changes depending on how well it will perform.

3. Implementation

The BitTorrent protocol makes use of a file (usually with .torrent extension) of metadata to describe the so called *torrent*. Btfs creates a Torrent struct in memory to represent this torrent and its metadata. Similarly, a Peer struct is used to hold the properties of each peer which btfs is communicating with.
The program is divided into four main parts as of now: the file server operations, parsing of the torrent file, operations on the Torrent structure, and communications with the tracker(s) and other peers.

The file server provides a synthetic tree created with *alloctree(2)* (mounted by default on /n/btfs) on which the user can act upon. The basic structure of the tree is envisionned like the following:

```
/
|-ctl
|-torrents
        |-c3cb0dfd1d2839861ea79367145b202db7c09c52
                |-tracker
                        |-announce
                |-pieces
                ...
        |-9a5a04f1ddff16de8f7bca8714e945083c4c53c7
        ...
```

where torrents are identified with their infohash (like c3cb0dfd1d2839861ea79367145b202db7c09c52). One can imagine a lot of others files which, when read, would return some useful information. Writing commands to the *ctl* file will be the basic way to control btfs, ie to add new torrents, stop some active torrents, throttle the upload/dowload rate, etc...
At the moment, only adding a torrent is supported, with the 'add /path/to/file.torrent' command.

The torrent file metadata are organized as a bencoded structure - a simple encoding which consists of a dictionary (associative array) of key/value pairs, the values themselves being of one of the following types: byte string, integer, list, dictionary. For now, these data are just read sequentially and analysed according to a documented set of dictionary keys. The corresponding values are stored in the Torrent structure as the decoding goes. This part should and will most certainly be rewritten as a more generic parser.

Most of the operations on the Torrent structure relate to the torrent pieces. Those are represented with linked lists: a global one, and one for each peer. The elements of the global one correspond to the whole torrent's pieces, and hold information such as the position of the piece in the torrent. The elements of a peer's list represent the pieces which are yet to be downloaded from this peer.

The communications start with a call to the tracker, after the right request have been forged, using the information from the torrent file. The tracker reply is parsed in the same fashion the torrent file was. As said before, a thread is created for each peer reported by the tracker, with the following limit: the maximum number of peers is arbitrarily fixed (to 20) for now - some specifications on the protocol state that for best

---

See patched /sys/src/cmd/webfs/url.c: http://plan9.bell-labs.com/sources/contrib/lejatorn/url.c

efficiency this number should not be too high (< 55) anyway.
Each thread requests from its dedicated peer all the pieces, in random order, that the peer announced it had, until all of them have been acquired.

4. Current state

A lot more needs to be done before btfs displays most of the expected functionalities from a BitTorrent client. In particular, the current main issues are: absolutely no support for seeding, no reply from some specific trackers, and dealing with the concurrency for communicating with several peers at the same time.

No seeding not only means that it is currently impossible to distribute a resource, it also means that the download rates will be impaired since most clients choke peers which do not seed in return.
The trackers issue will be investigated soon, it probably comes down to the tracker expecting an optional parameter in the query, like the peer key or the tracker id.
As mentionned before, the general problem of multi peers communications without the threads blocking each other should have been solved by the use of ioprocs. However, the current behavior is not what was expected (all of the connections to the peers get closed except for the first one). This is the most immediate concern and is what needs to be fixed first and foremost, therefore any help on the matter would be greatly appreciated. Especially since once this is solved, btfs is expected to be usable to download in quite a few cases.

5. Todo

On a longer time scale, the following items are planned:

- General improvements of the code: better error handling, fix some corner cases, hardcoded values, and ugly algorithms.
- Better communications: requery the tracker regularly to get fresh info, reconnect to some peers, keep a pool of threads ready. Use the other trackers in the tracker list.
- Enhance the file server: more "features" through the ctl file, and more files to read for info.
- Add rarest pieces first, and end game algorithms.
- Add dht support.
- An acme program as an additional user interface.

6. Check it out

Btfs can be downloaded from http://plan9.bell-labs.com/sources/contrib/lejatorn/; the directory usually contains the latest updates while the tarball should be a bit more stable. One will of course need a valid torrent file to try btfs out. Either get it from any of the numerous torrent hosting sites or create it yourself, with the mainline torrent creator *btmakemetafile* for example.
A quick tutorial:

```
webfs
btfs [-d datadir] [-m mountpoint] [-v]
echo 'add /path/to/the/torrent/file.torrent' > /path/to/the/mountpoint/ctl
```

Datadir is the directory where the downloaded files will be written (defaults to the user's home), mountpoint is where the btfs tree will be mounted (defaults to /n/btfs), and -v is for some debug verbosity (very verbose, therefore much slower).

See http://wiki.theory.org/BitTorrentSpecification#Tracker_Response

The btfs binary is not copied out of its source directory, so one will have to copy it where suitable, or bind it, or invoke it with its path.

Dynamic resource configuration and control for an autonomous robotic vehicle

Abhishek Kulkarni, Bryce Himebaugh and Steven D Johnson
School of Informatics and Computer Science,
Indiana University, Bloomington, 47401, USA
{adkulkar,bhimebau,sjohnson}@cs.indiana.edu

ABSTRACT

This Work-In-Progress report describes an application of the 9P distributed protocol to configure and control resources on an autonomous robotic vehicle, ERTS. The vehicle was designed and developed by the participants of a graduate level course on Embedded and Real-Time systems at *Indiana University, Bloomington*. The goal of the ERTS project is twofold – to teach students about embedded system development through the interaction with the robotic vehicle and to act as a prototyping platform for researchers seeking to meet experimental objectives in areas such as computer vision, artificial intelligence, situated cognition and learning and others.

SyncFS is a synchronous, double-buffered RAM-based virtual file system that defers writes and stats to a simulated "clock edge", thus governing the asynchronous sensor and actuator components around a central common clock. The SyncFS component model allows dynamic configuration of sensor and actuator components and remote resource access for these components when running in impoverished computing domains. This model allows rapid prototyping of components on a laptop and "importing" of the autonomous cart resources on the laptop during field tests. The file servers are implemented on Linux using the 9P library implementation, npfs, and use the in-kernel 9P client, v9fs, to mount them.

We are working on a native Windows userspace file system driver to support platform heterogeneity. We plan to build on prior work on organizing sensor networks and abstracting real-time embedded systems through a file system interface and seek to extend this to integrate with higher-level navigation systems. Some of the points under active consideration include modifying 9P protocol to support aggregate communication in a sensor network, using 9P over various embedded-network protocols, and exploring alternative programming models for synchronous/reactive (SR) systems.

1. Introduction

ERTS (which stands for Embedded and Real-Time Systems) is a computer-controlled golf cart developed for and by the participants of the introductory course on *Embedded and Real-time Systems*. It was developed to demonstrate autonomous real-world navigation and serve as a research platform to students and experimenters at Indiana University. This mission imposes a need for a flexible, modular and composable architecture for rapid prototyping and faster integration of software and hardware components in the embedded system. ERTS is a reactive distributable embedded system, as the components (sensors, actuators, embedded controllers) can be placed apart and communicate with each other through a channel interface. These components maintain a persistent interaction with the environment, reacting to the inputs from the environment and responding by sending outputs to it. Using a synchronous model coupled with appropriate file system abstractions, we have developed a distibuted embedded system runtime framework for ERTS.

At the heart of the ERTS software architecture is a synchronous commit file system, SyncFS, explained in section 2. Section 3. describes the modular component framework which encapsulates SyncFS to support distributed embedded devices in the system. Finally, we discuss the work in progress pertaining to the runtime framework of the ERTS vehicle.

2. SyncFS

SyncFS is a single-writer, multiple-readers file system with synchronous, system-wide commits. The synchronizing element is a simulated clock edge modeled on the functionality of

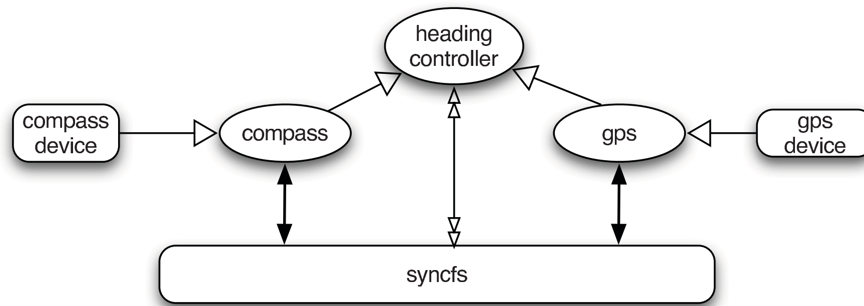


Figure 1: SyncFS component environment

globally clocked D flip-flop. In synchronous digital hardware, signals change only at clock edges. The results of these changes, in absence of propogation delay, are conceptually instantaneous. This makes synchronous systems easier to design and reason about. Several issues related to concurrency, paralellism and fault tolerance are simplified in these models. Automotive embedded systems, on the other hand, are inherently asynchronous (and so is the world around us!). SyncFS provides a synchronizing element to the component layer above it.

SyncFS supports a synchronous design model through a file-system interface. Part of this support is provided through modification to standard file-I/O (FIO) handlers. Equally important is a collection of coding conventions, governing use of files acting as communication channels. We prefer a light-weight imposition of coding conventions, if only because the design model and methods are still evolving. In other words, SyncFS is more an example of methdological support, than a end in itself.

SyncFS implements synchronization by:

1. modifying file *write* calls to defer all actual write commits until a triggering event derived from the server's physical clock.
2. modifying file *stat* calls to block the caller until all write commits are completed.
3. requiring all components to *stat* a common *clock* file at the outset of each cycle.

Under the SyncFS regimen, there are no data races, provided that all component-tasks execute within a clock cycle.A file write may make successive changes to a file without effecting its visible state to the rest of the system. On the server's virtual clock tick, the writer's version is "latched" and committed so that it becomes visible to the readers. SyncFS update their state concurrently within a clock edge. Like other reactive systems, this causes reactions to compete with each other. New inputs arrive before the end of a reaction. For explicit synchronization and to provide a common global time reference, SyncFS updates a read-only *clock* file on each tick.

SyncFS builds on the embedded file system approach of Brown and Pisupati[3], whose work was based on the original concepts prevalent in the Plan 9 Operating System from Bell Labs. The file namespace hierarchy provides a shared, language-independent region through which the ERTS system components interact. Thus, any language with standard I/O—that is, any language—can interact with system components. In class projects, for example, we use the *Python* scripting language for initial prototpying, and can incrementally convert components to C, Java, or other targets.

3. Component Framework

The component framework is a implementation-independent, conceptual model describing the components and interaction between them. Components can either be real components accesing the device through a driver, or virtual components which interface with other real components in the component ecosystem.

Figure 1 shows real sensor components like compass, GPS and virtual components like the heading controller that control the dynamics of the cart. A physical resource (sensor or actuator) in the system is represented by a textual data file that shadows the data image of the device. The interaction with these components is translated into physical transactions with the sensors or actuators if it is a real component or inter-component transactions in case of a virtual component. Components internally consist of a reactive kernel and a data handling layer coupled with the data-in and data-out interfaces.

Each component waits for a simulated clock edge, reads from its input channels, executes a function f and writes to its output channels. A component exposes a file system hierarchy consisting of the *command* file, through which the component is controlled, a *status* file, through which the state of the component can be captured. Some complex compoeenents have other auxiliary files for specialized control. Since the components model a design function behavior of a state machine, they appear as cyclic reactive processes that read/write files at the beginning/end of every cycle, performing a certain function, f. Components have input and output ports connected to other components in the environment. These ports go through SyncFS which acts as a mediator to ensure synchronous access behavior on these channels.

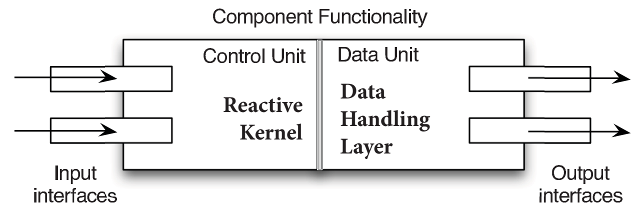


Figure 2: Component architecture

All components are peers that may access each others' resources as exposed by the SyncFS file server. In contrast to Plan 9, each component is expected to explicitly add itself to the global SyncFS namespace to interact with other components. A *terminal* component is the one that only accesses its own data.

3.1. Golf cart components

All file writes are globally synchronized when a ubiquitous system tick (a macrotick in Kopetz's terminology[2]) occurs. SyncFS provides an implicit clock component to which new and reentering components can explicitly synchronize. We have demonstrated ERTS in scenarios involving autonomous vehicle navigation. Several components for devices like compass, GPS, joystick and virtual components like bearing control, steering amplifier, configuration process and others interact to achieve autonomous navigation. The file format used by us to exchange structured data between these components is JavaScript Object Notation (JSON) for it is widely used, relatively light-weight, simple and readable.

4. Work in Progress

4.1. Windows file system support

ERTS uses the *v9fs* modules shipped with the Linux kernel since 2.6.14 to mount the SyncFS file system. Linux has primarily been the prototyping and development platform used until now. One of the primary missions of the ERTS robotic vehicles is to serve as a experimental platform for research in areas ranging from robotic vision, human-robot interaction to safety-critical systems. As we engage the cart in more collaborative research at Indiana University, we feel the need to take into consideration heterogeneous development environments and platforms. Oftentimes, the most preferred platform of development for researchers achieving experimental objectives in allied fields is Microsoft©Windows. This can be partly attributed to the lack of availability of specialized tools and software on other platforms.

This compelled us to add native support for 9P file system in Windows. We chose the approach of writing a userspace file system driver over a Windows Installable File System (IFS) driver to save us from the effort of writing a driver in the Windows kernel, and save time to focus on other important aspects of the project.

This driver is partially implemented and we hope to have full support in a few weeks. This would enable us to write components around SyncFS in Windows, and enable interaction of these components with our existing computer vision algorithms for horizon detection, visual tracking and object recognition, collaboratively developed under the hood of ERTS vision project at Indiana University.

4.2. Component Framework in Inferno

We are experimenting with implementing the components in hosted Inferno to provide us with the desired platform heterogeneity. We intend to rewrite some of the components in Limbo and use some of the abstractions provided by the language, in the form of typed channels and CSP paradigm to implement a synchronous, time-triggered embedded programming framework.

5. Future Directions

In our effort to incorporate a homogeneous resource access interface in the underlying file system, we have encountered several potential directions to explore in the ERTS project.

5.1. Programming Models for Synchronous Design Methodology

A file namespace hierarchy provides a homogeneous representation of heteregeneous resources, but it provides no explicit means of governing process synchronization and scheduling. Real-time embedded systems, safety- critical systems usually employ a synchronous design methodology through a higher level abstraction of the underlying architecture and system.

Time-triggered programming languages like ESTEREL, GIOTTO can aptly specify and model concurrent reactive systems. We intend to explore synchronous programming models and illustrate its use in real-time embedded systems using ERTS. In its current stage of implementation, we have an almost working functional interface to the SyncFS component model, written in the Scheme programming language.

5.2. Support for embedded-network protocols

As the DARPA Grand challenge demonstrated, more and more autonomous vehicles are now employing "drive-by-wire" and "steer-by-wire" technologies to off-load critical navigation functionality to a group of networked computers. We would like to push the frontier of SyncFS beyond ERTS's existing support for Ethernet/Linux network. Embedded-network protocols lay emphasis on higher data rates, low power consumption, time and event-triggered behavior. The delays due to sequential component dependencies can be reduced by implementing 9P over these protocols. Automotive systems have been using specialized bus-based protocols like CAN-bus and other fieldbus protocols for over a decade.

Realtime Ethernet protocols like Ethernet Powerline, EtherCAT provide synchronized, real-time network access over standard Ethernet. These are most likely our initial favorable targets to port 9P to minimize the inter-component communication delays.

5.3. Distributed Clock Synchronization

When used in a distributed context, there is a certain performance penalty incurred by SyncFS owing to its design. Performance and resiliency concerns thus make distributed clock synchronization desirable. This is an important goal in further development of SyncFS.

