Programming Distributed Applications using Plan 9 from Bell Labs

Sape Mullender
Dave Presotto

Bell Laboratories
Murray Hill, New Jersey 07974

1. Plan 9 Overview

Sape Mullender

When Plan 9 from Bell Labs was first released in 1995, the principal developers wrote

> ''By the mid 1980's, the trend in computing was away from large centralized time-shared computers towards networks of smaller, personal machines, typically UNIX 'workstations'. People had grown weary of overloaded, bureaucratic timesharing machines and were eager to move to small, self-maintained systems, even if that meant a net loss in computing power. As microcomputers became faster, even that loss was recovered, and this style of computing remains popular today.
> ''In the rush to personal workstations, though, some of their weaknesses were overlooked. First, the operating system they run, UNIX, is itself an old timesharing system and has had trouble adapting to ideas born after it. Graphics and networking were added to UNIX well into its lifetime and remain poorly integrated and difficult to administer. More important, the early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems. Timesharing centralized the management and amortization of costs and resources; personal computing fractured, democratized, and ultimately amplified administrative problems. The choice of an old timesharing operating system to run those personal machines made it difficult to bind things together smoothly.
> ''Plan 9 began in the late 1980's as an attempt to have it both ways: to build a system that was centrally administered and cost-effective using cheap modern microcomputers as its computing elements. The idea was to build a timesharing system out of workstations, but in a novel way. Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers. For the central machines, the coming wave of shared-memory multiprocessors seemed obvious candidates. The philosophy is much like that of the Cambridge Distributed System. The early catch phrase was to build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes.
> ''The problems with UNIX were too deep to fix, but some of its ideas could be brought along. The best was its use of the file system to coordinate naming of and access to resources, even those, such as devices, not traditionally treated as files. For Plan 9, we adopted this idea by designing a network-level protocol, called 9P, to enable machines to access files on remote systems. Above this, we built a naming system that lets people and their computing agents build customized views of the resources in the network. This is where Plan 9 first began to look different: a Plan 9 user builds a private computing environment and recreates it wherever desired, rather than doing all computing on a private machine. It soon became clear that this model was richer than we had foreseen, and the ideas of per-process name spaces and file-system-like resources were extended throughout the system to processes, graphics, even the network itself.
> ''By 1989 the system had become solid enough that some of us began using it as our exclusive computing environment. This meant bringing along many of the services and applications we had used on UNIX. We used this opportunity to revisit many issues, not just kernel-resident ones, that we felt UNIX addressed badly. Plan 9 has new compilers, languages, libraries, window systems, and many new applications. Many of the old tools were dropped, while those brought along have been polished or rewritten.''

Plan 9 had its second public release in 2000. A few thousand users run Plan 9 as their primary operating system today. The third edition of Plan 9 is available for download at `plan9.bell-labs.com/plan9` and it can be ordered in a box containing a Plan 9 CD-ROM and a set of printed manuals from `www.vitanuova.com`.

A typical Plan 9 configuration at Bell Labs consists of diskless workstations, PDAs, CPU servers and file servers, interconnected by wireless and wired Ethernet. To the casual user, the system appears as a single, centralized system. Most users configure their systems to run interactive programs locally and compute-intensive ones on one of the CPU

servers.

The resources in the system are named and accessed using a hierarchical name space. The internal nodes of the name space are called *directories*, and the leaf nodes are called *files*, even though as we shall see they often aren't really files in the conventional sense. Each file or directory is served by a *file system*. Some of these are conventional file systems that store files on disk or tertiary storage. Others are only file systems in the sense that they interact with applications through a file-system interface; they do not store data on disk, however.

Consider, for example, mouse and keyboard of a workstation. The operating system makes mouse events and keystrokes available through a kernel-based file system that serves `/dev/mouse` and `/dev/cons` (the name was derived from *console*), much like Unix.

Plan 9 implements process management through another kernel-based file system, called *proc*(3), normally bound to `/proc`. The files in the directory `/proc/`*pid*, where *pid* represents a (decimal) process number, allow examination, debugging and control of a process. Writing the string *''kill''* to the file `/proc/`*pid*`/ctl`, for example, will kill the process whose process identifier is *pid*. Linux, FreeBSD, NetBSD, etc., now implement portions of this interface too; in Plan 9, however, writing a string to a process' `ctl` file is the *only* way to kill it.

The name space of a Plan 9 process is a tree, composed of the naming trees of the file systems *mounted* into its name space. This is just like in Unix. But where Unix maintains one name space per machine, Plan 9 can maintain many name spaces, one per process, if desired.

A *mount*(2) operation takes a connection to a file system (represented by a file descriptor for a pipe or a network connection) and attaches it to a point in the name space. That point then represents the root of the mounted file system. Multiple file systems may be mounted at the same point and the search order for lookups in the name space is determined by the order in which the *mount* operations were carried out and order parameters to those operations. This is a *union mount* (and the semantics are not quite the same as those in FreeBSD).

There is also a *bind* operation which takes two points in the name space (file or directory names) and makes the former visible in place of the latter. As in *mount* it is possible to bind multiple directories to one location.

The Plan 9 kernel manages the mount table and redirects operations on files to the appropriate file system. File systems receive these operations and respond to them in the form of *messages*. The 9P protocol describes these messages. We shall review them in Section 2.

Applications can be more or less oblivious of the 9P protocol. They use the system calls familiar to several generations of Unix users to read and write files: *open*, *close*, *read*, *write*, *seek*, *create*, etc. The kernel translates these calls into 9P requests and translates 9P responses back into the return values of these system calls. The mapping of system calls to 9P requests is not one-to-one, by the way.

When a process creates a new process (using the Plan 9 *rfork* system call), the child process normally inherits its parent's name space, either by copying or by sharing. If the name space is shared, *mount* and *unmount* operations of the parent are visible to the child and *vice versa*. If it's copied, this is not the case.

Name space inheritance and name space modification are used to configure the environment in which processes run. As an example, let us consider `/dev/mouse` and `/dev/cons` in the context of the window system.

