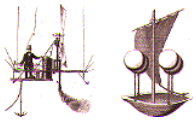
Plan 9 Internals

Lucent Technologies
Bell Labs Innovations



Tad Hunt
tad@csh.rit.edu
Computer Science House

History of Plan 9 — Origins

- Name comes from the movie "Plan 9 From Outer Space"

- Developed by

The Compuing Sciences Research Center at Bell Labs

Began in the late 1980s

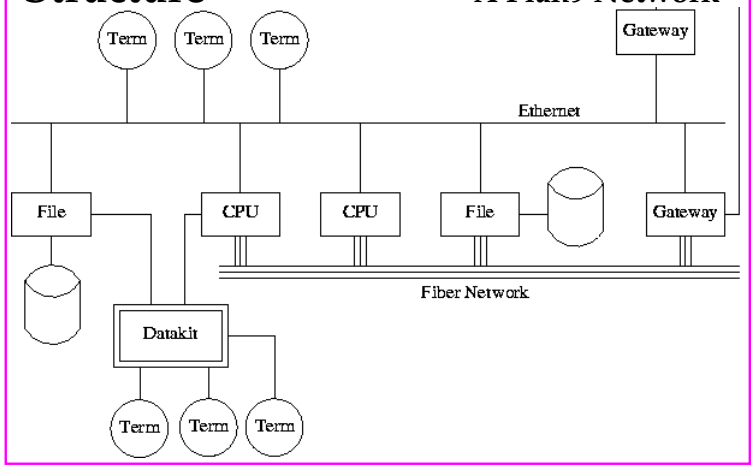
History of Plan 9 — Goals

- Fix some weaknesses with Unix

- Cleanly integrate use of network resources

- Centralized administration

Structure — A Plan9 Network



Unixisms

- Most user utilities (cat, ls, wc, etc)
- "Devices as files"
- Kernel as resource manager
- most system calls (fork(), exec(), etc...)

New concepts

- "Devices as files" taken to a whole new level
- Network more transparent to the user
- A new windowing system
- New text editing paradigm
- Servers as filesystems

Namespace

- Most resources exist in the filesystem namespace

  Exceptions:
  - memory
  - Locking primitives
  - process creation primitives
  - etc...
- Customized on a per process basis

Modifying the Local Namespace

System Calls

- bind(2)
- mount(2)
- unmount(2)

These calls modify the namespace of all processes in the namespace group

Modifying the Local Namespace

- bind(char *name, char *dest, int flag)

This system call is used to attach an existing file or directory onto another directory

Flags:

MREPL - replace the destination with the source

MBEFORE - src and dest must be dirs Unions src before dest

MAFTER - as MBEFORE but swapped

Modifying the Local Namespace

- mount(int fd, char *dir, int flag, char *aname)

Given an open file descriptor, ready to respond to 9P messages, attach it to the specified directory as specified by the flags MREPL, MBEFORE, MAFTER (as in bind(2))

Modifying the Local Namespace

- unmount (char *name, char *old)

If name is NULL, unbinds or unmounts everything bound or mounted on old.

Otherwise, name is evaluated as for bind(2), and the result is unbound or unmounted from old.

Modifying the Local Namespace

The kernel mnt(3) device

Converts read and write requests into 9P messages.

- Sends the 9P messages down the connection
- Waits for the responses.
- Sends the response to the server attached to the file descriptor

The 9P Protocol

- Can run over any reliable communications mechanism
- Is a stateful protocol
- Consists of Requests and Replies (T-messages and R-messages)

The 9P Protocol Structure

- The first byte specifies the message type (T-x or R-x, where x is the operation)
- The following bytes depend on the message type.
- All integer fields larger than a byte hold little endian unsigned integers

The 9P Protocol Operations

Tnop Rnop	Tclone Tclone	Tremove Rremove	Tcreate Rcreate
Tsession Rsession	Tclwalk Rclwalk	Tstat Rstat	Tread Rread
Tflush Rflush	Twalk Rwalk	Twstat Rwstat	Twrite Rwrite
Tattach Rattach	Tclunk Rclunk	Topen Ropen	Rerror

9P Protocol Data Types

tag[2]	uid[28]	perm[4]
chal[8]	name[28]	offset[8]
authid[28]	ticket[72]	count[2]
authdom[48]	auth[13]	pad[1]
ename[64]	qid[8]	data[count]
fid[2]	mode[1]	stat[116]

Some 9P Protocol Data Types

tag[2] - chosen by the client to identify the message. The client must insure that no two messages outstanding on the connection have the same tag.

ename[64] - text string describing the error

fid[2] - "File identifier" chosen by client to identify a "current file" on the server.

qid[8] - The servers unique id for the file.

Some interesting fileservers

- 8 ½ windowing system
- ftpfs
- exportfs
- procfs
- tcp, ip, etc...

Device Drivers

Plan 9 takes the "devices as files" Unix concept to extreme levels.

Every device is treated as a filesystem server. The only difference from a server attached using mount(2) is the fact that the device is in the kernel.

in fact, kernel devices are mounted via the mount(2) syscall.

Device Drivers

The device driver is implemented as a server, but the 9P messages aren't bundled into an RPC call, they're sent directly via a local function call.

Filenames beginning with the character "#" tell the kernel that what follows is a device driver

For example, "#c" generally means the console device.

Device Drivers

the ioctl(2) functionality from Unix is cleaned up considerably. If a device supports modification of variables controlling it, then it must provide a file in its filesystem that the user write's commands to.

For Example, /dev/eia0ctl takes standard textual commands to do things like set baud rate.

Device Drivers

Device drivers are placed in the kernel via a file with the following format:

/sys/src/9/{arch}/{arch}

```
dev
        root        /
        cons        c
        env         e
        ip          I
        scsi        S
        ...         ...
boot
        local
        il
        tcp
```


Async Process Notification

Unix equivalent: signal(2)
Plan 9: notify(void (*f)(void*,char*))

- A process registers a "note handler" by calling notify(handler).
- the handler is called with two args, the saved registers, and a pointer to a null terminated string containing the note.
- The handler finishes by exiting the program or calling noted(2)

Async Process Notification

atnotify(int (*f)(void *u,char*e),int in)

- This is the recommended way to register a note handler
- It will call all registered handlers until one returns non-zero.
- "in" is used to register (= 1) or unregister (= 0) a note handler

References

http://plan9.bell-labs.com/plan9/doc/9.html

http://plan9.bell-labs.com/vol1.html

http://plan9.bell-labs.com/plan9/distrib.html

The Plan 9 Distribution

My porting efforts