6. Conclusion

In a couple of years, commercial premium-class vehicles will contain over 1 GB of onboard system software. Autonomous vehicles are also geared with drive-by-wire capability which enforces the need for distributed, reusable component-based models for representing embedded systems. We believe the approach we have taken is quite viable for rapid prototyping in resource-rich environments. For our purposes, language independence and light-weight, composable tool chain are extremely important benefits.

7. References

1. Steven D Johnson, Bryce Himebaugh and Scott A. Dial. Homogeneous resource configuration and access for an autonomous robotic vehicle. *S*AE Int. J. Commer. Veh, **1**(1): 534–543, October 2008.

2. Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* 1997, Kluwer, Boston

3. Bhanu N. Pisupati & Geoffrey Brown. File system interfaces for embedded software development. *2005 IEEE Intl. Conf. on Computer Design* (ICCD 2005), October 2005, 232–235.

Two Enhancements for Native Inferno

Brian L. Stuart
University of Memphis
blstuart@bellsouth.net

ABSTRACT

Here I report on recent work done in the process of moving native Inferno to a tablet PC. One part of the work was the porting of Francisco J. Ballesteros's new Plan9 USB support to Inferno. The other was a VGA driver that operates with the 640x480x4 mode common to all VGA controllers.

Introduction

Recently, I worked on getting native Inferno to run on a tablet PC on loan from the vendor. This ruggedized unit had a camera/2D barcode reader, a touch panel, Ethernet, and 802.11. What it did not have was PS/2 keyboard or mouse interfaces. Unable to get the legacy support working and knowing that USB support would probably be needed anyway, I ported to Inferno the new Plan9 USB support from Franciso J. Ballesteros[1]. After getting it running in a text-only mode, it was natural to expect some form of graphics, particularly when the touchpad was being used. After a little searching, it became clear that it was possible to write a driver that handled the VGA features that are common to all controllers with only a moderate amount of effort. Here are the results of those efforts.

USB

There are four main components of the Plan9 USB support: the [uoe]hci drivers, a USB support library, the usbd daemon, and the drivers for individual USB devices.

USB HCI Drivers

Only minimal changes were required to port the files usbohci.c, usbuhci.c, and usbehci.c to Inferno. One change was in the calls to *kproc*(10). Inferno's version takes one more argument than Plan9's. The EHCI driver uses clink as a structure member name. However clink is a macro in Inferno. The OHCI driver required a small change to the initialization sequence on the system where OHCI testing was done. Without this change, if no device was plugged into some port when the system came up, it would never detect one inserted after the system was up. The biggest change to these drivers was moving some of the *ilock*(10) and *iunlock*(10) calls around a bit. In particular, *wakeup*(10) could not be called while an ilock was held. Otherwise, there was a good chance the system would panic when an attempt was made to acquire an already-held ilock.

USB Support Library

The most obvious change in porting the USB support library is the translation of it from C to Limbo. One major change in functionality was introduced, however. In particular the USB library provides support for serving a small file tree separate from the files provided by the HCI drivers. This functionality was dropped, with the plan to implement it later if needed. Otherwise, the library was a pretty straightforward merging of the C source files, translating them into one Limbo file. The resulting library is /dis/lib/usb/usb.dis.

USB Daemon

As with the library, the daemon port consisted of merging and translating the C files for usbd into Limbo. However, the loading of the device database and the starting of child driver processes was modified. Specifically, the existing Inferno device database format and calling sequence for child drivers was retained. As a consequence of the removal of file server support in the USB support library, the file /dev/usbdctl was not implemented.

USB Device Drivers

The original Inferno USB daemon took advantage of the nature of Inferno modules to implement the drivers for individual devices attached to USB ports. All such drivers have a common initialization entry point. When a newly inserted device is recognized, the relevant driver module is loaded and a new process is spawned on that driver's initialization function. To date, three such drivers have been implemented.

Keyboard Driver

This driver borrowed elements from several sources including the original Inferno USB keyboard driver, the Inferno native PS/2 keyboard driver, and the new Plan9 USB keyboard driver. It handles a basic ASCII keyboard with repeat. The implementation includes the function keys, the arrow keys, and the home, end, page up, and page down keys.

Mouse Driver

The mouse driver takes USB mouse messages and translates them into the form necessary to write them into /dev/pointer. To support this, devpointer.c was modified to allow more than one process to have /dev/pointer open at a time.

Mass Storage Driver

Much of the original Inferno mass storage driver was kept in this implementation. The driver has been tested on a USB memory stick and on a USB-connected CD-ROM drive.

Baseline VGA Driver

The world of PC video controllers is notorious for its complex and often undocumented register interfaces. Identifying which controller is present, then applying the right magic formula of settings to get it into a particular mode is nightmarish. However, practically all VGA controllers implement the same interface as the original IBM VGA controller. That controller's two most useful modes were a 320x200x8 mode and a 640x480x4 mode. Although woefully inadequate for use on a desk top or most laptops, they are not entirely unreasonable for use in a touchpanel-based application, where the minimum size of objects is limited by the size of a person's finger. Furthermore, the 640x480 mode covers 63% of the width and 80% of the height of a typical 1024x600 netbook screen. Therefore, although no substitute for full VGA support, there is enough utility in implementing a common denominator VGA driver to invest the effort.

The driver vgabase.c supports these two video modes on all VGA controllers it's been tried on, including that of qemu. It provides the enable, drawinit, and flush functions of the VGAdev structure. It also provides the enable, load, and move functions of the VGAcur structure, implementing a software cursor. (The original VGA controller did not have a hardware graphics cursor.)

The driver itself handles the appropriate register initialization when the drawinit command is given to /dev/vgactl. Prior to that the resolution and depth were written into the VGAscr structure, and they are referenced to determine which of the two supported modes is required. Getting the exact register setting is a little tricky because the original IBM documentation was vague on a few points. However, there are a number of resources out there that help fill in the details, most notably the svgalib implementation. In the case of 640x480x4, the default colormap doesn't work very well, so the driver overrides it with a fixed colormap. It is possible to assign some colors along with shades of gray that are mostly acceptable with existing Inferno applications, a straight 16-level grayscale colormap is often preferred. Furthermore, the colormap can be overridden by an application program.

Implementing the flush operation is an exercise in deciphering the memory layout details for the frame buffer. In the 320x200x8 mode, things are much as one would expect. A block of the address space is assigned to the frame buffer and each pixel is stored in the frame buffer as a one-byte index into the colormap. The 640x480x4 mode is another matter entirely. In this mode there are four planes of memory each laid out as a 640x480x1 screen. Each pixel is made from one bit of each of these planes. Although a little messy, this by itself wouldn't be too bad. However, all four planes share the same address space, and other controller registers

determine how each byte of data written is applied to the four planes. The way it gets handled in vgabase.c is to split the screen image out into the four bit-planes. Then for each row, the controller is programmed to load each plane in sequence and the row is written to that plane.

As mentioned above, the graphics cursor is implemented in software. Each time the cursor position is set, the contents of the screen are read and the and/or operations for the cursor are applied before the data is written back. To keep it updated, there is a 50mS timer that checks to see if flush has overwritten any part of the cursor, and if so repaints it.

Future Work

The first avenue for further work is adding support for other types of USB devices. The Plan9 support includes support for Ethernet, audio, printer, and serial devices, none of which have been included in the Inferno support reported here. Also little testing has been done on the OHCI and UHCI drivers, so there's a good chance that some issues will arise.

For the VGA driver, the primary opportunity for improvement is performance. On slower machines, painting a screen in wm/man is noticeably slow. Whether more performance can be found is uncertain, but in some settings more is needed.

References

[1] Ballesteros, F J, "Plan 9's Universal Serial Bus," Proceeding of the 4th International Workshop on Plan9, 2009.

Ircfs and wm/irc

Mechiel Lukkien

mechiel@xs4all.nl

ABSTRACT

Irc, internet relay chat, is a popular chat protocol. *Ircfs*(4) is an irc client that maintains a connection to an irc server, and exports irc functionality through the styx/9p protocol. *Wm/irc* is a Tk program that provides a user interface to the file system interface. *Ircfs* and *wm/irc* have been used from early stages of development with only a few remaining features to be implemented and bugs to be fixed.

Introduction

Ircfs is typically run on a computer with a stable internet connection, with its files exported over styx. Another machine then mounts the file tree and accesses it using *wm/irc*. *Ircfs+wm/irc* was intended to replace the common *screen+irssi* set up. It has already done that for me and turned out to be an improvement: *Ircfs* is a more generic and reusable irc client, with no user interface logic in it. *Wm/irc* is a simple user interface program, not (very) specific to irc. Because the two programs are separate, they have been written and can be debugged and started independently. The user interface is run locally and thus always responsive.

Ircfs and *wm/irc* are both written in Limbo. More information including the code is available on the *ircfs* home page* and in the manual pages *ircfs*(4) and *wm–irc*(1).

This report continues with an overview of *ircfs*, followed by a description of *wm/irc*'s features and concludes with an explanation of design decisions and ideas for improvements.

Ircfs

Let's start with an example:

```
% mount {ircfs freenode} /mnt/irc
% cd /mnt/irc
% echo 'connect net!irc.freenode.net!6667 mjl0' >ctl
% ls -l
d-r-xr-xr-x M 2 ircfs ircfs 0 Sep 28 15:45 0
---w--w--w- M 2 ircfs ircfs 0 Sep 28 15:45 ctl
--r--r--r-- M 2 ircfs ircfs 0 Sep 28 15:45 event
--r--r--r-- M 2 ircfs ircfs 0 Sep 28 15:45 nick
--r--r--r-- M 2 ircfs ircfs 0 Sep 28 15:45 pong
--rw-rw-rw- M 2 ircfs ircfs 0 Sep 28 15:45 raw
% cat nick
mjl0
```

The `ctl` file accepts plain text commands ranging from connecting and disconnecting

* *Ircfs*, `http://www.ueber.net/code/r/ircfs`

to joining channels. `Nick` returns the current nick of the user. `Raw` allows reading and writing of raw irc commands (not normally used). `Pong` is a file that returns the delay of irc server responses to irc ping commands periodically sent by *ircfs*. *Ircfs* users can use this to detect disruption of the irc connection between *ircfs* and the irc server, and the styx connection between *wm/irc* and *ircfs*. `Event` returns a line for each change of the user's nick, and added and removed channels/users (due to `join` or `part` commands, or queries from other users). Reads on `raw`, `pong` and `event` block until data becomes available.

Each irc channel and user is represented by a directory. These directories directly map to windows in *wm/irc*. Again an example:

```
% echo 'join #inferno' >ctl
% cat 2/name
#inferno

% ls -l 2/
---w--w--w- M 2 ircfs ircfs 0 Sep 28 15:45 2/ctl
--rw-rw-rw- M 2 ircfs ircfs 0 Sep 28 15:45 2/data
--r--r--r-- M 2 ircfs ircfs 0 Sep 28 15:45 2/name
--r--r--r-- M 2 ircfs ircfs 0 Sep 28 15:45 2/users
% echo hi! >2/data
```

The example above shows how to join a channel and say something. Note that this is normally done in *wm/irc*, with the command `/join #inferno`, after which the message `new 2` would be returned on the `event` file, and a new window for directory 2 would be opened by *wm/irc*.

A special directory 0 always exists, e.g. the irc server message of the day. Each directory has a `ctl` file for writing commands (all those accepted by the top-level `ctl` file, plus some only for channels/users), a `data` file for reading and writing text, a `name` file for reading the name of the channel/user, and a `users` file from which changes of user presence can be read, i.e. users joining, leaving or renaming. Reads of the files `data` and `users` block until data becomes available. Lines written to the data file are sent to the channel/user as text. Lines other users write, or meta messages such as users joining/leaving or a change of the topic, are read from the data file with a character prefixed to indicate the type of the line (text, meta information). The `users` file returns lines when users join/leave the channel. This is used by *wm/irc* to provide tab completion for names.

Wm/irc

Wm/irc is a user interface consisting of three parts: On the left a list of names of channels/users, for each of which *wm/irc* maintains a window†. In the middle/right a text area that holds text from the data file, showing the text of the currently selected *window*. At the bottom is a text field for typing text and commands.

Wm/irc can handle multiple *ircfs* file trees, connections to multiple irc servers. Multiple file trees are typically exported using one *styxlisten*(1) and mounted on the machine running *wm/irc*. Note that Inferno's *styxlisten* and *mount*(1) can authenticate and encrypt the connection.

Wm/irc can be started with multiple paths of file trees as arguments. Paths can be added and removed during operation. For each file tree, the status window (special directory 0) is opened for executing commands on, e.g. `connect` to get started. New windows are created for new channels/users, as indicated by the continuously read `event` file.

The term *window* is misleading, it is just a Tk text widget.

Wm/irc stays informed about the user's own name for each irc file tree, and will highlight windows with lines containing the user's name. Additional patterns to highlight can be specified on the command-line. Unread windows with highlighted text have an = before their name in the list on the left, unread non-highlighted text a +, meta messages a – and delayed status windows are marked with a ~. In the text area itself, highlighted text has a yellow background.

Wm/irc has keyboard shortcuts for navigating among windows, e.g. to the next window with a highlight or the previously selected window. Clicking on the name in the list switches to that window.

Text can be copy-pasted with acme-like chording. A plumb button allows selected text to be plumbed. Searching in the text area (reverse by default, from most to least recent) is done from a dedicated text field. Matches are marked by an orange background.

A line typed in the text field is written to that window's `data` file when *return* is pressed, unless it starts with a single slash. A single slash makes the line a command. If the first word of the line (minus slash) is `win`, the remainder is interpreted by *wm/irc*. Otherwise, the line minus slash is written to the window's `ctl` file. For example, `/win quit` causes *wm/irc* to quit while `/quit` causes `quit` to be written to the *ircfs* `ctl` file, causing it to disconnect from the irc server.

Implementation details

Line counts:

```
1656 ./appl/cmd/ircfs.b
1446 ./appl/wm/irc.b
 397 ./appl/lib/irc.b
 125 ./module/irc.m
3624 total
```

The library parses and packs irc messages, converting between strings and adt's representing a message. *Ircfs* uses this library and otherwise just maintains the irc connection and the state of all channels/users, continuously handling irc and styx messages. The functions for handling a styx message, handling an irc message and handling writes to `ctl` files are the largest, followed by the code maintaining the data structures, including history for all channels/users and accounting for blocked styx reads.

Discussion and future work

Ircfs and *wm/irc* are separate programs with distinct functionality, but together provide an easy to use irc client. *Wm/irc* has practically no irc-specific code and the *ircfs* styx file tree is not very irc-specific either. *Wm/irc* could be reused for other instant message protocols, perhaps requiring small modifications to the styx interface. The styx interface has evolved during development of *ircfs* and *wm/irc*, each time adjusting the behaviour of one program to the needs of the other while keeping the code and mechanisms simple.

Ircfs only maintains a single irc connection. To connect to multiple servers, just start multiple *ircfs*'es. Of course, *wm/irc* does support multiple *ircfs*'es, and multiple *ircfs'es* can be exported on a single styx connection.

Most irc clients reconnect to the irc server when disconnected. *Ircfs* does not. First, I haven't needed that feature since the machine I run *ircfs* on has a very stable internet connection (the machine I run *wm/irc* on does not however). Second, it can be tricky to determine whether a disconnect should be followed by a reconnect. Constantly reconnecting is rarely appreciated by server operators. Perhaps a third reason is that such a feature would require quite some code, especially if channels must be joined automatically too.

Channels and users are represented by a directory in *ircfs*' styx interface. The names of

those directories are unique numbers, not the names of the channel/user. I cannot recall the original reason for this choice, but there are problems with directories named after the channel/user. First, irc is case insensitive for channel and user names (with quirks for special characters), so there is no unique or even a canonical name for a channel. Second, users can change their name, requiring a change of the directory name, making the path of open file descriptors unusable for e.g. reopens or opens of the `ctl` file given a previously opened `data` file. In short, the semantics of such directories are tricky while the semantics of numbered directories are not.

Not all irc commands have been implemented, e.g. those used by irc server operators. I have not needed those commands, and because documentation of irc commands is often incomplete and/or does not match practice, it makes little sense to write code them.