Rio(1) is the window system in Plan 9. Among its tasks is demultiplexing mouse and keyboard input among the processes running in the windows on the screen. Which process gets a character or mouse event depends on the position of the mouse and the window that is current. To do this, *rio* is also a file system. Its name space contains files called `mouse` and `cons`. (There are many more files as well, but we will not discuss them here.) When a window is created and a process started to run in it, the new process inherits its name space from *rio*. But rio first *mounts* its own file system in the new process' name space so that, instead of accessing the kernel's `mouse` and `cons` files, it accesses *rio*'s instead. Thus, as far as mouse and keyboard input is concerned, processes cannot distinguish between running directly on the hardware mouse and keyboard, or running in a window. The same is done for screen output so that processes cannot tell the difference as far as output is concerned either.

As a side effect of this architecture, *rio* cannot tell either whether it runs on the raw display or in a window. *Rio*, therefore, is one of the few windows systems that allows itself to be resized. We have ported MIT's X Window system to Plan 9 also and it, of course, cannot tell either whether it runs in a window.

The 9P protocol that connects application to file systems was designed to run well over networks. As a result, it is quite simple to set up a diskless workstation with 9P connections to remote file systems. Thus, the environment presented to a user looks pretty much the same wherever the user logs in.

We have seen how Plan 9 manipulates the name space to present virtual versions of `/dev/mouse` and `/dev/cons` to applications running in a window. The Plan 9 name space obviously is not a global one in the sense of DNS or X.500 where a name refers to the same object everywhere. It is clear that truly global name spaces make a lot of sense in many contexts. It is useful that `sape@plan9.bell-labs.com` refers to Sape Mullender's mailbox anywhere in the internet. In Plan 9 the idea is that the same name *means* the same thing everywhere: `/bin/sort` is the sorting program everywhere even though the binaries are different on different machines. The `/bin` directory is typically populated at boot time by *binding* into it directories such as `/$objtype/bin`, `/rc/bin`, `$home/bin/$objtype`, and `$home/bin/rc` (*rc*(1) is the name of the Plan 9 shell).

The name space that a process sees is the result of a sequence of *mount* and *bind* operations. By repeating this sequence on another host (with a little interpretation), the same name space can be constructed there. This is used when a user starts a shell on the *cpu server* giving the user the same environment on the cpu server as on the terminal. The terminal's operating system actually exports its file systems to the user on the cpu server so that programs running on the cpu server continue to have access to devices on the terminal.

2. Plan 9 File Systems

Dave Presotto

The 9P protocol provides access (locally or remotely) to a hierarchical set of objects. The objects can be actual bits on a disk, e.g., a file system, or an abstraction provided by a kernel device or user level process.

9P has remained pretty much unchanged since the original release of Plan 9 in 1995. This is the version we will be teaching in this tutorial The next release will finally have a new version of the protocol, 9P2000. It is philosophically the same as the previous 9P. However, it has been enhanced in a number ways:

mtu negotiation (for encapsulating 9P streams inside 9P streams)

variable length fields for file name elements and user ids

larger (64 bit) object and file ID's

message restructuring to reduce the number of messages and hence round trip delays

blind encapsulation of authentication information to make 9P2000 authentication protocol agnostic

Through a trick (hack?) in the marshaling routines that convert 9P messages between C structures and their line encodings, servers talking 9P2000 can talk to both old 9P clients and newer 9P2000 clients. The marshaling routines just convert between the old and new versions.

We'll also introduce this version and some of the differences that will be important to programmers.

2.1. Client Side, setting up communications

The client side of 9P is painless. All clients use 9P indirectly via UNIX-like system calls (open, read, write, ...). Object naming and access controls are much like Unix file names, i.e., '/' separated path elements and 9 bit rwxrwxrwx permissions. Details differ but the ideas and feel are the same. Putting all process accessible objects in the same namespace removes the distinction between where things are implemented making the system much more distributable.

It is the kernel's job to convert system calls into 9P messages, write them onto the appropriate communications channel (TCP, IL, serial, etc), and convert and demultiplex the replies.

Clients add new services into their namespace in four basic ways:

1) binding local kernel devices (com ports, disk drives, etc)

```
bind '#p' /dev
```

2) mounting locally advertised services from `/srv`.

```
mount /srv/cs /net
```

3) directly dialing and mounting new servers.

4) importing a complete name space from another system.

2.1.1. Local Switchbard, #s

#s is a local switchboard for connections to services. It is normally bound onto `/srv`. A server process can leave a file descriptor there for other processes to reopen. The following example implements what would be called a named pipe in Unix:

```
int
postpipe(void)
{
     int pfd[2];
     inf fd;

     if(pipe(pfd) < 0)
          sysfatal("pipe failed: %r");
     fd = create("/srv/quux", ORDWR, 0666);
     if(fd < 0)
          sysfatal("can't create /srv/quux: %r");
     sprint(buf, "%d", pfd[1]);
     if(write(fd, buf, strlen(buf)) < 0)
          sysfatal("posting: %r");
     close(fd);
     close(pfd[1]);
     return pfd[0];
}
```

Postpipe() returns one end of the pipe. A client process opening `/srv/quux` will have a file descriptor to the other end. If the client now mounts this file descriptor somewhere in its name space (see *mount*(2)), any access at or below that spot in the name space will result in 9P messages sent through the pipe to the server.

2.1.2. Dialing a Network Service

A process could also directly dial a server using *dial*(2). This will be covered in the network section. The result is once again a file descriptor that can be mounted into the client's name space. The command *srv*(4) can be used, to call up a server, post its file descriptor into `/srv`, and optionally mount it into your name space. By posting it, we make it available for other users on the system should they also want to connect to the same resource.

2.1.3. Importing a Remote Namespace

Finally, one can import a complete namespace from another machine and use it to access resources on that machine. The client program is called *import*(4) and the server side *exportfs*(4). *Import* takes a remote system name, a path on the remote system, and an optional local path on your system. It effectively mounts the remote namespace starting at the remote path at the local path in your namespace. For example:

```
import achille.cs.bell.labs.com /net.alt
```

makes the external IP stack from our gateway system accessible from my local system. This is a handy way to implement a cheap firewall. We'll talk more about this in the networking section.

Import/*exportfs* performs its authentication up front before 9P is started. We'll expand on this in the authentication section.

Combining two of these ideas we can allow processes on two machines to talk. Before starting the processes, import the */srv* directory of each machine onto the other. That way each process can use the *postpipe*() routine shown above to create named pipes visible on both machines.

2.2. Server Side, talking 9P

How difficult a server is to write depends on how robust and responsive it needs to be. At the very least, it has to understand 9P messages, process them, and act on them. A 9P connection consists of request messages from the client followed by replies from the server. If any action can block, the server may need to be multithreaded to provide acceptable response. Since multiple users may send requests on a single connection, the server may also need to handle user authentication and access controls. Finally, since blocked requests may be aborted, a certain amount of asynchrony may be involved. A single-threaded server with no asynchrony is pretty easy to write. A long lived, multithreaded one, with a lot of asynchrony and memory allocation can take years to make solid. Start small, copy `/sys/src/cmd/ramfs.c` and work your way to something more complex.

2.2.1. Tags

9P messages are not necessarily processed in order. The server is encouraged not to let one blocked request delay another. Therefore a 16 bit tag is included in every message to match replies to their requests.

2.2.2. FID's

9P, unlike NFS, is a stateful protocol. Each message contains a 32 bit FID. The FIDs are chosen by the client and represent indices for the shared state known by both sides. Associated with each FID is an object instance and the authenticated user id performing the access. FID's are relative to a single connection so that when a connection is closed, FID state is lost.

Whenever a process mounts a connection to a server into its namespace, an *attach* request is sent to the server containing both a FID for the server's root and a user id of the attaching user plus any authenticating information. The attach FID is a handle for the root of the name space. The client may then *clone* that FID to a new FID and *walk* the cloned FID to any object in the space, subject to permissions. This new FID can be stored away to be walked again later or opened so that reads and writes can be performed on the referenced object.

For example, each process has associated with it 2 FID's; its root FID and its current directory FID. Paths starting with '/' are resolved by cloning the root FID and walking each element. Resolving relative paths starts with cloning the current directory FID. A call walks a path and the resulting FID is saved as the current directory FID.

In general, it is illegal to have multiple conflicting operations outstanding on a single FID. To prevent this, *walk*s are disallowed following an *open*. In addition, the server is free to impose added restrictions by serializing or reordering concurrent read and write requests.

2.2.3. Server structure:

The following can be taken as a template for a server program:

```
#include <u.h>
#include <libc.h>
#include <fcall.h>

typedef struct Fsrpc Fsrpc;
struct Fsrpc
{
     ...
     Fcallwork;      /* Plan 9 incoming Fcall */
     Fcallreply;          /* Plan 9 reply Fcall */
     uchar*buf;      /* Data buffer */
};

void (*fcalls[])(Fsrpc*) =
{
     [Tnop]          Xnop,
     [Tsession] Xsession,
     [Tflush]   Xflush,
     [Tattach]  Xattach,
     [Tclone]   Xclone,
     [Twalk]         Xwalk,
     [Topen]         slave,
     [Tcreate]  Xcreate,
     [Tclunk]   Xclunk,
     [Tread]         slave,
     [Twrite]   slave,
     [Tremove]  Xremove,
     [Tstat]         Xstat,
     [Twstat]   Xwstat,
     [Tclwalk]  Xclwalk,
};
```

The array *fcalls* has one function per message type. It is up to the function to look up the state associated with any included FIDs, perform the request, and reply.

```
void
main(void)
{
     char buf[16*1024];
     Fsrpc *r;

     fmtinstall('F', fcallconv);
     ...
     for(;;) {
          r = getsbuf();
          if(r == 0)
               fatal("Out of service buffers");

          do
               n = read9p(netfd, r->buf, sizeof(r->buf));
          while(n == 0);

          if(n < 0)
               fatal("server read");

          if(convM2S(r->buf, &r->work, n) == 0)
               fatal("format error");

          DEBUG(DFD, "%F0, &r->work);
          (fcalls[r->work.type])(r);
     }
}
```

The main loop processes one message at a time. It leaves replying up to the per message function. That way, if the function might block it can launch a thread to handle the blocking request and return without replying. The new thread is responsible for performing the reply.

The standard reply routine is shown below:

```
void
reply(Fsrpc *r, char *err)
{
	char data[MAXFDATA+MAXMSG];
	int n;

	r->reply.tag = r->work.tag;
	r->reply.fid = r->work.fid;
	if(err) {
		r->reply.type = Rerror;
		strncpy(r->reply.ename, err, ERRLEN);
	}
	else
		r->reply.type = r->work.type + 1;

	DEBUG(DFD, " %F0, &r->reply);

	n = convS2M(&r->reply, data);
	if(write(1, data, n)!=n)
		sysfatal("mount write");
	free(r);
}
```

Any reply may return an arbitrary-length error string instead of the expected reply.

2.2.4. Flush, aborting blocked requests

The big fly in the ointment is the *flush* request. It contains two tag's, it's own and the target tag. The intent is to terminate some previous request and represents the only asynchrony in the protocol. The function processing the *flush* must find the thread handling the previous request, cause it to terminate, and then reply to the *flush*. This is often difficult since it may requires sending a note (see and handling it correctly in the noted thread. To avoid a note at a bad time, there usually needs to be some synchronization mechanisms (*rendezvous*(2), *lock*(2), or *qlock*(2)) between the different threads. Be afraid, be very afraid: This is the source of most server errors.

2.2.5. The 9P library, for simpler servers

Russ Cox at Harvard has provided a library for writing 9P server, *9p*(2). This library provides scaffolding for the server. With it, the programmer need supply only the message processing routines and the library does the rest, posting a file in `/srv` and providing the main loop of the server. It's *respond* routine is much like the *reply* routine shown above. The 9P library even provides routines for creating a file hierarchy and will automatically walk it, stat directories, and perform access checks. When the server is straightforward, many programmers find libp9 very useful.