Wm/irc could be made to start up faster. Many files are opened and the Tk interface is updated a lot while starting. More files could be opened concurrently at start up, and Tk updates batched. However, *ircfs* does not distinguish existing state from new state to users of its files, so *wm/irc* cannot know when the start up phase ends. It would also require quite some code, while recently added more concurrent file opening has improved start up time already.

A larger issue that will require some redesign: *Ircfs* and *wm/irc* have no good way to know whether some lines from `data` files have been read by the user. This sometimes causes messages to be overlooked. I have not yet found a satisfactory solution to this problem. Currently *ircfs* keeps track of which data has been read from a data file. Data that has been read before is returned in multiple lines per styx read request, new data one line at a time. When *wm/irc* receives multiple lines in one read, it knows those lines have been read before and will never highlight these lines. This mechanism is inaccurate for two reasons. First, some irc directories contain only one line and thus will always appear unread and potentially cause highlights. Second, and more serious, any running *wm/irc*, or any other user of the *ircfs*, will cause data to be marked as read. There is currently no direct relation to human interaction with e.g. `wm/irc` and data being marked as read.

Wm/irc opens all windows by default. In same cases, e.g. when on a low-bandwidth connection, it is not desirable to open all windows or real all history. *Wm/irc* has an option to keep all windows closed by default. The *wm/irc* commands to open, close and list (unopened) windows have not been used a lot and might need improvement. *Ircfs* also has a mechanism to limit the amount of history to send: by a *wstat*(2) that only sets the `length` field. This is not a very clean mechanism, and at least too bothersome to tool-users of the `data` file, when history must often be ignored.

Connecting to an irc server is done by the *connect* `ctl` command (or *reconnect* to reuse the last parameters). This styx write for these commands does not return until the connection succeeds or fails. This can take some time since some irc servers stall the connection while verifying it. *Wm/irc* can handle this now (before this was fixed it would block the entire user interface), but other users of *ircfs* might find this behaviour troublesome.

Wm/irc adds colors and underlines to the editable text areas. Editing sometimes causes the mark up to be lost or mangled due to how these are configured with Tk. I doubt this problem has an elegant solution.

Wm/irc allows acme-like chording (in both *text*(9) area and *entry*(9) fields, each requiring different code), and plumbing by clicking a button. It is a pity this has to be implemented by *wm/irc*. It would be nice if this code would be provided as generic Tk functionality.

KNX Implementation for Plan 9
Work In Progress

Gorka Guardiola Múzquiz
Enrique Soriano Salvador
Francisco J Ballesteros

Laboratorio de Sistemas
Universidad Rey Juan Carlos

ABSTRACT

We are working on a project to interconnect and control home devices from a hive of mini computers. We have started with sensors and actuators for the home. In order to do this we have written an implementation of the Knx [1] protocol stack for Plan 9. We plan to use gumstix computers and therefore we are porting Plan 9 to this platform.

Introduction

Nowadays homes are full of devices begging to be interconnected and controlled. Big screens, audio devices, computers and computer peripherals, switches, lights, sensors for temperature and movement, alarms and possibly many others depending on the tastes and economic possibilities of the owner. We envision a network interconnecting these devices and a computer controlling them to make them an integral system which gives the user a seamless experience.

We are working on a project to export these devices as filesystems using small minicomputers. As part of this project we have started a port of Plan 9 to the gumstix (a small arm pxa270 based minicomputer) and written a port of the Knx protocol stack to Plan 9. This article describes the Knx usb driver we implemented and the development of a Knx protocol stack under Plan 9, and its filesystem interface. The filesystem interface is still under development.

Knx

Knx is the successor of EIB, European Installation Bus. Knx lets you connect a network of sensors, actuators and small devices using a Knx bus, which defines various physical mediums including a dedicated cable, and an insanely complicated protocol. Knx is an standard supported by many companies and it is easy to find sensors and actuators which have been reported to be quite robust and which consume very low power.

We have finished the Knx protocol development and we are now working on a good interface to export the filesystem.

Most of the available devices to control Knx networks just export the network through a gateway which lets you inject packets into the bus. This approach just translate the difficulties of programming the Knx protocol adding another layer without abstracting the problem itself.

We plan to export a synthetic filesystem in the Plan 9 tradition, which will make it much more simple to control the devices. Also, the devices themselves can run many different "applications", because they can be programmed in some metalanguage

combined with assembler. We want to control Knx devices as much as possible, but we are not particularly concerned in programming the devices themselves, just with configuring and controlling them.

The devices can be programmed using the standard tools to program them with the default binaries provided by the vendors as we have intelligence in the computer to do the rest for us. Once programmed, the devices offer "objects", data types with a network address which can be accessed through the devices or through their special object address. We plan to have unique object addresses throughout the device and the network if possible. This way, we can control each device and its properties from a computer (the gumstix or the client to its file server).

Knx USB device

Knx defines a protocol for a USB [2] "coupler" to connect to the bus and act as a bridge between the bus and a PC. The first thing we had to do was write a driver to control the "coupler" itself.

The Knx USB devices provides us with two 64 bytes interrupt endpoints besides the control endpoint, used for communicating with the device. The devices announces itself as an HID [3] device and uses HID report headers and bodies format. The sequence low nibble is used for multisequence packets, but we have not found a device for which we needed to implement this.

The USB device has two parts, the Bus Access Server (Features) and (possibly) various Emi Servers. Emi, external message interface is the name of the protocol as seen outside of the bus. The Knx messages have two representations internal (Imi) and external (Emi) and the Emi Servers translate between one and the other. The Emi Servers are more than that, they filter and rewrite incoming packets depending on their layer they are configured to understand and the address of the packet. Essentially they implement all the protocol stack. There are two kind of packets which we send to the USB device in the report body. Packets directed to the Bus Access Server, which decides which Emi Server to turn on and how to configure it, and Emi packets which are intended to be sent to the BUS.

Knx network protocol

Knx uses the ISO request, confirmation, indication structure for its communications. Each time a request message is sent the sender receives a confirmation from his local Emi Server and the receiver receives an indication. We have ignored confirmations as they are generated locally (so they do not confirm anything) and when we are waiting for a response we always set a timeout, in some cases defined by the standard and in many other cases just a sensible limit. There are also retransmissions, so there is no point in confirmations at all.

The Knx standard defines a the whole link/transport/network layer and modes for each of this layers (and extra routing properties) for the Emi Server. We configured the Emi Server in the (link layer, bus monitor), which is an essentially transparent mode so that we could have complete control and see what was exactly happening with the devices themselves. Even in this mode connected communications between remote devices are not seen, which makes it difficult to program a real sniffer (other than continuously changing address). In any case, at least using the lowest possible mode lets us do broadcast to program an address an in general program operations without having to change modes continuously at the risk of creating a race condition and losing packets. When the Emi Server is configured in an higher layer the packets arrive with the upper part erased (zeros) or do not arrive at all.

As a consequence of having the device configured at the lower layer, we see broadcast packets, and other packets we could ignore (confirmations), and we have to deal ourselves with retransmissions and timeouts.

The Knx stack

We created two I/O procs to take care of the blocking I/O of the USB endpoints. The rest of the protocol stack is composed of threads all living in the same proc. `Usbwriter` and `usbreader` are threads representing the I/O procs and which serve a channel each. These threads convert the raw USB packets int structs, taking care of the HID report headers and the Knx transport headers. This interface permits the stack to send Bus Access Feature packages whenever needed and inject them at low level while at the same time not having race conditions with other layers of the stack.

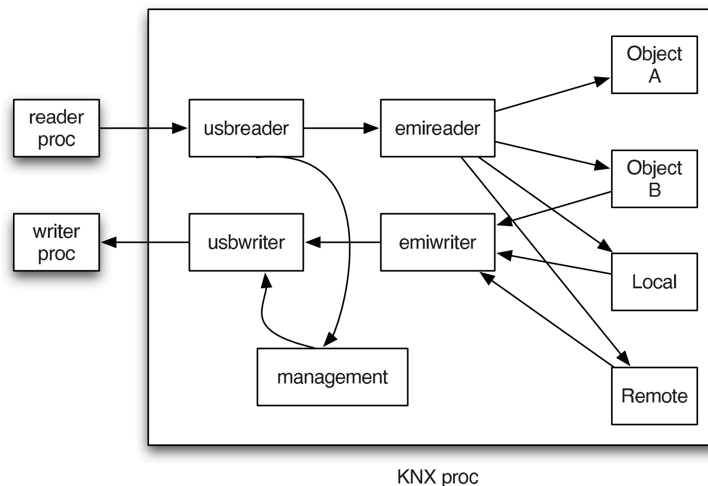


Figure 1: Architecture of the Knx driver.

Two other threads, `emireader` and `emiwriter` read from these channels, decode and interpret the packets and send them to the appropriate thread. There is an another thread representing each addressable Knx object plus three extra threads, **Management**, **Local** and Remote. **Management** is the thread representing raw non-emi packets, essentially Bus Access Features. **Local** injects emi packets in raw, normally used to configure parts of the Emi Server. **Remote** is used for injecting raw Emi packages into the bus for things like configuring devices which have no address yet.

We have implemented stop and wait, taking care of retransmissions as part of the sending functions, to make things simpler. At the moment, there is no need to optimize throughput in any case.

Current state and Future work

The protocol stack as it is lets us already program addresses for the devices and control all the devices we have. We can read the state of switches, control binary outputs and read the state of temperature and light sensors. The Gumstix port is already working. The only work left is some minor fixes and device drivers. The ethernet device driver is working in Inferno and has to be ported to Plan 9.

We cannot (yet) discover the objects within a device we have not programmed ourselves. We cannot program Knx devices, though this was never our goal it would be nice to be able to write a binary image on them to program them. A filesystem to control the objects in under way. We plan to have a directory representing each of the object threads, though we are working on the how many files it will contain and their semantics.

References

1. K. Association, KNX Association Official Website, http://www.knx.org.
2. Compaq, Hewlett–Packard, Intel, Lucent, Microsoft, NEC and Philips, USB 2.0 Specification, 2000.
3. U. I. Forum, USB Device Class Definition for Human Interface Devices (HID) , 2001.

A File System for Laptops

Brian L. Stuart
University of Memphis
blstuart@bellsouth.net

ABSTRACT

Laptop computers do not fit well into the type of network that is built around one or more file servers. The need to approach diskless operation, keeping data on the file server is in direct conflict with the need for mobility of data. Here I report on work currently underway to develop a filesystem for laptops that resolves this conflict by transparently keeping copies on both the laptop and the file server.

Problem Statement

Experience has shown that there are many advantages to a file server separate from other systems in a network. Typically a file server is backed up more regularly than an individual's machine. The file server makes it easier to move to a new machine when the old one is replaced. It also allows users to make use of any machine available and still see their data. Finally, improvements to the hardware of a file server benefits all of the user community in a way that's much easier and often more cost-effective than upgrading all user machines.

In recent years, however, there has been a significant shift in computing resources. Unlike in the late '80s, most of us now use laptop computers at least part of the time, and many of us use them almost exclusively. Last year (2008), marked the first time that sales of laptops exceeded those of desk-top PCs. Unfortunately, there is a fundamental mismatch between the laptop computer and the file server-centric network. The very nature of a mobile machine demands that its persistent data be carried with it, where a file server is essentially data storage separate from an individual's machine.

Previous Solutions

More often than not, people who use laptops in an environment that includes a file server, approach things from the perspective that the file server is essentially a backup device for their laptop. Solutions created from this perspective generally make use of tar or rsync or something similar. Periodically, the contents of the laptop are mirrored to the file server. Some users take a full tar image each time, others use rsync or something similar to update the server. Approaches such as tra[1] and unison[2] improve on this synchronization by flagging a conflict when a change is made on two different machines between synchronizations.

Some have designed distributed file systems with an eye to handling the needs of laptop users. The most well-known is Coda[3]. One key element of these distributed systems is the push-like behavior of the file server. When a file changes on the server, it sends out a notification to any clients who are subscribed to that file so they can retrieve the updated version.

A Different Perspective

To a some degree the approach most people take with laptops is to treat the laptop as the primary repository of data and the file server as a form of backup. If we reverse this perspective and consider the file server primary and the laptop storage as a cache for times it's disconnected, we are led to a little different approach. In particular, when connected to the network, the laptop behaves as a write though cache. All writes are immediately relayed to the file server. Data read from the file server while connected are also stored on the laptop, making it available when disconnected. During times when the laptop is disconnected, it behaves as a sort of write back cache. Writes are recorded in a write log which is played back to the file server when the laptop is again connected to the network. Laptops that move among multiple home networks

can be handled by multiple instances of the file system with one running in connected mode and the others in disconnected mode when the laptop is on one of the home networks.

Within this perspective, there are several desirable characteristics we would like to have. First, we would prefer not to have to modify the file server. Depending on the environment, we might not have the privileges necessary to make changes. Furthermore, modifications to the file server would have to be implemented for each server OS we connect to. Second, we want the implementation to run in user space on the client. As with the file server, changes at the system level require privileges we may not have and require different implementations for each OS we might run on some laptop. Third, we want the user to be able to function the same way whether the laptop is connected to the file server or not.

First Approach

When I first started to implement a file system based on this perspective, I began with Plan9 and implemented a simple file system that maintained a local copy while talking to a file server. This server is called lapfs. The general approach was to use ordinary *read*(2) and *write*(2) calls both to the local file system and to the file server. The precondition was that the file server be mounted in the local name space. For example, suppose the file server is mounted on /n/remote and we have directories, fscache and fs in the home directory. The lapfs server is mounted to fs where it presents a namespace which is a copy of that rooted at /n/remote. Fscache is where lapfs keeps its cache. In this first implementation, lapfs maintained its own directory structures stored in regular files within fscache. When connected, on each open, lapfs would query the file server to determine if the file's last modification time was newer than the cached copy. If it was, or if there was no cached copy, lapfs would copy the file from the file server to the cache before completing the open request. Reads were generally served from the cache and writes went both to the cache and to the file server, though some experiments waited until the file was closed before sending writes.

After the success of the initial experiment, it was clear that I needed to implement this design in Linux as that is what my laptop at the time ran the vast majority of the time. This reimplementation was done using fuse. Fuse was chosen so that flapfs (as that version was called) could present POSIX semantics including symbolic links. This was important for some potential users at the time who wanted flapfs to serve their entire home directory, and some applications such as evolution were heavily dependent on symbolic links. The implementation was used heavily by myself and experimentally by co-workers. For two to three years, my laptops maintained caches of both my home file server and the file server at work. This arrangement worked quite well. I no longer had to worry about whether I had copied the latest version of a grading file to the laptop before going to work. When the time came to replace a laptop, I didn't have to retrieve any of my data from the old machine. I could simply install flapfs and begin using the file servers, and the cache would begin building on the laptop.

A New Approach

Recently, I once again had to replace my laptop. Even before making sure that the version of flapfs on my local file server was indeed the latest, I began to consider another, cleaner approach. The key observation was that there is no reason why the laptop file system should have to query the status of files on the file server all the time. When connected, all operations can go directly to the file server with appropriate copying to the cache. A substantial portion of the flapfs code had been devoted to modification time checking, and the associated copying operations. This along with attempts to improve performance mitigating the large number of *stat*(2) calls that were being made on directory searches was the most complicated part of the code. If we no longer worry about the versions, we can work at the level of individual operations, rather that at the level of files. In effect, the file system becomes a bidirectional form of tee.

Because I've been more immersed in Inferno than Plan9 of late, I decided to implement this new approach in Limbo. It was only after I began that I realized that this makes the file system operable on any system that supports Inferno, not just those that have a port of fuse. (However, there are a few open questions about using it on systems without P9Ps ability to mount 9P/Styx servers.)