2.2.6. Marshaling

Section 5 of the Plan 9 manual (included with your notes) describes the 9P messages. The unmarshaled version of a 9P message is the *Fcall* structure. It contains unions for the contents of the different message types. A server's message functions take an *Fcall* structure as a request and produce another as a reply. *ConvM2S* and *convS2M* respectively unmarshal and marshal *Fcall* structures. *ConvM2D* and *convD2M* do the same for *Dir* structures, the C structure representing a directory entry.

2.2.7. Authentication

9P allows each user to be separately authenticated as he mounts the connection into his name space. However, most servers (i.e. all except the actual file servers) authenticate once up front and perform all accesses as a single user. See *auth*(2) for the client and *srvauth*(2) for the server. If *srvauth* returns 0, then the connection has been successfully authenticated and the server process is now running as the authenticated user. The process is still in the old name space so that it has access to the same files it did before authenticating until after a subsequent *newns*(2) call. A variant of these routines, *authnonce*(2) and *srvauthnonce*(2) also return a 64 bit secret than can be used to create encryption keys for the conversation. For example:

```
static int
p9auth(int fd)
{
	uchar key[16], digest[SHA1dlen];
	char fromclientsecret[21], fromserversecret[21];
	int i;

	if(authnonce(fd, key+4) < 0) return -1;

	if(ealgs == nil) return fd;

	/* exchange random numbers */
	srand(truerand());
	for(i = 0; i < 4; i++)
		key[i] = rand();
	if(write(fd, key, 4) != 4 || readn(fd, key+12, 4) != 4)
		return -1;

	/* scramble into two secrets */
	sha1(key, sizeof(key), digest, nil);
	mksecret(fromclientsecret, digest);
	mksecret(fromserversecret, digest+10);

	fflag = 0;

	/* set up encryption */
	return pushssl(fd, ealgs, fromclientsecret,
		   fromserversecret, nil);
}
```

This code authenticates to a server (as the current user) and sets up encryption on the connection. *Mksecret* just sprints the key into a form acceptable to our ssl link encryption device. The corresponding server code is:

```
static int
srvp9auth(int fd, char *user)
{
	uchar key[16], digest[SHA1dlen];
	char fromclientsecret[21], fromserversecret[21];
	int i;

	if(srvauthnonce(fd, user, key+4) < 0) return -1;
	fchown(fd, user);

	if(ealgs == nil) return fd;

	/* exchange random numbers */
	srand(truerand());
	for(i = 0; i < 4; i++)
		key[i+12] = rand();
	if(readn(fd, key, 4) != 4 || write(fd, key+12, 4) != 4)
		return -1;

	/* scramble into two secrets */
	sha1(key, sizeof(key), digest, nil);
	mksecret(fromclientsecret, digest);
	mksecret(fromserversecret, digest+10);

	fflag = 0;

	/* set up encryption */
	return pushssl(fd, ealgs, fromserversecret,
		fromclientsecret, nil);
}
```

Here *fchown* is a routine that changes the ownership of the network directory (described later) to the user id just authenticated. See *pushssl*(2) and *ssl*(3) for information on setting up encryption on a connection.

3. Plan 9 Networking

Dave Presotto

Plan 9 has no sockets. Instead, multiplexed devices and protocol connections (TCP, UDP, etc) are all in the namespace and look like file systems. They're accessed and controlled by reading and writing files. Hence they can be exported or imported.

3.1. Multiplexed Devices

The simplest example of a multiplexed device is an ethernet:

```
% cd /net/ether0
% ls -l
d-r-xr-xr-x l 0 presotto presotto 0 Apr 11 13:00 0
d-r-xr-xr-x l 0 presotto presotto 0 Apr 11 13:00 1
--rw-rw-rw- l 0 presotto presotto 0 Apr 11 13:00 addr
--rw-rw-rw- l 0 presotto presotto 0 Apr 11 13:00 clone
% cat addr; echo
00104b9b8172
```

At the top level there are two files and three directories. Each directory represents a conversation, in this case one per ethernet type in use.

Inside each conversation directory are a number of files representing different aspects of the conversation; control, data, statistics, packet type:

```
% cd 0
% ls -l
--rw-rw-rw- l 0 presotto presotto 0 Apr 11 13:00 ctl
--rw-rw-rw- l 0 presotto presotto 0 Apr 11 13:00 data
--r--r--r-- l 0 presotto presotto 0 Apr 11 13:00 ifstats
--r--r--r-- l 0 presotto presotto 0 Apr 11 13:00 stats
--r--r--r-- l 0 presotto presotto 0 Apr 11 13:00 type
% cat ctl
          0
% cat stats
in: 139707
out: 118657
crc errs: 0
overflows: 0
soft overflows: 0
framing errs: 0
buffer errs: 0
output errs: 0
prom: 0
addr: 00104b9b8172
%
```

Reading the control file tells you which conversation directory you're in. Writing it controls the conversation:

```
% echo connect 32 > ctl
% cat type
         32
%
```

A free conversation can be found (or created) by opening the `clone` file, which returns the fd of the `ctl` file of the new conversation:

```
% cat clone;echo
          2
ether0@pt109% ls -l
d-r-xr-xr-x l 0 presotto presotto 0 Apr 11 13:00 0
d-r-xr-xr-x l 0 presotto presotto 0 Apr 11 13:00 1
d-r-xr-xr-x l 0 presotto presotto 0 Apr 11 13:00 2
--rw-rw-rw- l 0 presotto presotto 0 Apr 11 13:00 addr
--rw-rw-rw- l 0 presotto presotto 0 Apr 11 13:00 clone
%
```

Finally, reading and writing the `data` file results in sending and receiving data.