Rather than use ordinary file calls to interact with the file server and the cache, it made more sense to establish channels to them and pass Styx messages back and forth. If the server (or

even the cache) is over a network connection, then the channel is established with *sys->dial*(2). If it is provided by a local server, *sys->pipe*(2) and *sys->export*(2) are used to establish the channel. In most cases, the file server will be on a network connection and the cache will be managed in the host OSs namespace by way of the #U server. In this implementation, all T messages from the client are forwarded to the file server, and all file server R messages are relayed back to the client. Client T messages, with the exception of Tread and Tstat, are also passed to the cache. In the case of Tread and Tstat, lapfs waits for the Rread and Rstat coming back from the file server. It then transforms them into corresponding Twrite and Twstat messages that are passed on to the cache. Special treatment must be given to Twalk. If both the file server and the cache produce successful results indicating that the full path has been traversed, then we need do no more. However, if the cache doesn't have all of that branch of the tree, then we must create it. From wherever the traversal stopped in the cache to the point where the traversal stopped in the file server, we create directories in the cache, except for the last path component where we might create either a directory or a file. In the current implementation, the handling of walk is the most complex part of the code.

Status

As of the time of this writing, the Limbo version of lapfs, working at the message level, has been implemented with the exception of three significant mechanisms. First, the disconnected operation has not yet been implemented. For this, the messaging strategy changes to simply relaying all client requests to the cache channel and all cache replies back to the client. This mode of operation also requires the second missing mechanism, that of the write log. The third major limitation is in the walk implementation. What is currently coded works as long as the cache can make one step of the walk so that it returns an Rwalk message. If, however, it is unable to traverse the first path component, then it returns an Rerror message. Path creation in response to the Rerror message has not been implemented as of this time.

It should also be pointed out that the current implementation has been *very* lightly tested. There is no doubt that significant bugs will be encountered and fixed as it is used on a regular basis.

Finally, the current implementation is based on a simplifying assumption, namely that the client will not have more than one T message in flight at a time. Lapfs serializes requests in the sense that it will not take or process another T message until the previous R message has been processed and returned. This avoided some fairly significant complications related to keeping T messages around until we know we can drop them. They would be needed because in transforming Rread to Twrite for the cache, some of the information in the original Tread is necessary.

Further Development

Obviously, the first step is the completion of the missing features. This must be done before it can be used seriously enough to evaluate further. However, once it is working to a functional level, it will be important to determine if it imposes a significant performance hinderance. This is of concern, because the Linux (and FreeBSD) flapfs did create noticeable delays in many circumstances. It also remains to be seen whether the serialization of requests is a significant limitation. If it is, then the bullet will have to be bitten and a list of T messages will have to be maintained and then referenced upon receipt of the corresponding R message.

References

[1] Cox, R and Josephson, W., "File Synchronization with Vector Time Pairs," MIT Computer Science and Artificial Intelligence Laboratory Technical Report, MIT-CSAIL-TR-2005-014, Feb 2005.

[2] Pierce, B C and Vouillon, J., "What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer," Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[3] Satyanarayanan, M, "Coda: A Highly Available File System for a Distributed Workstation Environment," Proceedings of the Second IEEE Workshop on Workstation Operating Systems, Sep 1989.

Levitating Across the River Styx

Jeff Sickel [a]

Corpus Callosum Corporation
Evanston, Illinois 60202

ABSTRACT

The Styx (9p) protocol has been well documented for use in various distributed systems [1, 2]. Demonstrations have proven that it works for communication with embedded devices [3, 4, 7]. This paper presents an implementation of 9p for the 16-bit dsPIC33 family of digital signal controllers. It is used to collaborate multiple distributed nodes to achieve stable aero-acoustic levitation of a sample by tuning sound pressure levels and managing spin control[b].

1. Introduction

Aero-acoustic levitation (AAL) has been achieved in the past using analog systems with purely manual controls [8, 9]. A newly designed AAL instrument is being built based around a cluster of digital control boards. Each board includes a dsPIC33 embedded controller and an FPGA to handle manual inputs paired with position feedback control sensors used to drive a transducer and produce a stable acoustic field during experiments where significant changes in temperature will be applied to a sample. Temperature changes modify the speed of sound and thus sound pressure levels at the desired focal point. The distributed system of transducer control boards is used to calibrate and adjust acoustic phase and amplitude along three axes to levitate and hold a sample in a fixed position during solid to liquid-phase processing studies.

In addition to a manual control interface, this new system implements a fully digital controlled interface available on an end user's terminal. This newly designed system uses 9p for tuning core parameters on each node as well as communication with external terminals. Exporting all the node sensor data to a single namespace on a user's terminal provides for clean logging and real-time analysis of state changes during an experiment.

1.1. Aero-acoustic Levitation

Research into containerless liquid-phase processing of materials led to the invention of aero-acoustic levitators [8]. The combination of gas jet (aero) and acoustic forces help to stabilize the sample in a levitated (containerless) field such that rapid heating and

[a] jas@corpus-callosum.com

[b] Additional advice and leadership provided by Physical Property Measurements, Inc.
Paul C. Nordine, Steve Finkleman, and Eduado Lopez-Gil.

cooling techniques can be applied. In this example it is principally used to conduct experiments on samples that may be super heated to temperatures above 2700°K and rapidly cooled without requiring a crucible or other physical container. Such experiments test the viscosity and surface tension of a sample within a controlled environment able accommodate key state changes important for theoretical and applied material science studies.

1.2. Controller Boards

A distributed feedback control system is used in order to achieve aero–acoustic levitation of samples during significant fluctuations in temperature. A gas jet counteracts gravitational force and is controlled by the user or a program interfacing with an electronically controlled flow valve. Three acoustic axes, a pair of transducers each, are used to provide levitation, stabilization, and spin control of a sample. Position detectors are used perpendicular to each acoustic axis, three in total, to aid in stabilization and to coordinate placement of a sample positioned at the intersection point of the gas flow and the two lasers used in melting the sample. Each component is wired to one of eight controller boards sharing a system bus with software running on the dsPIC33.

By tuning the acoustic phase of each axes to focus a standing wave above the gas jet and near the ideal point for heating a sample, the deadline in which any system event needs to respond becomes quite long. This leaves more than enough time for handling 9p messages at relatively regular intervals. By implementing 9p on the dsPIC33 embedded controller there ends up being enough memory and free cycles to handle feedback loop calculations without requiring further development like the *Styx IP–Core* on an FPGA [7]; however, the option is still there for future research.

2. The River Styx

Each of the eight nodes in the AAL cluster resides in a single chassis with three communication channels: one exposed through a RS–232 interface, and two through the backplane. The first dsPIC33 code used a simple protocol requiring the use of a dumb terminal connected to each board over the RS–232 interface. This configuration allowed for single board initialization and verification, but made debugging inter–node communication over the 485 and SPI busses all the more difficult.

After reviewing file system interfaces for mobile resources [1, 5, 6], the case was made to implement 9p not just for the serial communication from the host terminal to each control node, but as the principal means of communication along the backplane connecting all the nodes in the cluster. This conceptual switch to responding through the same protocol over each serial interface made the design and debugging of the system all the more practical. A master node constructs a representation of the rest of the cluster and exports its namespace using 9p over its RS–232 port. All other nodes are dedicated to reporting the state of their various sensors and calculating responses based on their dedicated feedback loops.

2.1. Embedded Server

Each node with a dsPIC33 serves a single–level namespace with two files: ctl, and status. As with devices on Plan 9 and Inferno (e.g., *uart(3)*) the ctl file is used to write configuration parameters to the board. The status file returns the measured state of all the sensors accessible to the embedded controller: output voltage, output current, phase,

mic amplitude, mic phase, temperature, etc.

Future implementations could expand the board to provide register and memory details of the PIC, allowing for even greater debugging options as well as dynamic updates beyond the simple scope of variables required for achieving levitation.

2.2. Embedded Client

A single master board is used as a gateway between all the controller nodes and a user's computer terminal. After board initialization, the master node scans the backplane for all other connected boards, and uses 9p to set up a multilevel namespace representing the system. All the non-master nodes are hot pluggable, so the master node's exported representation of the system will change as a node is disconnected or inserted. In turn, the master board exports a namespace over its RS-232 serial interface, providing a synthesized file system to the user's application with a synopsis:

```
mount /dev/eia0 /n/aal

/n/aal/ctl
/n/aal/status
/n/aal/transtatus
/n/aal/[0-6]/ctl
/n/aal/[0-6]/status
```

The transtatus file contains the last polled state of all the transducer controller nodes, numbered 0-5, eliminating the need to re-poll each node in order to respond to the application's request.

3. Experimental Control Application

The whole exercise of getting 9p on all of these embedded controllers is to provide a consistent API for firmware development and expose a relatively simple interface for end user application programmers to design and support experimental controls over various sample materials in order to have precise position and spin control. There are multiple built-in feedback loops that adjust the system's parameters in real-time when enabled, though there is programmatic control over these events as well. All changes get reported back over timed reads by the master node (host application) of all other status files. Doing so allows for an experimenter/programmer to create events that can be programmed through scripts and run once the sample is levitated. A typical startup example is:

```
# create namespace
bind -a '#t' /dev
mount -bc /dev/eia0 /n/aal
linkflowmeter /dev/eia1
mount -a /net/flowmeter /n/aal
linkheatsource /dev/eia2
mount -a /net/heatsource /n/aal

echo off > /n/aal/heatctl
echo off > /n/aal/flowctl

# acoustic calibration routines
echo findq > /n/aal/ctl
echo pick > /n/aal/ctl
echo phase > /n/aal/ctl
```

```
# prep for sample insertion
echo A .2 > /n/aal/ctl
echo 1000 > /n/aal/flowctl

# change phase on one axis
echo p+1 > /n/aal/[01]/ctl
# increase amplitude from top transducers
echo a+1 > /n/aal/[1,3,5]/ctl

# pseudo code to decrease amplitude over 30s
for(i in ‘{seq −1 0}){
    echo a−1 > /n/aal/ctl
    sleep 1
}
```

4. Conclusion

The use of 9p for inter-process communication in a distributed cluster provided a valuable interface for reading and writing sensor data on a dsPIC33 within certain deadline constraints on each of the eight nodes in the cluster. It also allowed for application development to use a simple file hierarchy to monitor and control a running system. Initial shell script prototypes were used on Plan 9 or Inferno depending on the developer’s host system. Subsequent applications were written in Limbo and provided as reference implementations for the use of 9p over the RS-232 interface.

5. References

[1] R. Pike and D. M. Ritchie. The styx architecture for distributed systems. *Bell Labs Technical Journal*, 4(2):146-152, June 1999.

[2] R. Sharma. Distributed Application Development with Inferno. In *DAC ’99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 146-150, New York, NY, USA, 1999. ACM,

[3] C. Locke. Styx-on-a-brick. Vita Nuova Limited, York, England, June 2000.

[4] B. Ellis. 9p for embedded devices. In *Proceedings of the Third International Workshop on Plan 9*, pages 39-42, October 2008.

[5] P. Stanley-Marbell, Implementation of a distributed full-system simulation framework as a filesystem server. In *Proceedings of the First International Workshop on Plan 9*, 2006.

[6] N. C. Audsley, R. Gao, and A. Patil. Towards a File System Interface for Mobile Resources in Networked Embedded Systems. In *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 913-920, Prague, 20-22 Sept 2006.

[7] N. C. Audsley, R. Gao, and A. Patil. *The Styx IP-Core For Ubiquitous Network Device Interoperability*. Perspectives in Pervasive Computing. Cambridge University Press, London, UK, October 2005.

[8] J.K. R. Weber, D. S. Hampton, D. R. Merkley, C. A. Rey, M. M. Zatarsk and P. C. Nordine. Aero-acoustic levitation: A method for containerless liquid-phase processing at high temperatures. *Review of Scientific Instruments*, 65(2):456-465, February 1994.

[9] J.K. R. Weber, J. J. Felten, B. Cho, and P. C. Nordine. Design and Performance of the aero-acoustic Levitator. *J. Jpn. Soc. Microgravity Appl.*, 13(1):27-35, 1996.

How to Make a Lumpy Random-Number Generator

Michael A. Covington

Institute for Artificial Intelligence
The University of Georgia
mc@uga.edu

ABSTRACT

Normally, random-number generators are designed to produce numbers with a uniform distribution. The sum of uniform random variates has a bell-curve-shaped distribution. Using bell curves like wavelets, individual uniform random variables can be summed to produce arbitrary nonuniform distributions. The result is a simple, customizable non-uniform random-number generating algorithm that has been prototyped on Plan 9 and is equally suitable for other computing environments, including very small embedded systems.

1. Problem definition

Normally, random-number generators produce uniformly distributed values. However, nonuniform random numbers are needed for a number of purposes. In simulation, one often needs random numbers conforming exactly to the observed or theoretical distribution of an input variable, in order to produce an authentic distribution of simulated output [1].

In other situations, the requirements are much less precise, but random numbers with a preference for certain values or ranges are still desired. Examples include equalizing wear on machinery by having a machine return to approximately but not exactly the same position each time; introducing "dither" to avoid unwanted synchronism with external processes; and correcting for nonuniformity in some process downstream from the random number generator.

Traditionally, nonuniform distributions are generated by transforming the output of a uniform random-number generator, often using elaborate floating-point computations. In this paper I outline an alternative that is especially suitable for the latter set of cases, where quick computation is more important than hitting a specified distribution exactly.[1]

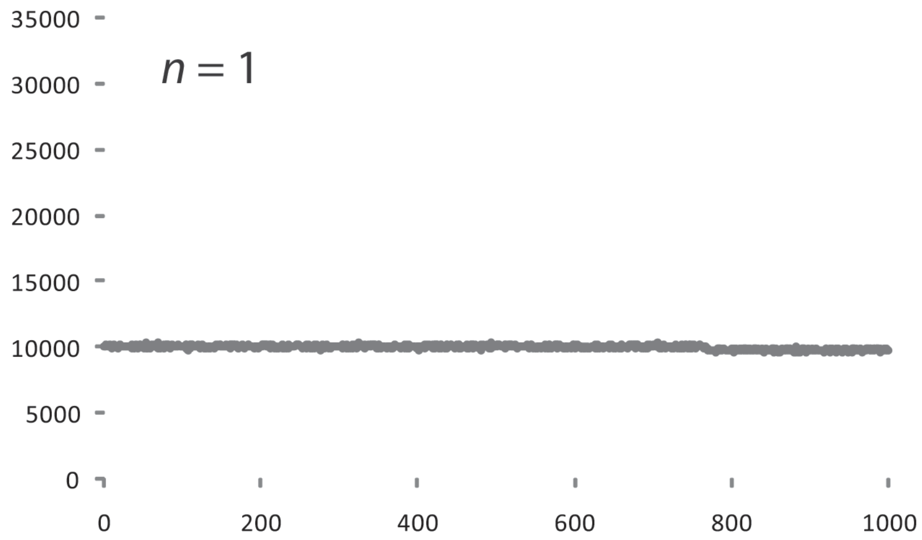
2. Approach

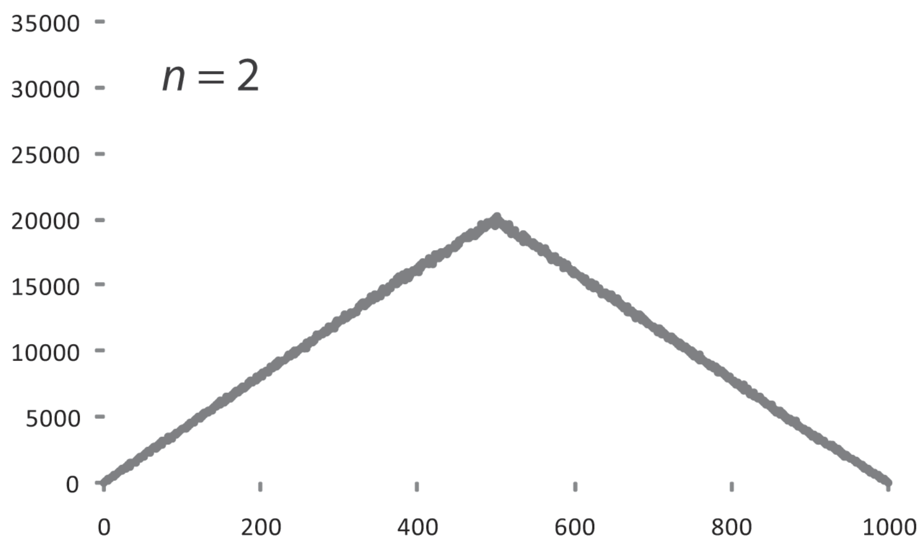
As Fig. 1 shows, the sum of n uniform random variables is an $(n-1)$th-degree polynomial approximation to a normal distribution (bell curve) ([2], p. 22). For practical purposes, the curve with $n=3$ is smooth enough.

Thus, one can generate a bell curve distribution by merely generating three random numbers each time, adding them, and dividing by three.

Bell curves can be used like wavelets to synthesize more complex curves. For example, the distribution in Fig. 2 was synthesized from a nonzero baseline mixed with three bell curves by the code in Fig. 5.

[1] This research was done for CORAID, Inc. (*www.coraid.com*) while the author was on summer leave from the Institute for Artificial Intelligence, University of Georgia (*www.ai.uga.edu*). The author thanks Brantley Coile for posing the initial question from which this project arose.





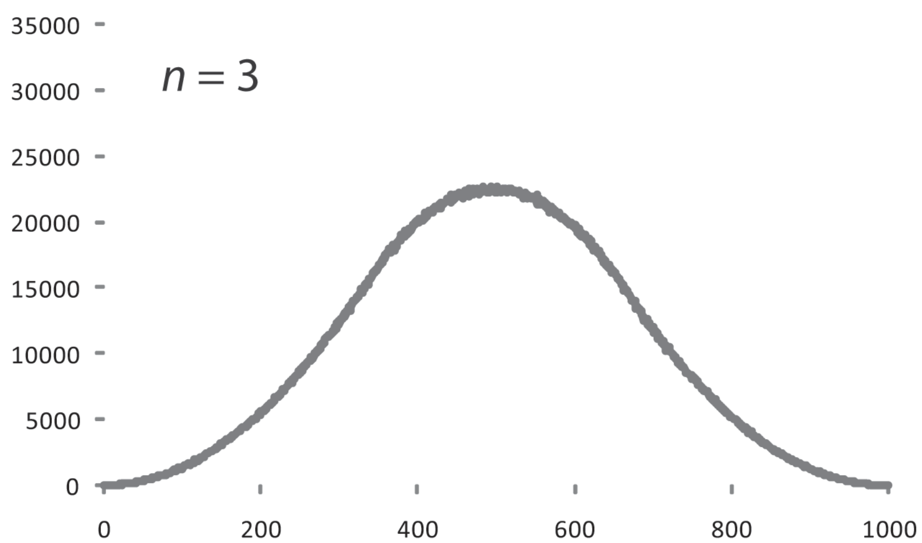
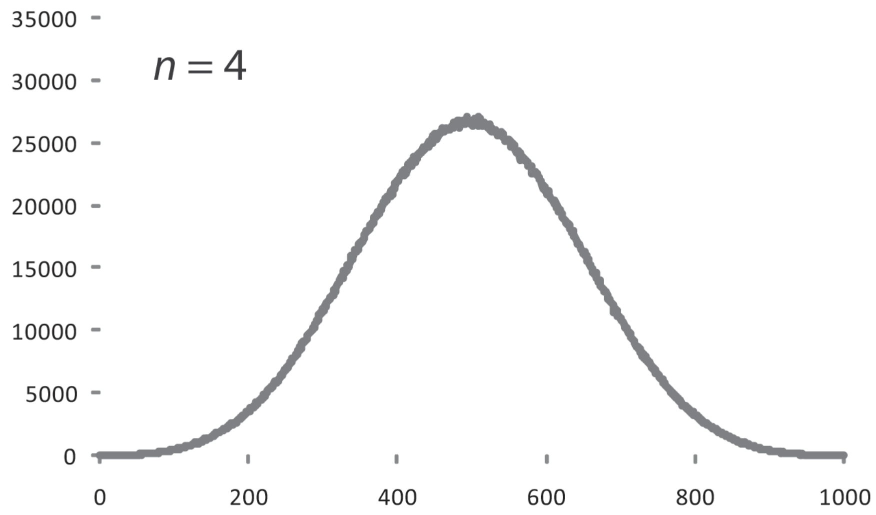


Figure 1: Histograms of the sum of n uniform random variables, from 10 000 000 trials.



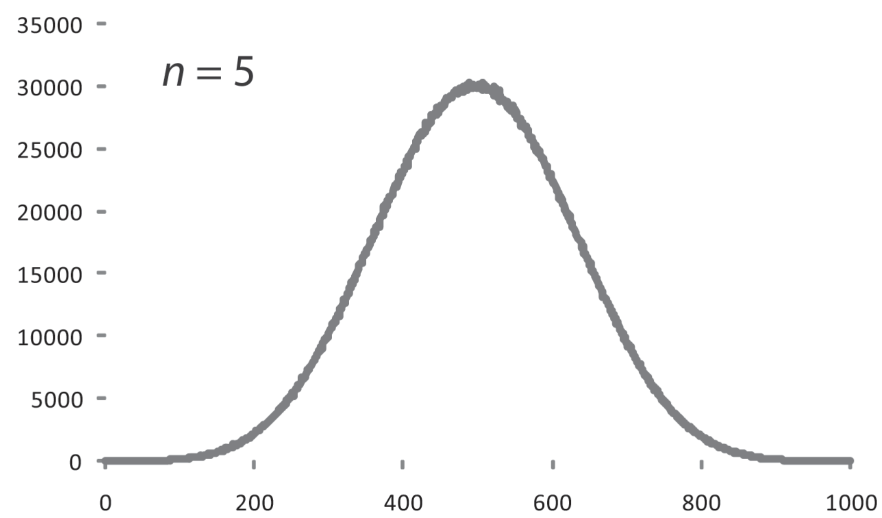


Fig. 1, continued.

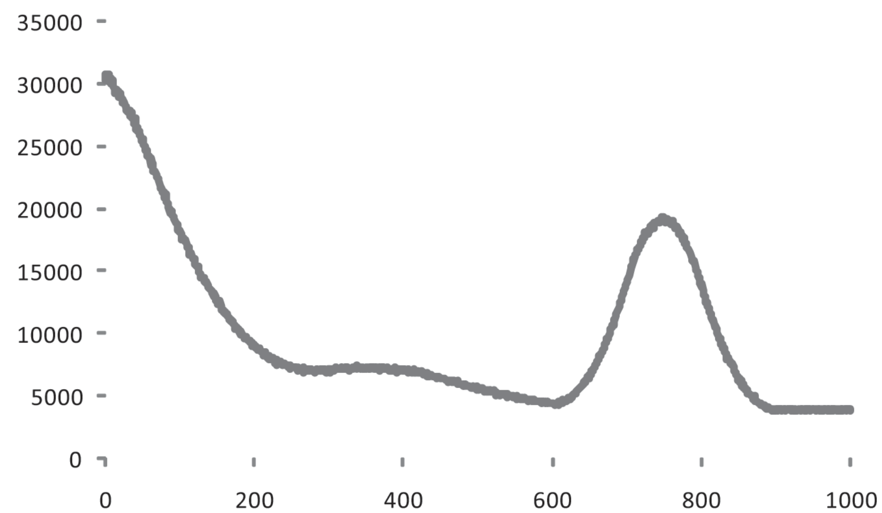


Figure 2: Histogram of custom random number generator in Fig. 5.

```
int genrand(int bmin, int bmax, int rmin, int rmax, int n) {
    // Generalized random number generator;
    // sum of n random variables (usually 3).
    // Bell curve spans bmin<=x<bmax; then,
    // values outside rmin<=x<rmax are rejected.
    int i, u, sum;
    do {
        sum = 0;
        for (i=0; i<n; i++) sum += bmin + (rand() % (bmax - bmin));
        if (sum < 0) sum -= n-1;          /* prevent pileup at 0 */
        u = sum / n;
    } while ( ! (rmin <= u && u < rmax) );
    return u;
}
```

Figure 3: Generalized random number generator (sum of n uniform random variables).

Bell curves lack one property of wavelets [3]: they do not have an average value of zero, and in fact they never dip below zero at all. Thus, adding another bell curve to a synthesized function can only raise it, not lower it. For histograms, this is not a serious objection because the height of the curve has only relative significance; the whole histogram can be raised or lowered by generating more or fewer random numbers.

To understand that bell curves can be used like wavelets, consider some arbitrary distribution $f(x)$, some approximation to it $g(x)$, and the difference $g(x) - f(x)$. Either the difference is a constant, in which case it can be filled by mixing in a uniform distribution, or else it has one or more "humps." Fill one of the humps with a bell curve sized to fit it, and you have a better $g(x)$ and a new $g(x) - f(x)$ from which that hump is missing. Repeat as desired until the fit is sufficiently close.

3. Implementation and testing

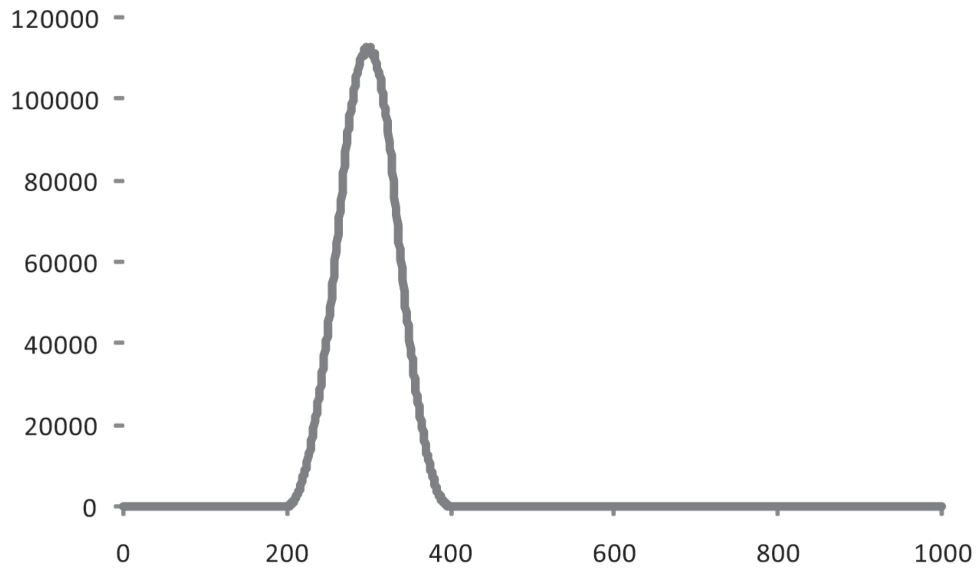
To put this idea to the test, the all-integer algorithm shown in Fig. 3 was coded in C and executed under the Plan 9 operating system [4] on a 1.87-GHz Pentium processor. Each call to *genrand* with n=3 took, on the average, less than 0.01 microsecond. The histograms in all the illustrations were made with this function, by downloading its output to a PC and graphing with Microsoft Excel.

The arguments of the function specify two ranges, the range that the bell curve should span and the range of values acceptable as output. Thus, it is possible to compress or truncate the histogram (Fig. 4). Naturally, if a substantial part of the bell curve is discarded, CPU time is wasted, but this is still a quick way to generate a partial bell curve.

In the algorithm, if the sum of individual random variables is negative, it is decremented by $n-1$ before performing the integer division by n. The reason is that without this step, there are more ways to get 0 than any other number. For example, if n=3, then not only do 2/3, 1/3, and 0 truncate to 0, so do $-1/3$ and $-2/3$. By shifting all negative results farther negative by $-2/3$, we get the latter two to truncate to -1.

Fig. 5 shows how the synthesis of a distribution is done. This particular function has a 40% chance of choosing the first call to *genrand,* a 30% chance of choosing the second, a 20% chance of choosing the third, and a 10% chance of choosing the fourth. Thus, the bell curves are mixed in the desired proportions.

One limitation inherent in this technique is that the random numbers need to fall within in



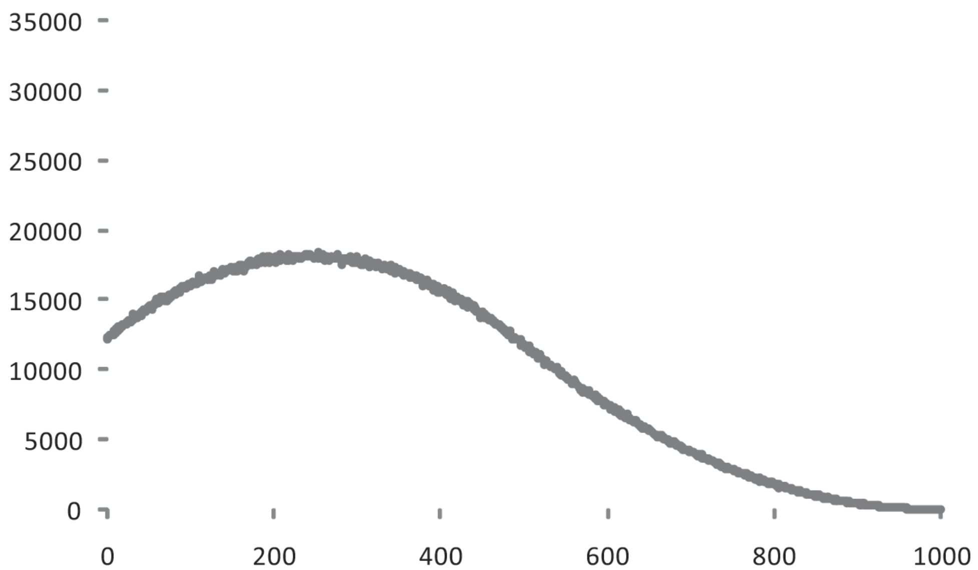


Figure 4: Generated third-degree ($n = 3$) bell curve spans any specified range of values. Bell need not fit within desired range; in that case, values outside range are generated and rejected.

```
int customrand(void) {
    switch (rand() % 10) {
        case 0:
        case 1:
        case 2:
        case 3:
            return genrand(0,1000,0,1000,1);    // flat baseline
        case 4:
        case 5:
        case 6:
            return genrand(-400,300,0,300,3);   // large peak beyond left edge
        case 7:
        case 8:
            return genrand(600,900,600,900,3);  // peak at 750
        default:
            return genrand(0,700,0,700,3);      // very low, broad peak at 350
    }
}
```

Figure 5: Sample code to interleave calls to `genrand` with different parameters, to combine multiple bell curves into the single distribution shown in Fig. 2.

a range considerably smaller than that of the underlying built-in random number generator. In Plan 9 C, *rand* produces integers from 0 to 32 767 inclusive. When taken modulo 1000, as in the examples, these are not quite uniform because (for example) there are 32 ways to get 767 but only 31 ways to get 768. This nonuniformity is just visible in the topmost curve in Fig. 1 but is usually negligible. If the modulus were 10 000 or 20 000, it would be serious.

References

[1] P. Bratley, B. L. Fox, and L. E. Schrage, *A Guide to Simulation,* 2nd edn. New York: Springer-Verlag, 1987.

[2] L. Devroye, *Non-Uniform Random Variate Generation.* New York: Springer-Verlag, 1986.

[3] I. Daubechies, "Wavelet transforms and orthonormal wavelet bases," in *Different Perspectives on Wavelets,* I. Daubechies, Ed. (Proceedings of Symposia in Applied Mathematics, 47.) Providence, R.I.: American Mathematical Society, 1993, pp. 1–33.

[4] Thompson, Ken (1990) "Plan 9 C compilers." *http://plan9.bell-labs.com/sys/doc/compiler.pdf*

Mails as (real) files

Francisco J Ballesteros
Laboratorio de Sistemas
Universidad Rey Juan Carlos

ABSTRACT

Understanding mail is a complex task. There are many standards involved and many formats for headers, text, and attachments. On the other hand, reading mail is simple: There is some text shown and one or more files included as attachments. This paper describes a mail system built by considering this. It stores mails in the file system as shown in a mail reader, and attachments decoded as regular files. Thus making file tools become mail handling programs.

Introduction

Long ago someone said regarding mail in Plan 9:

> *Seems our user base is growing larger than main memories of our imap servers. Anyone have any ideas to keep systems from running out of memory?*

By that time Plan B was using the file server program (Plan 9's *fossil*) also as a mail server program. To do so, the only requirement was to keep mail undecoded and stored in the file system as any other document would be. Today Nupas [1] can do the job, although we keep on using the tools described here.