3.2. Protocol Directories

Protocols, like TCP are a bit more complex than multiplexed devices but not much:

```
% cd /net/tcp
% ls -l
d-r-xr-xr-x I 0 presotto presotto 0 Apr 11 13:00 0
d-r-xr-xr-x I 0 network  presotto 0 Apr 11 13:00 1
--rw-rw-rw- I 0 network  presotto 0 Apr 11 13:00 clone
--r--r--r-- I 0 network  presotto 0 Apr 11 13:00 stats
```

The `stats` file for each protocol type is our equivalent of a MIB, just easier to access.

```
% cat stats
MaxConn: 512
ActiveOpens: 16
PassiveOpens: 0
EstabResets: 0
CurrEstab: 1
InSegs: 5779
OutSegs: 5514
RetransSegs: 28
RetransTimeouts: 3
InErrs: 0
OutRsts: 2
CsumErrs: 0
HlenErrs: 0
LenErrs: 0
OutOfOrder: 0
%
```

The `clone` file acts the same as the one for the ethernet, i.e., opening it finds a free conversation. A side effect of opening an unused conversation is that ownership of the the conversations files are is changed to your user id. You can then *chmod*(2) them as you see fit.

The main difference between protocols and network devices is the ability of servers to receive new calls. This is performed using the `listen` file.

```
% cd 0
% ls -l
--rw-rw---- I 0 presotto presotto 0 Apr 11 13:00 ctl
--rw-rw---- I 0 presotto presotto 0 Apr 11 13:00 data
--rw-rw---- I 0 presotto presotto 0 Apr 11 13:00 err
--rw-rw---- I 0 presotto presotto 0 Apr 11 13:00 listen
--r--r--r-- I 0 presotto presotto 0 Apr 11 13:00 local
--r--r--r-- I 0 presotto presotto 0 Apr 11 13:00 remote
--r--r--r-- I 0 presotto presotto 0 Apr 11 13:00 status
%
```

After announcing that this connection is listening on a particular port, we then listen for a call by opening the *listen* file. This file is very similar to the clone file at the top level directory. When it returns, it will return with the open fd of the *ctl* file of the incoming call.

```
% echo announce 25 > ctl
% cat listen
          3
% echo go away don't bother me > ../3/data
```

3.3. Network Addressing

A uniform interface to protocols and devices is not sufficient to support the transparency we require. Since each network uses a different addressing scheme, the ASCII strings written to a control file have no common format. As a result, every tool must know the specifics of the networks it is capable of addressing. Moreover, since each machine supplies a subset of the available networks, each user must be aware of the networks supported by every terminal and server machine. This is obviously unacceptable.

Several possible solutions were considered and rejected; one deserves more discussion. We could have used a user-level file server to represent the network name space as a Plan 9 file tree. This global naming scheme has been implemented in other distributed systems. The file hierarchy provides paths to directories representing network domains. Each directory contains files representing the names of the machines in that domain; an example might be the path `/net/name/usa/edu/mit/ai`. Each machine file contains information like the IP address of the machine. We rejected this representation for several reasons. First, it is hard to devise a hierarchy encompassing all representations of the various network addressing schemes in a uniform manner. Datakit and Ethernet address strings have nothing in common. Second, the address of a machine is often only a small part of the information required to connect to a service on the machine. For example, the IP protocols require symbolic service names to be mapped into numeric port numbers, some of which are privileged and hence special. Information of this sort is hard to represent in terms of file operations. Finally, the size and number of the networks being represented burdens users with an unacceptably large amount of information about the organization of the network and its connectivity. In this case the Plan 9 representation of a resource as a file is not appropriate.

If tools are to be network independent, a third-party server must resolve network names. A server on each machine, with local knowledge, can select the best network for any particular destination machine or service. Since the network devices present a common interface, the only operation which differs between networks is name resolution. A symbolic name must be translated to the path of the clone file of a protocol device and an ASCII address string to write to the

`ctl` file. A connection server (CS) provides this service.

The format we chose for addresses is a string composed of three '!' separated elements. An example would be the SMTP server at Stanford's CS department:

```
/net/tcp!cs.stanford.edu!smtp
```

The first element is a path in our namespace indicating the protocol directory. The second is a machine name interpreted relative to that protocol type. The final is the service on that machine. As we show later, it's up to CS to convert that into an IP address, port number, etc. No other program has to know such specifics. They can just deal with it as a string.

3.4. Network Database

On most systems several files such as `/etc/hosts`, `/etc/networks`, `/etc/services`, `/etc/hosts.equiv`, `/etc/bootptab`, and `/etc/named.d` hold network information. Much time and effort is spent administering these files and keeping them mutually consistent. Tools attempt to automatically derive one or more of the files from information in other files but maintenance continues to be difficult and error prone.

Since we were writing an entirely new system, we were free to try a simpler approach. One database on a shared server contains all the information needed for network administration. Two ASCII files comprise the main database: `/lib/ndb/local` contains locally administered information and `/lib/ndb/global` contains information imported from elsewhere. The files contain sets of attribute/value pairs of the form *attr=value*, where *attr* and *value* are alphanumeric strings. Systems are described by multi-line entries; a header line at the left margin begins each entry followed by zero or more indented attribute/value pairs specifying names, addresses, properties, etc. For example, the entry for our CPU server specifies a domain name, an IP address, an Ethernet address, a Datakit address, a boot file, and supported protocols.

```
sys = helix
      dom=helix.research.bell-labs.com
      bootf=/mips/9power
      ip=135.104.9.31 ether=0800690222f0
      dk=nj/astro/helix
      proto=il flavor=9cpu
```

If several systems share entries such as network mask and gateway, we specify that information with the network or subnetwork instead of the system. The following entries define a Class B IP network and a few subnets derived from it. The entry for the network specifies the IP mask, file system, and authentication server for all systems on the network. Each subnetwork specifies its default IP gateway.

```
ipnet=mh-astro-net ip=135.104.0.0 ipmask=255.255.255.0
      fs=bootes.research.bell-labs.com
      auth=1127auth
ipnet=unix-room ip=135.104.117.0
      ipgw=135.104.117.1
ipnet=third-floor ip=135.104.51.0
      ipgw=135.104.51.1
ipnet=fourth-floor ip=135.104.52.0
      ipgw=135.104.52.1
```

Database entries also define the mapping of service names to port numbers for TCP, UDP, and IL.

```
tcp=echo        port=7
tcp=discard     port=9
tcp=systat port=11
tcp=daytime     port=13
```

Many programs, such as the connection server and database server, read the database directly so consistency problems are rare. However the database files can become large. To speed searches, we build hash table files for each attribute we expect to search often. The hash file entries point to entries in the master files. Every hash file contains the modification time of its master file so we can avoid using an out-of-date hash table. Searches for attributes that aren't hashed or whose hash table is out-of-date still work, they just take longer.