A mail including some text along with several PDF documents is not different from a note made on a file along two PDF files. And that is the format used by the mail tools described here.

The editor used (be it *Acme* or *O/live*) becomes a mail reader as soon as there is a convenient way of producing mail listings. That can be done using *grep*, although it is more convenient to use a program built for such task. A mouse click on a mail name (a mail path) opens it for reading. The same happens to any attachment.

Furthermore, attachments using weird names may be renamed using *mv*, or perhaps deleted using *rm* if they are of no interest. This program also becomes the ultimate spam processing tool. As another benefit, *Venti* coalesces storage for multiple copies of the same attachment, if present on different mails.

The same approach can be applied for sending mail. A program may collect files written by the user into an agreed–upon directory. These files are simple text files in the format used by *Acme*'s *Mail*. Only that now any editor is able to compose mail for delivery, without requiring an specific tool.

Considering that files in the file server are remotely available, mail becomes remotely available as well. For small remote terminals (such as phones) *upas/fs* may be instructed to use the new mailbox format, making IMAP available as well.

Overall, we think this approach is a good strategy for handling mail. The approach consists mostly on avoiding the need for software to handle mail. If there is no software, it will hardly run slow or out of memory.

Of course this is feasible only after having decoded mail messages, which means that indeed we require software for the task. Its job is now to unpack a Plan 9 mailbox into an already decoded set of files. This is described in the rest of the paper.

Mail box format

In Plan B, mails for users are parsed and decoded first, and then stored in a file hierarchy where these and other tools can be used to process them. The main tool is *mail2fs*, which performs this processing. A mailbox is a directory, usually under `/mail/box/$user/`, that contains one directory per month (e.g., `200603/` for mails processed on March 2006). In these directories there is one directory per message.

The convention is that message (directory) names starting with ''`a.`'' correspond to archived messages not to be usually shown to the user. Names starting with ''`s.`'' correspond to messages that seem to be spam (not usually shown either). Names starting with ''`d.`'' correspond to deleted messages not yet removed from the file system. Any other rune can be used instead of `a`, `s` or `d` as a convenience (the meaning would be up to the user). But for this optional prefix, messages use a serial message number as their directory name.

The directory for a message contains at least two files: `text` and `raw`. The `text` file has the mail headers and body already processed for reading. Its contents are similar to what *Acme*'s *Mail* would show for the message. The file `raw` has the original mail headers without any processing, including the UNIX header line. This file is kept both for debugging and also for obtaining message ids when replying to mails.

Any attachment in the mail is kept stored in a separate file (possibly with the file name indicated in the MIME header) ready to be used. That is, decoded. When the attachment is a mail, the message is stored in a subdirectory following the same conventions stated above. For mails with attachments, the `text` file contains additional text indicating the relative path names (from the mail's directory) that can be used to open the attachments. This is convenient to *plumb*(1) them while reading.

A Plan B mail box also contains two files: `seq` and `digest`. Messages are given sequence numbers while they are added to the mail box. The file `seq` contains the sequence number for the last message (or zero) and is DMEXCL to provide locking for multiple programs using the mail box. The file `digest` contains digests for mails added to the mailbox using *mail2fs* (but not for those added by hand using file tools). When a message has a digest that was already seen in the past the message is silently discarded as a duplicate.

Virtual mail folders may be created by storing text files with mail lists that contain a mail description per line starting with the path for each mail. Copying the text shown for some messages in a mail listing into another text file would ''save'' such messages into that file. The program *mlist* writes to standard output a clean listing for messages with paths found in the standard input.

Other programs in the suite (most of them scripts) provide tools as a convenience for reading mail on *Acme* and *O/live*.

Examples

Move all mails from the Plan 9 mailbox to the Plan B one, and create the later if it does not exist:

```
; mail2fs -c
```

List mails:

```
; mails
200910/1153/text     nemo          Re: FDP: Ámbito
200910/1152/text     enrique.sor   FDP: Ámbito
200910/1151/text     paurea        Re: [9fans] clarification on man 9p
200910/1150/text     esoriano      Re: cómo traducís contention?
200910/1149/text     paurea        Re: cómo traducís contention?
```

Plumb all PDF attachments from Glenda:

```
; for (f in `{mails | grep Glenda | awk '{print $1}'}^/*.pdf)
        plumb $f
```

Create a virtual folder for messages in 9fans:

```
; mails | grep 9fans > 9fans
```

Of course we may use *sort* to present the list of mails in the order desired:

```
; mails | grep '^[0-9]' | sort -t/ +0nr | sort -t/ +1r | mlist
```

Most of the times there are convenience scripts to process mail and there is no need to execute complex command lines. These scripts can be used within *Acme* or *O/live* and do their job for a panel showing a single message, for messages named in the standard input, or for messages given as argument.

For example, within *O/live*, executing "`!Mails`" at `/mail/box/$user/msgs` produces an initial list of mails. This list can be refreshed by executing "`,<Mails`" for the panel containing the mail list. To read a mail we just click (button-3) on the mail path. To remove mails from a virtual folder with cut them from the list.

To select mails according to text shown in the mail index we use the Sam command language. For example, "`,x/9fans/+-p`" produces a mail index for mails coming from *9fans*. Should we want a mail folder for just those messages, we may simply write the text shown in the panel to a text file. That file becomes a virtual mail box.

To archive a set of mails we send their index text as standard input to *Arch*. For example, "`.>Arch`" archives all mails selected in the panel. In the same way, *Spam* flags mails as spam. In both cases, *mv* is the underlying tool; it renames the directories to start with "`a.`" and "`s.`" respectively. (*Mails* lists all mail, archived or not, if given the −a flag).

Locate mails about *mail2fs*:

```
; looktags nemo msgs mail2fs
/mail/box/nemo/msgs/200801/a.9955/text
/mail/box/nemo/msgs/200801/a.9955/raw
/mail/box/nemo/msgs/200801/a.9937/text
/mail/box/nemo/msgs/200801/a.9937/raw
/mail/box/nemo/msgs/200801/a.9937/1.mai2fs.c
 ...
```

Here, *looktags* is a general purpose file search tool. Since mails are included in the file server as regular files, they are also indexed by such tool.

References

1. E. Quanstrom, Nupas: Scaling upas., *Intl. Workshop on Plan 9*, 2008.

Inferno for the Sheevaplug

Mechiel Lukkien

mechiel@xs4all.nl

ABSTRACT

The Sheevaplug is a ‘‘development kit’’ based on the *Marvell Kirkwood* system-on-chip. This chip has an ARM processor and controllers for USB 2.0, gigabit ethernet and SDIO, among others. *Inferno-kirkwood** is a port of Inferno to the Sheevaplug, and hopefully in the future to other devices based on this chip.

Introduction

Inferno-kirkwood is a port of Inferno to the Sheevaplug, a ‘‘development kit’’ based on the *Marvell Kirkwood* system-on-chip. Specifications of the Sheevaplug:

- ARM processor, 1.2 Ghz, with 16KB instruction and 16KB data caches, and 256KB L2 cache.
- 512MB DDR2 RAM, 512MB NAND flash
- serial console and JTAG interface, over USB
- single 1Gbps ethernet port
- single USB 2.0 port
- SDIO slot, supporting up to SDHC cards
- RTC, GPIO, hardware crypto support (for DES, AES, MD5, SHA1), and hardware XOR/DMA copy support.

The *kirkwood* chip has a second gigabit ethernet controller, a SATA controller, a PCI Express interface, and controllers for SPI and I2S. However, these require connectors or connector pins that are not present on the Sheevaplug. It is unfortunate that the Sheevaplug has no SATA connectors, storage must be delivered through USB. Other devices (e.g. *OpenRD*) with those connectors have become available recently.

The Sheevaplug uses just below 3 watts when idle. It comes with Linux that runs from flash, configures its network with DHCP and runs an ssh server with a default root password: getting started with the device is trivial.

The *Inferno-kirkwood* project was started after the Sheevaplug was mentioned on the Inferno mailing list. The Sheevaplug is reasonably cheap, has enough hardware on it to provide useful network services, and its hardware is documented. It looked like a nice device to get experience writing device drivers and other low-level code. Without specifications it would be highly unlikely I would be able to run Inferno on these devices. It did turn out that some documents referenced from the main specification document† were not publicly available, but that has not hampered development yet.

---

* *Inferno-kirkwood*, `http://code.google.com/p/inferno-kirkwood/`

† *Kirkwood function specification*, `http://www.marvell.com/files/products/embedded_processors/kirkwood/FS_88F6180_9x_6281_OpenSource.pdf`

The initial code to get a kernel booted, was written by me. Soon Salva Peiró got interested and started developing. Before his Sheevaplug arrived he developed remotely, connecting to a machine that had console access to a Sheevaplug. One of the first things he implemented was rebooting with `^t^tr`. Development has recently shifted to other projects, but is expected to shift back soon.

Progress

A kernel can already be booted, with a working serial console, real-time clock and ethernet controller. Some devices are partially supported, some do not have any support at all.

The UART, for serial console, was the first device to be used. We probably do not yet set all parameters for the serial console properly, instead relying on the boot loader *u-boot* to initialize them.

Ethernet has better support, but not all hardware features are supported. For example use of multiple queues for different types of network traffic each with its own packet memory, or TCP/UDP checksum offloading. Perhaps these will not be implemented in the future either though. Interrupt coalescing has already been implemented. Before the processors *dcache* can be turned on, the ethernet driver must be modified to flush the cache of the descriptors for packet memory.

The real time clock is read at start up and can be set as well. The RTC hardware also has an alarm, but we do not support it.

The NAND flash is detected but cannot yet be accessed. Once it is supported, Inferno can be booted from flash.

SD controller support is not complete either, but data can be read from SD cards. A FAT32 partition has already been mounted, but the driver has stability problems (kernel crashes). Writing to SD cards has not been tested but is not very different from reading.

A driver for the cryptography hardware has been written, but is not currently enabled. It is not clear if all hardware support will be used in the future. The DES library used in some parts of Inferno has a library interface incompatible with the hardware's interface to DES, so the changes required may be too intrusive and the gain too small. The hardware supports DMA for cryptography operations, but that has some caveats and is not the easiest way to use the crypto hardware.

The XOR controller copies memory using DMA and can optionally XOR data sources. This is aimed at RAID implementations, but perhaps plain DMA memory copies can be used by Inferno to lower CPU load.

GPIO signals have been set, but there is no generic driver that gives access to GPIO functionality yet.

Finally, we have a driver for the *efuse*: small memory that can be written once and read often.

History

Initial development went surprisingly quickly, considering how little experience with low-level programming I had. I started with Inferno on OpenBSD, but any other Unix that Inferno runs on would have worked just as well. The first goal was to create an Inferno kernel and convince the Sheevaplug to load it. The Sheevaplug comes with the *u-boot* boot loader, which supports BOOTP/DHCP and can fetch a kernel over tftp. The kernel has to be in *u-boot*'s *uimage* format, which turned out to be easy to create. Before my Sheevaplug arrived, I had a *mkuimage* program that would take an Inferno kernel and add a *uimage* header. Code from another Inferno ARM port was taken as a starting point. That gave me skeleton code that could be compiled into a kernel. I could turn that into an image that the Sheevaplug was willing to load and start executing

code from. Of course, the first kernel did nothing useful. Initially, I was not even sure which starting point addresses I had to put in the *uimage* header and kernel image, so whether *u−boot* was jumping to the right instructions. The best way to show a sign of life seemed to be to write a character to the serial console. I now usually connect my Sheevaplug through a power meter: power consumption shows me whether a broken kernel is hanging, spinning or causes too many interrupts. After some trial and error, trying a few kernel starting addresses and various UART configurations, the first character appeared on the console! That was encouraging. The existing Inferno ARM code was extremely helpful for getting to that point. After that, support for more controllers has gradually been added.

Developing

Currently, development proceeds as follows: The standard Inferno build process is used to create a kernel, in `os/kirkwood/`, which is then wrapped in a *uimage* header. The resulting image is copied to a tftp server (the `mkfile` has a target to copy it to `/n/tftp`). The Sheevaplug is rebooted, and the DHCP+tftp boot method fetches the new kernel and starts it. *U−boot* can also boot from flash, and from the SD card, and with the latest experimental version even from a USB mass storage device. Eventually we want to boot from flash, and probably also from SD card.

Future

Obviously there is still a lot to do. The current *inferno−kirkwood* code is not yet ready for normal use. This is exemplified by the lack of NAND flash support that would allow writing to and booting and running from flash. Beside the obvious need for making the code more stable, faster (enabling the *dcache*) and finishing existing drivers such as for SD cards, new drivers need to be written. Apparently the USB controller is a standard EHCI controller, so hopefully it will not be too hard to port Plan 9's USB EHCI driver. The SATA controller does not need support yet because the Sheevaplug does not have connectors for it. Other devices using the *kirkwood* chip have become available now, and they do have ESATA ports and the second gigabit ethernet port. They also have more USB ports and one device has a PCI Express-connected video card. Support could be extended to these devices, but it is not currently planned.

When enough hardware is supported, the Sheevaplug can provide network services. Inferno does not have programs for all common network services. Some of those I have started on, e.g. a DHCP server (only a BOOTP server exists in Inferno), a simple anonymous FTP server, an NFS server, etc. Plans for other network services exist too, e.g. an ssh 2 server. The goal is to replace an existing OpenBSD server with this more power−efficient Sheevaplug running Inferno, providing similar network services. Somewhat unfortunately (but unavoidably), implementing those has kept me from improving hardware support.

Ssh

Mechiel Lukkien

mechiel@xs4all.nl

ABSTRACT

*Ssh** is a client for the *secure shell 2* protocol, written in Limbo for Inferno. It also includes *sftpfs*, an *sftp* client that translates between sftp and styx/9p messages. They currently support most of the popular key exchange, authentication, encryption and digesting methods used by the *ssh* protocol, but is not yet ready for daily use. No code for a server has been written yet.

Introduction

The *ssh* project aims to bring secure shell 2 support to Inferno. It has both a secure shell client for executing a shell or other commands on an ssh server and an sftp client that gives access to files on the ssh server. No code for an ssh server or a program like Plan 9's *sshnet*(4) for using the server's TCP stack has been written. At the time of writing, the basics work, i.e. logging in to an ssh server; basic sftp works too. Most of the standard key exchange, authentication and encryption and digest methods are supported: *Diffie–Hellman* key exchange with *rsa* and *dss* host verification; password and *rsa* and *dsa* public–key authentication; *sha–1* and *md5* for digests; and *des–cbc*, *3des–cbc*, *aes* (with 128, 192 or 256 bit keys, in *cbc* or *ctr* mode), *arcfour* (with 128 or 256 bit keys) and *idea* (though untested because no one uses it). New *factotum*(4) support handles authentication for the public–key methods. The two major limitations are:

- Channel windows are not updated. Both client and server maintain a window for each communication channel on the ssh connection. The window indicates how many bytes the other side is willing to consume, thus how many bytes can be written without blocking. A large initial window is set by the current code, but the window is never updated during the connection. This works for most types of communication, but will block when all data from the initial window has been consumed.
- Session key renegotiation has not yet been implemented. During connection set up, encryption and digest keys are exchanged between the client and server (and in the process the client verifies the server's *host key*). These are used to encrypt and sign the protocol packets. The protocol specification dictates that keys be renegotiated after they have been used for some time or for a certain number of encryption operations. We currently never do that. Thus, when the server wants new keys, the connection will break.

The protocol

Some details about the protocol. The protocol is specified in *RFCs* 4250–4254. Later *RFCs* clarify, extend and/or deprecate functionality. The protocol is logically layered, with a *transport*, *user authentication* and *connection* 'layer'. These layers are more like phases of the connection. Protocol messages are not actually layered or nested, all have

---

* *Ssh*, `http://www.ueber.net/code/r/ssh`

the same packet format with one set of simple encoding rules. *Sftp* is not part of the secure shell protocol. It can be used outside of ssh too, though this is uncommon. It can be implemented as a separate program that speaks the sftp protocol over a channel by provided by an ssh connection. Sftp is only described in expired *work in progress RFC* drafts The most recent versions of those drafts should be ignored: they are not commonly implemented and only add complexities such as *ACL* schemes.