3.5. Connection Server

On each system a user level connection server process, CS, translates symbolic names to addresses. CS uses information about available networks, the network database, and other servers (such as DNS) to translate names. CS is a file server serving a single file, `/net/cs`. A client writes a symbolic name to `/net/cs` then reads one line for each matching destination reachable from this system. The lines are of the form *filename message*, where *filename* is the path of the clone file to open for a new connection and *message* is the string to write to it to make the connection. The following example illustrates this. `Ndb/csquery` is a program that prompts for strings to write to `/net/cs` and prints the replies.

```
% ndb/csquery
> net!helix!9fs
/net/il/clone 135.104.9.31!17008
/net/dk/clone nj/astro/helix!9fs
```

CS provides meta-name translation to perform complicated searches. The special network name `net` selects any network in common between source and destination supporting the specified service. A host name of the form `$`*attr* is the name of an attribute in the network database. The database search returns the value of the matching attribute/value pair most closely associated with the source host. Most closely associated is defined on a per network basis. For example, the symbolic name `tcp!$auth!rexauth` causes CS to search for the `auth` attribute in the database entry for the source system, then its subnetwork (if there is one) and then its network.

```
% ndb/csquery
> net!$auth!rexauth
/net/il/clone 135.104.9.34!17021
/net/dk/clone nj/astro/p9auth!rexauth
/net/il/clone 135.104.9.6!17021
/net/dk/clone nj/astro/musca!rexauth
```

Normally CS derives naming information from its database files. For domain names however, CS first consults another user level process, the domain name server (DNS). If no DNS is reachable, CS relies on its own tables.

Like CS, the domain name server is a user level process providing one file, `/net/dns`. A client writes a request of the form *domain-name type*, where *type* is a domain name service resource record type. DNS performs a recursive query through the Internet domain name system producing one line per resource record found. The client reads `/net/dns` to retrieve the records. Like other domain name servers, DNS caches information learned from the network. DNS is implemented as a multi-process shared memory application with separate processes listening for network and local requests.

3.6. Library routines

The sections on multiplexed devices and protocol directories described the details of making and receiving connections across a network. The dance is straightforward but tedious. Library routines are provided to relieve the programmer of the details.

3.6.1. Connecting

The `dial` library call establishes a connection to a remote destination. It returns an open file descriptor for the `data` file in the connection directory.

```
int  dial(char *dest, char *local, char *dir, int *cfdp)
```

`dest` is the symbolic name/address of the destination.
`local` is the local address. Since most networks do not support this, it is usually zero.
`dir` is a pointer to a buffer to hold the path name of the protocol directory representing this connection. `Dial` fills this buffer if the pointer is non-zero.
`cfdp` is a pointer to a file descriptor for the `ctl` file of the connection. If the pointer is non-zero, `dial` opens the control file and tucks the file descriptor here.

Most programs call `dial` with a destination name and all other arguments zero. `Dial` uses CS to translate the symbolic name to all possible destination addresses and attempts to connect to each in turn until one works. Specifying the special name `net` in the network portion of the destination allows CS to pick a network/protocol in common with the destination for which the requested service is valid. For example, assume the system `olive.cs.bell-labs.com` supports the `cpu` service using both the IL and TCP protocols on a few different interfaces 135.104.9.15 and 135.104.9.17. The call

```
fd = dial("net!olive.cs.bell-labs.com!cpu", 0, 0, 0, 0);
```

tries in succession to connect to
il!135.104.9.15!17013
tcp!135.104.9.15!17013
il!135.104.9.17!17013
tcp!135.104.9.17!17013

`Dial` also accepts addresses instead of symbolic names. For example, the destinations `tcp!135.104.117.5!513` and `tcp!research.bell-labs.com!login` are equivalent references to the same machine.

3.6.2. Listening

A program uses four routines to listen for incoming connections. It first `announce()`s its intention to receive connections, then `listen()`s for calls and finally `accept()`s or `reject()`s them. `Announce` returns an open file descriptor for the `ctl` file of a connection and fills `dir` with the path of the protocol directory for the announcement.

```
int  announce(char *addr, char *dir)
```

`Addr` is the symbolic name/address announced; if it does not contain a service, the announcement is for all services not explicitly announced. Thus, one can easily write the equivalent of the `inetd` program without having to announce each separate service. An announcement remains in force until the control file is closed.
`Listen` returns an open file descriptor for the `ctl` file and fills `ldir` with the path of the protocol directory for the received connection. It is passed `dir` from the announcement.

```
int  listen(char *dir, char *ldir)
```

`Accept` and `reject` are called with the control file descriptor and `ldir` returned by `listen.` Some networks such as Datakit accept a reason for a rejection; networks such as IP ignore the third argument.

```
int  accept(int ctl, char *ldir)
int  reject(int ctl, char *ldir, char *reason)
```

The following code implements a typical TCP listener. It announces itself, listens for connections, and forks a new process for each. The new process echoes data on the connection until the remote end closes it. The "*" in the symbolic name means the announcement is valid for any addresses bound to the machine the program is run on.

```
int
echo_server(void)
{
     int dfd, lcfd;
     char adir[40], ldir[40];
     int n;
     char buf[256];

     afd = announce("tcp!*!echo", adir);
     if(afd < 0)
          return -1;

     for(;;){
          /* listen for a call */
          lcfd = listen(adir, ldir);
          if(lcfd < 0)
               return -1;

          /* fork a process to echo */
          switch(fork()){
          case 0:
               /* accept the call and open the data file */
               dfd = accept(lcfd, ldir);
               if(dfd < 0)
                    return -1;

               /* echo until EOF */
               while((n = read(dfd, buf, sizeof(buf))) > 0)
                    write(dfd, buf, n);
               exits(0);
          case -1:
               perror("forking");
          default:
               close(lcfd);
               break;
          }

     }
}
```