Ssh provides secure communication between two systems. The server accepts incoming connections and plays the role of the server. The connection goes through various stages: Key exchange (including host verification), user authentication and finally normal operation during which communication channels can be created and data sent and received on those channels. An ssh client creates a channel and requests a remote service, typically a login shell. The server starts this service and essentially connects the channel's communication descriptors to the shell's standard input, output and error. The client does a similar thing. The protocol allows many channels to be opened on a single ssh connection. For example for multiple shells, or a shell and an sftp connection.

The sftp service can be requested on a channel, with the same mechanism used to start a shell. The sftp service reads sftp protocol requests from the channel and writes the sftp responses to the channel. Sftp maps surprisingly directly to styx, but (as most such protocols) cannot do all of a *wstat* operation atomically, needing multiple sftp requests. *Sftpfs* does not wait for an sftp response before sending the next sftp request, so has some accounting to do (e.g. for flushes, and the two-stage *wstat*).

Additional services have been specified: X11 forwarding, authentication agent forwarding. No support for those is planned.

Future work

The two missing bits of important functionality have been explained earlier. Many more small ones exist and it is likely that large chunks of code need to be rewritten. The design might need to change, as a consequence of how it was developed: I wanted to get some useful packets exchanged with a server as soon as possible, so I dialed an ssh server (running *OpenSSH*) and saw it sent a banner. Finding how to respond to that banner was easy, ssh packets followed soon. By the time encryption was needed, generic packet parsing and packing code was usable. In the mean time I had realised I could enable debugging output (including protocol message printing and diagnostics) on the *OpenSSH* server. It would tell me if packets were malformed, unexpected, etc. So the *OpenSSH* server has been a great help during development. This approach resulted in quick initial results, and the protocol was learned along the way. It did not result in very clean code though, but that will be fixed.

Both the ssh client and sftp client need lots of polishing. At some point a terminal emulator for Inferno would be useful, to be able to use *curses* programs on unix systems.

Other missing features:

- Ssh version 1 is not supported. It is being phased out on the internet, few people still use it and every new ssh server deployment supports ssh version 2 and often refuses to speak ssh version 1 because it is less secure. Support for version 1 will probably not be implemented.
- The reasonably popular *blowfish* encryption algorithm is not supported yet. It seems there are various versions of *blowfish* in use, with different endianness for data and/or keys. Newer key exchange methods that use *SHA−2* are also not supported: Inferno does not yet have a *SHA−2* library. Both should be fixed eventually, though there is no hurry.
- The ssh protocol supports compression of the data packets with the *deflate* algorithm. *Ssh* will not support it any time soon because Inferno's *deflate* library does

not support flushing the compressed buffer on command, which is required.

- There is currently no ssh server, or sshnet-like program. Both are useful and may be implemented in the future.
- Host key fingerprints, used to verify that the host is who it claims it is, are stored in a file, `$home/lib/sshkeys`. Since this is security sensitive information, use and management of these keys should perhaps be handled by a *factotum*-like program.
- Inferno's factotum currently always prompts the user for credentials when a key was requested but none found. *Ssh* tries *rsa* and *dss* keys first, then normal passwords. In the quite common case of wanting to authenticate by password, this results in two unwanted factotum key requests.

File indexing and searching for Plan 9

Francisco J Ballesteros
Laboratorio de Sistemas
Universidad Rey Juan Carlos

ABSTRACT

In Plan9 most resources are provided as files, including regular on disk stored files. When the set of files grow, it is important to be able to quickly locate files based on their contents. This paper describes the set of tools documented in *tags*(1), which provides file indexing and content based searching for Plan 9, using a file system to provide the search interface.

1. Introduction

It is common today in most systems to be able to search files based on their content. MacOSX has the Spotlight tool, Google supplies a file indexing and searching tool for various systems including Windows and Linux, and UNIX has since long ago tools like `whereis` and *Glimpse* [2] that permit searching files by name or by content. There are many other content-based tools. See [4] for a survey or [3] for tools used for string matching.

The native version for Plan 9 of such tool was missing, although APE can be used to port others from other systems. In fact, Seft [1], another search tool, is an APE port already available at sources. We describe here our second attempt at providing content based file searching for Plan 9, after early experiments in Plan B with *adb*(1), a simple tag database.

Content based searching requires solving two problems: (1) indexing file contents and (2) searching the indexes to execute queries. How both things are done depends on the type of searches to be supported. The tools described here provide only exact word matching. Approximate text matching would require different techniques [3] and has not been considered yet.

For exact word queries, it is important to be able to extract the words of interest (probably all) from a file. The appropriate way of doing this is specific for each type of file. Therefore, the indexer has to be modular and permit the addition of specific word extractors for different file types.

Regarding searching, there is a compromise between maintaining the database in memory, leading to fast searches and to high memory consumption, and maintaining most of it on disk, which leads to slower searches but minimizes memory consumption.

Another issue is that on Plan 9, as of today, it is not uncommon for a user to have multiple terminals. At the very least, it is very common (by design) to have multiple terminals sharing a central file server and several CPU servers. This may be exploited to use (otherwise idle) machines to help in file searching, while keeping most of the terminal resources available for other uses.

It is also important for the indexer to support quick updates to the index, or changes to the file system would not be able to get incorporated quickly to the database, and users would miss most recent changes, as made to the file system. Considering that many searches refer to things just made, this is an issue.

In Plan 9 users rearrange their namespaces so that they incorporate possibly many file trees, from different servers. However, an indexing tool must keep file paths in a consistent way for all name spaces. Also, it is important to run the indexer utility close to the file server, to reduce I/O latency. As a result, the search database is best associated to a particular file tree (at a single file server), instead of being associated to a namespace. In that way the database used determines the file tree where the files reside, and paths are not ambiguous anymore (because they are not a function of the client's name space).

2. Data structures

The set of tools described in *tags*(1) builds upon two data structures:

1 A trie that maps words to file qids.

2 A hash table that maps qids to file names.

We keep the index and file names apart because Qids are more compact. Qids are used as values in the trie, instead of using strings. This permits the Trie to be more compact, which is important because the memory occupied by the database is significant. The more compact it is, the better. This can be done because the database refers to a file tree at a particular server, to keep paths meaningful as said before.

The trie is stored in a single file, e.g., `/lib/sys.trie.db`, and is fully read on main memory by tools that use it. The hash table is kept at a separate file, e.g., `/lib/sys.hash.db`, and is also entirely read onto main memory by the tool supporting it.

All words used to lookup files are kept in the trie. This means all words contained in text files, and all words extracted from other file types. As an aid, each file is considered to contain the path elements present on its name. This allows, for example, looking for `sys` and `src` to focus a search on system source files.

Each node in the trie represents a prefix (or a full word). The root node corresponds to the empty word. A trie node is described by the following structure:

```
typedef struct Trie Trie;
typedef struct Tent Tent;

struct Tent {
        Rune    r;
        Trie*   t;
};

struct Trie {
        Tent*   ents;   // ents[i].r are runes for children
        int     nents;  // ents[i].t are children
        int     aents;  // # of ents allocated
        uvlong* vals;   // values for this prefix
        int     nvals;  // # of values in use
        ulong*  svals;  // small values (fit in a long)
        int     nsvals; // # of small values in use
};
```

A node `t` maintains pointers (in `t->ents[i].t`) to child nodes that represent longer words sharing the prefix represented by `t`. Each link to a child node is labeled by `t->ents[i].r`, the rune that has to be added to the prefix represented by `t` to

obtain the prefix represented by the child.

For example, the prefix *a* representing either the word `a` or all the words starting with `a` would be represented by the child `t->ents[i].t` of the root node, provided that `t->ents[i].r` in the root node contains the rune *a*. Figure 1 depicts an example trie keeping the words (and their prefixes) *hi*, *hello* and *so*.

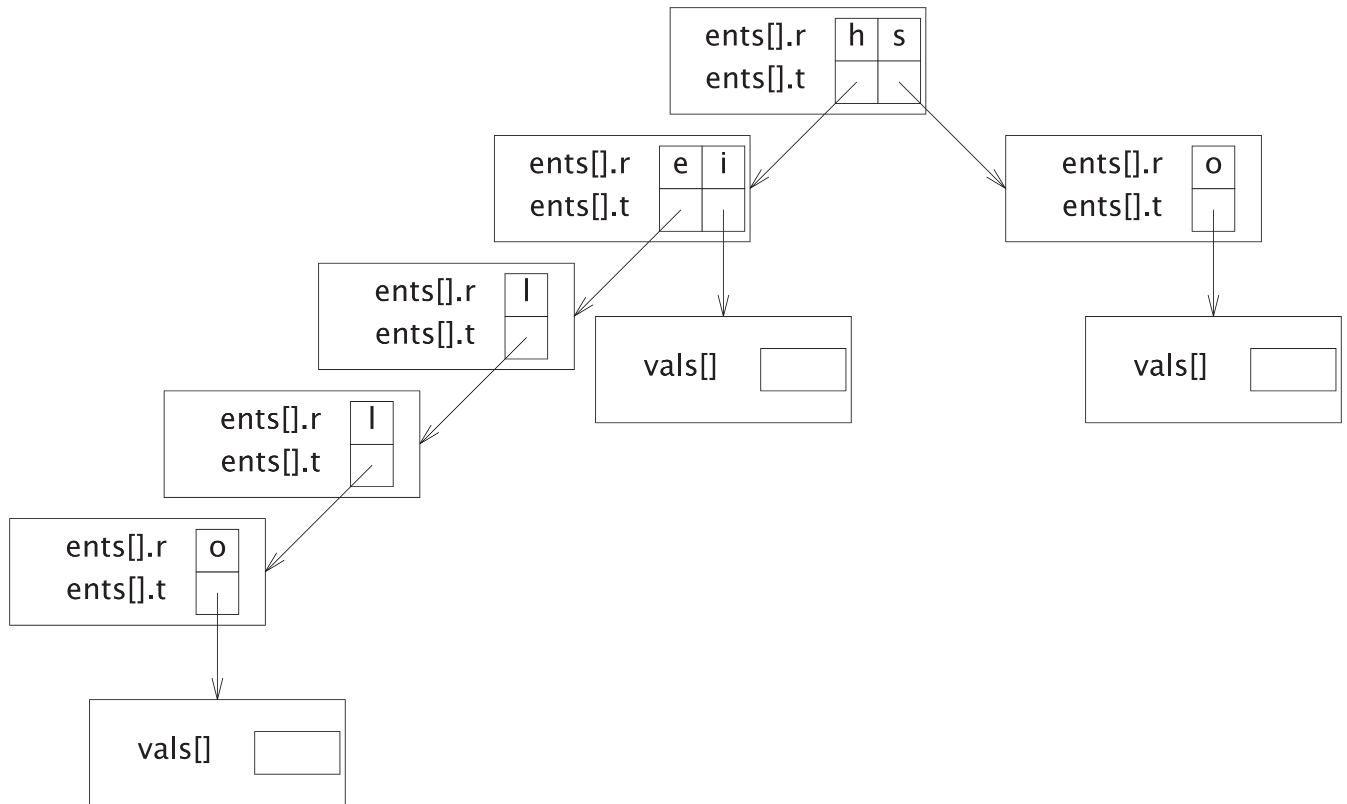


Fig. 1: *A trie with three words: hi, hello, and so.*

The trie is used to map words to Qids. Each node `t` that (besides being a prefix) represents a valid word (a key for a set of files) contains in `t->vals` an array of Qids. For example, in the figure, the two nodes holding the `o` rune would point to further trie nodes, used just to contain the Qids for files tagged with `hello` and `so`. In the same way, the entry for `i` in the left child of the root node would point to another trie node used just to contain the Qids for files tagged with `hi`. All nodes in the figure are similar, but we do not show empty arrays, for clarity.

Both `ents` and `vals` arrays are grown dynamically, as more space is needed. For `ents`, `nents` records the number of entries used and `aents` records the number of allocated entries (because in the future we might allow to delete entries in this array). For `vals` we grow the array in chunks of `Incr` nodes (a constant in the program) and there is no need to keep `avals`. The array `t->svals` is an optimization, discussed later.

Note that this implementation leads to a flat trie, whose depth is only as long as the longest word. Other implementations would use inner trees (mixed within the trie) to keep children conceptually contained at a single node in the trie.

We thought that it was best to keep the entire database in memory, to permit fast searching. That is considering that memory is cheap and that we may also be able to use the memory of a shared machine to keep the database there. Nevertheless, the database must be kept as compact as feasible while on memory to avoid consuming all the memory available. Therefore, using a single array to keep all the pointers to the children

(and all the values) seemed a sensible thing to do. It permits a compact representation.

Both arrays are kept sorted, which means that searches on them are still logarithmic. Additions to the trie are not that frequent and they do not require fast response times, compared to searches (which are interactive).

Many Qids fit in a `long` value, and the trie stores them in `svals` instead of doing it in `vals`. That way we use half the size for such Qids, at the expense of maintaining two more words at each node in the Trie, to maintain the array. We tried both with and without this optimization and the difference is about 50 Mbytes of main memory for our system database. In the future, if many Qids use the high long of their path, the two extra words may turn into a penalty. Searching time is not affected by this optimization.

The data structure mapping Qids to paths is a simple hash table. There is not much to say about it, other than showing the structure itself:

```
typedef struct Ent Ent;

struct Ent {
        uvlong  qid;
        char*   path;
        Ent*    next;   // in hash
};

Ent*    hash[Nhash];
```

Searching for lines matching the given query relies on a double search. First, the inverted index implemented by the trie reports the files relevant to the query. Second, *grep*(1) is used to search such files and show relevant lines.

One point of interest in both data structures is that neither one supports removal of entries. Removing qids from the trie would require iteration over all the nodes in the trie, which is utterly expensive. Instead, the tool searching the hash table checks that files being looked up still exist, before printing their paths. If they are removed they are simply discarded.

When a file no longer contains a tag, it may still be indexed by the tag. In any case, the tag is related to the file (because it did contain it). The search interface relies on *grep*(1) to show lines that match the query on the files retrieved from the database. If a tag is no longer in a file, no lines will be shown for such tag. This makes the problem of old tags mostly irrelevant.

From time to time, (e.g., once per month), the database may be regenerated to clean it up. That is the price for avoiding the time to remove entries while re-indexing files.

3. Tools

The software for indexing and searching files is split into different tools, as described in *mktags*(1). This is their synopsis:

mktags [**−d**] *dbpath file*...

[**DB=** *dbpath*] **looktags** [**−n**] *tag*...

tagfiles [**−d**] *triepath file*...

rdtrie *triepath* [*tag*...]

qhash [**−dv**] *hashpath* [*qid*...]

qhash [**−dv**] **−a** *hashpath* [*qid path*...]

qhash [**−dv**] **−c** *hashpath file*...

tagfs [**−abcD**] [**−s** *srv*] [**−m** *mnt*] *triepath*

The first two programs are Rc scripts providing the primary user interface. The other programs provide the actual software for indexing and searching.

Mktags creates a database named *dbpath* that maps from tags (words) to file names. Only given files are indexed (including subdirectories as well). Any word in the path name for a file, and any word contained in the file (for most files) becomes a valid search tag for the file. The resulting database is made of two files: a trie and a hash table. The name of the trie has the suffix `.trie.db` and the name of the hash has the suffix `.hash.db`. The path to the database files without their suffix is considered the name of the database.

By convention, there is a system wide data base at `/lib/sys` (that is, `/lib/sys.trie.db` and `/lib/sys.hash.db`) and a per-user data base at `$home/lib/$user` (that is, `$home/lib/$user.trie.db` and `$home/lib/$user.hash.db`).

Looktags searches the system and user databases for files that match the query specified by its arguments. By default, only file names are printed. Flag `−n` instructs *looktags* to run *grep*(1) to print some of the matching lines.

A query is made of lists of tags separated by the ‘‘`:`’’ character, each as a distinct argument. A file matches the query if it is associated to (contains) all the tags in one of the lists. For example,

```
looktags a b c : d e
```

would search for files either matching all of `a`, `b`, and `c` or matching all of `d` and `e`.

Looktags can be instructed to use a different database by defining the *DB* environment variable to contain a list of names for the databases to be used (without any file name suffixes).

Qhash maintains a file name hash table in the database. This data structure is used to translate Qids into file names.

The first invocation syntax (without using flags `−a` or `−c`) can be used to retrieve path names for the given *qids* in the command line. This is used by *looktags* to obtain paths for matching files. Under flag `−a` the program *qhash* adds the following argument pairs (each with a *qid* and *path*) to the *hash* file. Under flag `−c` *qhash* retrieves Qids and (absolute) path names for *file*(s) mentioned as arguments (recurring for directories), and adds them to the database. This is used by `mktags` to create/update the hash file in the data base.

In memory databases

Rdtrie can be used to inspect and query the Trie in the database. The Trie data structure keeps all the known tags in a trie, maintaining a list of Qids for each tag.

Without any *tag* argument in the command line, *rdtrie* reads and prints the entire Trie file, *trie*. Otherwise, *rdtrie* reads *trie* and then interprets any following arguments as a query. Qids matching the query are printed in the standard output. *Looktags* relies on this program to execute its query.

To speed up searches, the trie part of the database can be kept in memory using *tagfs*. For example, if the database is named */a/b/dbname*, *looktags* searches first for a file named `/srv/dbname.tagfs` (to reach a server holding an in-memory version of the trie part of the database), and uses it when available. Otherwise, *looktags* looks for the host identified by `$search` in the *ndb*(6) database. Should it be found, *looktags* imports its `/srv` directory to look for `/srv/dbname`.tagfs on it. This is used to share an in-memory database among several machines sharing a network. Only as a last resort would *looktags* read the database by itself to execute the query.

Tagfs can also be used to update a Trie, besides being an alternative to *rdtrie* to perform searches. The directory served by *tagfs* contains a `ctl` file that can be read to gather statistics about the Trie and can be written to modify the trie. A write of the string `sync` writes the in-memory database back to its file. A write of the form

`tag` *qid tag*...

adds *tag* to *qid* in the trie (but does not update the on-disk database).

A query can be made by creating a file, writing the query into it (being careful to separate different tags and `:` characters with white space), and then reading from the same file the list of qids that match the query. The query file is removed as soon as it is closed after having read from it.

Modularity

Tagfiles tags every file mentioned as an argument (recurring for directories) using the Trie stored in the given *trie* argument. *Mktags* relies on this program.

For each file indexed, *tagfiles* uses every word in its path name as a valid tag to search for the file. Also, *tagfiles* looks at the file name suffix and uses *file*(1) to determine the type of file and pick a particular indexing method. For text files, *tagfiles* reads entire file contents and associates each word contained in the file as a tag to search for the file. For other types of file, *tagfiles* tries to execute external programs to extract the list of tags for each file. Should the appropriate external program not exist, *tagfiles* would still try to index the file as text when appropriate.

The following programs may be executed by *tagfiles* to obtain tags for files. They are expected to write tags for the file given as an argument, one per line:

- *tagc* to tag C source.
- *taglimbo* to tag Limbo source.
- *taghtml* to tag HTML files.
- *tagman* to tag manual pages
- *tagrc* to tag Rc scripts
- *tagtroff* to tag roff source.
- *tagdoc* to tag Microsoft Office documents, including rich text format.
- *tagpdf* to tag Adobe PDF files.
- *tageps* to tag Adobe EPS files.
- *tagps* to tag PostScript files.

4. Examples of use

Create the per-user and the system database:

```
; mktags $home/lib/$user  $home /mail/box/$user/msgs
; mktags /lib/sys /cfg  /rc /sys
```

Look for files mentioning either list append or queue append, then repeat que query but using an alternate database kept at `/lib/other.trie.db` and `/lib/other.hash.db`:

```
; looktags list append : queue append
; DB=/lib/other looktags list append : queue append
```

Add (or update!) tags for files under `/usr/prof` to the personal database:

```
; tagfiles $home/lib/$user.trie.db /usr/prof
; qhash -c $home/lib/$user.hash.db /usr/prof
```

Place the system database in memory so that `looktags` can be faster, and add the tag `yoyoba` to file with qid `8345f`

```
; tagfs /lib/sys.trie.db
; echo tag 8345f yoyoba >/mnt/tags/ctl
; echo sync >/mnt/tags/ctl
```

Make the system database at `whale.lsub.org` available to other hosts: First, edit `/lib/ndb/local` to contain `search=whale.lsub.org` for the network entry. Second, at whale:

```
whale% tagfs /lib/sys.trie.db
whale% chmod a+rw /srv/sys.tagfs
```

Now from other hosts, *looktags* may use Whale's in-memory database.

5. Heuristics

The most important heuristic is the one used by *tagtext* in *tagfiles* to determine which pieces of text are words. In particular, words of less than three characters are ignored. Also, pieces of non-blank text of more than 50 characters are considered as non-text (see for example encoded attachments in mails). The remaining text is parsed to locate alphanumeric words to be used as tags.

6. Performance

We have not really made any performance measurements for the tool. In part because it is good enough to fit our needs. Nevertheless, we include some concrete measures here to give a glimpse of its behavior. The implementation contains 1876 lines of C code (not counting library functions used).

An important measure is the size for the database. This is what *ps* says for our system database and that for the author:

```
; rx whale ps | grep tagfs
elf             383    0:27   0:13   148736K Pread    tagfs
nemo            915   55:36  13:42   236808K Pread    tagfs
```

The personal database includes all mail besides indexing more than 300 Mbytes of (mostly text) files.

A search for files including *tags* and *doc* as tags takes 3.58 seconds (real time), and reports a total of 31 files in the system:

```
; time looktags tags doc
/mail/box/nemo/msgs/200102/a.997/text
/usr/nemo/doc/os/9intro/ch10.ms
/usr/nemo/doc/os/9intro/index
/sys/src/cmd/tags/tagfiles.c
 ...
0.28u 0.06s 3.58r        looktags tags doc
```

Using flag **-n** to ask for a listing of matching lines in these files (besides searching for them) takes 3.95 seconds of real time.

Adding *nemo* as another required tag makes the request take 1.66 seconds of real time.

All these measures are not implying anything regarding performance. They are not controlled experiments, but it can be seen that the set of tools behaves well enough for actual use.

References

1. O. Kretser and A. Moffat, SEFT: a search engine for text, *Software—Practice & Experience*, 2004.
2. U. Manber and S. Wu, GLIMPSE: A tool to search through entire file systems, *USENIX Winter Technical Conference*, 1994.
3. G. Navarro, A guided tour to approximate string matching, *ACM Computing Surveys 33*, 1 (2001), 31–88.
4. J. Zobel and A. Moffat, Inverted files for text search engines, *ACM Computing Surveys*, 2006.

Torrent

Mechiel Lukkien

mechiel@xs4all.nl

ABSTRACT

The *torrent** project contains a *BitTorrent* program and tools for creating *.torrent* files and *''tracking''* a torrent. It is written in Limbo, for Inferno. *Torrent/peer* connects to many peers and exchanges data blocks with them. It serves a styx/9p interface from which progress can be read and its behaviour influenced. This interface is used by *wm/torrent*, a Tk program that visualizes progress and allows stopping/starting and setting bandwidth limits.

Introduction

This report briefly introduces the BitTorrent protocol†, explains the functionality and styx interface of *torrent/peer*, the user interface provided by *wm/torrent*, implementation details, and concludes with a discussion and future work.

BitTorrent

BitTorrent is a popular peer to peer protocol for file exchange over the internet. A *.torrent* file references a *tracker*, SHA-1 hashes of *pieces* of the data that are exchanged, and file names belonging to the data. An *info hash* (SHA-1 again) can be derived from this information and is the unique identifier of the torrent file. Peers use the *info hash* to determine whether they are talking to a peer that exchanges the same data. The tracker helps peers find each other, returning lists of peers interested in the same data. The tracker is the only centralized component used during data exchange, though decentralized trackers also exist nowadays. The SHA-1 hashes in the torrent file allow verification of the received data. The file names in the torrent have no role in the protocol: multiple files are treated as a sequential stream of bytes during data exchange. All pieces (except the last) are of the same fixed size, typically between 64KB and a few MB. Smaller *blocks* of a piece, of typically 16kb, are exchanged at a time. Once all blocks for a piece have been received, the piece is verified and from then on exchanged with other peers. The *torrent* file is encoded in the *''bee''* format, a simple BitTorrent-specific format that can encode lists, dictionaries, integers and (octet) strings.

An implementation connects to the tracker periodically to fetch a list of peers, and then dials those peers (unless enough peers are already connected). It also listens for incoming connections from other peers. It keeps track of the pieces each peer has, and keeps all peers informed of all the pieces it has itself. A connection to a peer has two bits of state on both the *local* side of the connection and the *remote* side: whether each side is *interested* (i.e. wants a piece the other side has), and whether it has *choked* the

---

* *Torrent*, `http://www.ueber.net/code/r/torrent`

† *The BitTorrent Protocol Specification*, `http://www.bittorrent.org/beps/bep_0003.html`

connection (i.e. is not willing to send blocks). If the local peer is *interested* in the remote peer, and the remote peer has not *choked* the local peer, *requests* for blocks are sent to the remote which the remote peer answers with blocks.

To keep TCP working reasonably (with slow-start, back-off, etc.), only a limited number of peers are selected for sending data to, i.e. *unchoked*. The set of peers to send data to is evaluated periodically. The best performing peers are (kept) unchoked, all others are choked. Performance is measured by the peer's contributed bandwidth. A random peer is unchoked once in a while, hoping it will appreciate our bandwidth and reciprocate. This simple mechanism finds good peers to exchange data with.

The pool of connected peers is kept healthy too. In torrents with many peers (large *''swarms''*), replacing existing connections with new peers ensures good piece distribution and gives new peers a chance to get data.

These are all standard BitTorrent mechanisms. There are many details an implementation has to care of. For example, it has to defend against freeloading peers, or peers that send blocks with wrong data (whether deliberate or not).

Torrent/peer

Over the years, various extensions have been added to the protocol. Not all have been implemented. The feature that makes *peer* different from most implementations (but not *btfs*!) is its styx interface. This interface is probably not generally useful, but it does give a nice separation of the protocol details and controlling the process and showing its progress. Perhaps a *web interface* will be implemented in the future though.

The following example illustrates the current styx interface. Be warned that it will likely change.

```
% mount {torrent/peer glenda.torrent} /mnt/torrent
% cd /mnt/torrent
% ls -l
--rw-rw-rw- M 4 torrent torrent 0 Jan 01  1970 ctl
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 files
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 info
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 peerevents
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 peers
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 peersbad
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 peerstracker
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 progress
--r--r--r-- M 4 torrent torrent 0 Jan 01  1970 state
% cat info
fs 0
torrentpath glenda.torrent
infohash f52fe0191737e1c3e6e86f0081fa52d182e12a70
announce http://localhost/announce
piecelen 65536
piececount 10
length 654030
% cat files
spaceglenda300.jpg spaceglenda300.jpg 654030 0 9
% echo start >ctl
%
```

Commands can be written to the `ctl` file, e.g. to start/stop data exchange, or to set bandwidth limits. A read on `ctl` returns the values of configurable parameters. `Info` returns properties from the torrent file. `State` returns most of the progress (bandwidth rates and totals of the transfer) and e.g. the number of connected peers. `Files` lists the files described by the torrent file. Each line consists of a path (sanitized by default, so no spaces and other shell and text-selection unfriendly characters), total size

in bytes, and first and last piece that has bytes for this file. `Peers`, `peersbad` and `peerstracker` give information about the connected peers, a list of misbehaving peers, and addresses of peers that are known but not necessarily connected. `Peerevents` returns events about peers, one line per event. For example for newly connected peers, or a change of interestedness or chokedness, or when peers say they completed a piece. `Progress` returns events about progress *peer* itself is making, e.g. when a new piece is complete, or when checking the hash for a piece failed.

Wm/torrent

The styx interface exported by *peer* is used by *wm/torrent* to keep track of progress and allow setting of controls. *Torrent* shows information such as percentage of pieces completed, current upload and download rates, total number of bytes uploaded, downloaded and remaining, the number of connected peers. Two ''piece bars'' visually indicate which pieces have been downloaded and to what extend pieces are available at other peers. Another view shows information per peer, including their progress, network address, software version identifier, and upload/download rates and totals. A third view shows a list of ''faulty'' peers, those that did something wrong such as sending bogus BitTorrent messages or invalid data.

Torrent/track, torrent/create and torrent/verify

Track is a very simplistic tracker. It can be configured to serve a preset list of *info hashes*, or any *info hash* that comes along. It runs as an *scgi* program.

Create creates a *.torrent* file for a list of files that are to be exchanged. The tracker must be specified as well. *Create* logically divides the files into pieces and calculates their SHA-1 hashes for inclusion in the torrent file.

Verify calculates the SHA-1 hashes of files specified in a torrent file and compares them with the hashes in the torrent file. It prints which pieces are complete.

Implementation

The obligatory line counts:

```
  2824    9173   70618 ./appl/cmd/torrent/peer.b
   139     444    3022 ./appl/cmd/torrent/create.b
   100     256    1922 ./appl/cmd/torrent/verify.b
   719    2317   16644 ./appl/cmd/torrent/track.b
   214     583    3432 ./appl/lib/bitarray.b
  1007    3080   20351 ./appl/lib/bittorrentpeer.b
   346    1316    8852 ./appl/lib/bittorrentpeer.m
  1076    3889   25111 ./appl/lib/bittorrent.b
  1305    4267   30196 ./appl/wm/torrent.b
    39     166    1002 ./module/bitarray.m
   132     531    3443 ./module/bittorrent.m
  7901   26022  184593 total
```

Future work

The BitTorrent protocol has only ten very simple protocol messages. The file format of the torrent files is simple too, and the responses from the tracker are in the same format. Most of the work consists of managing all the connections, making sure all peers that are willing to transfer data receive requests, in a pipelined fashion. For each peer we have to keep track of the pieces they have, which of those we still want, which of those have not yet requested, etc. Preventing abuse plays an important role too. Thus, the most complicated part is all the accounting, keeping all the information in a consistent state and quickly accessible (at low cpu cost).

Decisions are made continuously: which piece to request next, which peers to unchoke. These decisions can be made with ''smart'' algorithms, e.g. based on previous actions by the peer. However, that greatly complicates the accounting and is susceptible to abuse. A simple and robust approach is to pick one of the options at random. It is cheap to execute, requires little bookkeeping and typically less prone to abuse.

There are many things that need improvement, listing them here would be too much (and too detailed). Some of the immediate or larger items on the list:

- Quality of peers should be taken into account more when requesting blocks. This provides robustness against malicious peers that send wrong data. A mechanism to divide clients by whether they have delivered a full piece, delivered blocks of a completed piece, are of unknown quality, or have mistreated us in the past has been implemented partially.
- *Torrent/peer* should handle multiple torrents at once. Currently multiple *peer*'s and *wm/torrent*'s have to be started. This is not necessarily bad. However, transferring multiple torrents in a single *peer* allows for better traffic and connection optimisation, i.e. getting more bandwidth in return for given bandwidth. For example the *n*m* best peers over all *m* torrents can be unchoked, instead of the best *n* for each torrent.
- UDP trackers, as opposed to the default HTTP over TCP trackers, might be useful, though mostly to lower the load on trackers.
- Http seeding extensions, for retrieving pieces from a web server when no peers with those pieces are available. It is not clear how commonly this is used though.
- *''Magnet URIs''* and the BitTorrent extensions protocol message could be implemented. It allows exchange of torrent files among peers, given en *info hash*. This makes BitTorrent more decentralized.
- Pieces are currently always requested in random order. Rarest-first piece selection could be implemented, to ensure better piece availability. It requires more accounting though, and is susceptible to manipulation.
- All pieces from the torrent file, thus all files specified in the torrent file are downloaded by *torrent/peer*. Support for selection a subset of the files may be implemented.

Testing is also a challenge, for example to test whether an anti-abuse measure works requires an abusing peer. Even though the protocol is simple, there are still lots of corner cases that need testing.