3.7. The IL Protocol

When we first built Plan 9, none of the standard IP protocols was suitable for transmission of 9P messages over an Ethernet or the Internet. TCP had a high overhead and did not preserve delimiters. UDP, while cheap, does not provide reliable sequenced delivery. Early versions of the system used a custom protocol that was efficient but unsatisfactory for internetwork transmission. When we implemented IP, TCP, and UDP we looked around for a suitable replacement with the following properties:

Reliable datagram service with sequenced delivery

Runs over IP
Low complexity, high performance
Adaptive timeouts

None met our needs so a new protocol was designed. IL is a lightweight protocol designed to be encapsulated by IP. It is a connection-based protocol providing reliable transmission of sequenced messages between machines. No provision is made for flow control since the protocol is designed to transport RPC messages between client and server. A small outstanding message window prevents too many incoming messages from being buffered; messages outside the window are discarded and must be retransmitted. Connection setup uses a two way handshake to generate initial sequence numbers at each end of the connection; subsequent data messages increment the sequence numbers allowing the receiver to resequence out of order messages. In contrast to other protocols, IL does not do blind retransmission. If a message is lost and a timeout occurs, a query message is sent. The query message is a small control message containing the current sequence numbers as seen by the sender. The receiver responds to a query by retransmitting missing messages. This allows the protocol to behave well in congested networks, where blind retransmission would cause further congestion. Like TCP, IL has adaptive timeouts. A round-trip timer is used to calculate acknowledge and retransmission times in terms of the network speed. This allows the protocol to perform well on both the Internet and on local Ethernets.

Since then machines have gotten faster and the performance of TCP became more than good enough. Also, TCP's congestion avoidance make it better across a congested Internet than IL. Therefore, we tend to offer most services on both and use IL locally and TCP externally. We may eventually decide that the dichotomy isn't worth the effort.

4. Programming with Threads

Sape Mullender

Back in the good old days of centralized systems and Teletypes programs were, by and large, deterministic. The program dictated what input would be processed next and when output would occur. Deterministic programs are still common today, of course, but distributed computing and programs with graphical user interfaces have made programs with indeterministic input and output processes very common too.

Editors, browsers, file servers, window systems, mail servers are examples of 'deterministic programs running in an indeterministic environment'. Writing and debugging them is non trivial. In effect, there have been two major schools of structuring such programs.

The first is the 'event-loop' approach: Central to the program is an event loop structured more or less like this:

```
for (;;) {
     event = getevent();
     switch(state + Nstates*event->type) {
          ...
     }
}
```

There are two big problems with this approach. One is that it is not possible to use blocking system calls; the other is that subroutines can't do I/O the main event loop is the only place where the program can block. Unix provides non-blocking versions of its system calls and *select* for blocking on a list a possible events. Programming in this fashion destroys program sructure and makes for very hard debugging.

The other major approach is to use *threads*. Threads are light-weight processes sharing a single address space. Posix threads, which are used on a variety of Unix systems are probably the best known example. They are implemented as a library in user space and use non-blocking system calls and *select* internally, while offering blocking system calls to the application. There are other, kernel based (e.g., Amoeba) or kernel-supported (e.g., Scheduler Activations) implementations of threads also.

Threads allow proper structuring of distributed and non-deterministic applications, but most thread-based applications have other problems. Unless the application has some control (or at least knowledge) of the way threads are scheduled, the application writer has to be very careful to avoid race conditions while accessing and updating shared data structures. Most shared data will need to be locked before access and unlocked after. This avoids the race conditions, but it may introduce deadlock.

Locking is very hard to get right. Programmers forget to set or clear locks, they set the wrong ones for the data in question, they don't realize locks are needed, or they try to set one they already hold. It would be nice to have threads without the hassle of needing locks. It would also be nice to have threads without the need for special non-blocking versions of all system calls and without having to write a library that provides threaded versions of all the system calls.

In Plan 9, we use such a thread package. It consists of fewer than 2000 lines of C and it is used extensively in all the major nondeterministic applications: *rio*, the window system, *acme*, the shell/editor. *sam*, the editor, *juke* and *music*, the pop and classical jukebox players, *ntpfs*, *wikifs*, *authfs* and other file systems, and the list goes on.

There are three major ingredients to the package: *channels*, *threads*, and *procs*. Channels provide synchronous or buffered communication between threads. Procs are Plan 9 processes sharing their address space. Apart from address-space sharing, they are ordinary Plan 9 processes and subject to normal, pre-emptive scheduling. Procs can contain one

or more threads. The threads in a proc are *co-routines*: at most one of the threads in a proc is runnable and this thread has to *yield* the processor to another thread. Inside a proc, therefore, the application has control over when a thread gives up the processor to another thread in the same proc. Threads have no explicit control over the scheduling of threads in other procs.

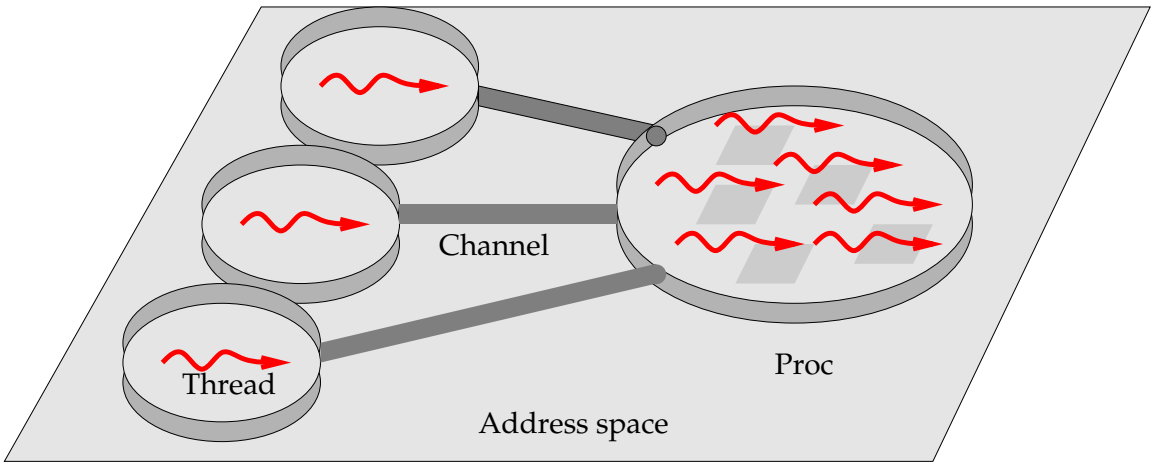

Figure 1. Threaded application structure

Plan 9 has no non-blocking versions of system calls; a *read* call on a terminal, for instance, blocks until input is provided on the terminal. While it blocks, it is still the active thread in the proc, so all other threads in that proc are blocked too. As a result, threaded applications in Plan 9 are typically structured as a single proc that may have many threads and that does all the work of the application, plus a number of procs that are all single-threaded that do the I/O. These threads communicate over the channels mentioned earlier. The structure is illustrated in Figure 1.

Channels provide a communication mechanism between threads. The basic operations on a channel are *send* and *receive*. A channel transmits elements of a fixed size which is specified when the channel is created. Since the threads using channels are in shared memory, it is common to use a channel of pointers to the actual data. This saves data copying and it allows communication of variable-size data structures. Channels can be buffered or non-buffered. In the non-buffered case, a rendezvous between sender and receiver takes place: the sender waits until the receiver receives and vice versa. They can be used between threads in the same proc or threads in different procs.

The *alt* operation groups multiple *send* and *recv* operations, allowing reception from, an transmission to, one of a group of channels. If none of the operations in the *alt* can proceed (when buffers are empty/full and unbuffered *send/recv* operations have to wait for the *rendezvous*) it blocks until an operation can proceed. If multiple operations in the *alt* can 'fire', one of them is chosen at random (to avoid starvation). *Alt* returns the index of the operation that fired.

For more details, we recommend you examine the man pages.

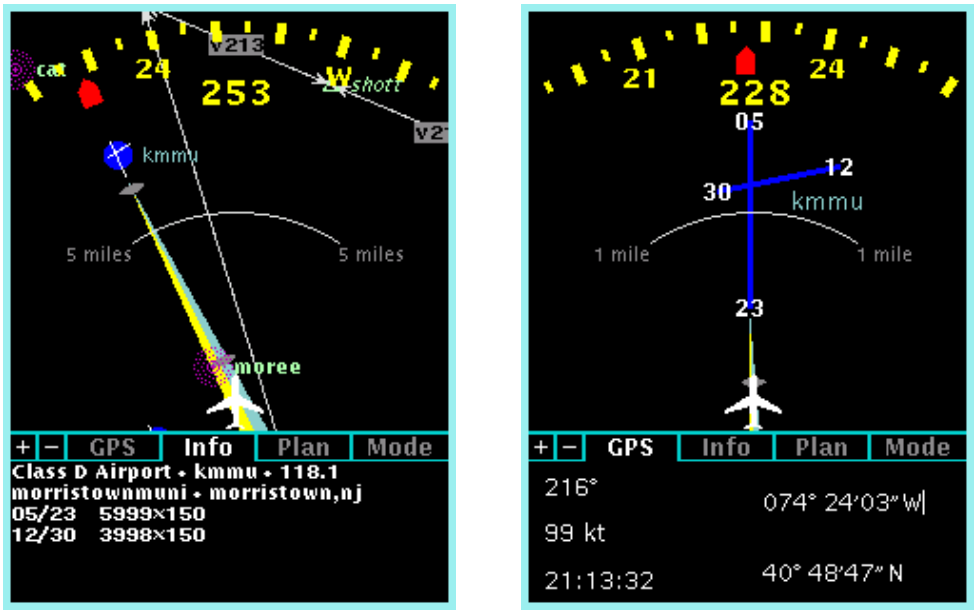

Figure 2. Plan 9 Flight information display

Let us examine a Plan 9 application to see how threading is used. The Plan 9 EFIS (Electronic Flight Information System) uses an aviation database and a GPS to display useful information about VFR or IFR flights in progress. Figure 2 shows a screen shot. The application consists of 5 *procs*, a single central proc, a *GPS* proc, a *mouse* proc, a *keyboard* proc and a *database* proc. The last of these communicates with the aviation database which is maintained by a separate application. Only the central proc is multithreaded. The mouse proc looks like this (and the keyboard and gps procs looks very similar):

```
void
mouseproc(void *arg)
{
	int n;
	char buf[1+5*12];
	Mouse m;
	Mousectl *mc;

	mc = arg;
	for(;;){
		n = read(mc->mfd, buf, sizeof buf);
		if(n != 1+4*12){
			_drawprint(2, "mouse: bad count %d not 49: %r0, n);
			continue;
		}
		m.xy = Pt(atoi(buf+1+0*12), atoi(buf+1+1*12));
		m.buttons = atoi(buf+1+2*12);
		m.msec = atoi(buf+1+3*12);
		send(mc->c, &m);
	}
}
```

The central proc takes care of the screen graphics. It has a little over 30 threads. Each button, tab, text entry widget, etc. has its own thread. These threads *alt* on five channels: the *mouse*, *keyboard* and *resize* channels on which the associated events arrive, the *ctl* channel on which application requests arrive, such as ‘‘change background color’’ ‘‘change icon’’, ‘‘set state,’’ and the *exit* channel, which tells the thread to clean up and exit. When an interesting event occurs, the widget thread sends a description on its *event* channel, where it is picked up by another part of the application. The implementation of these widgets is in the Plan 9 *control*(2) library.

These threads have very small stacks (1 or 2 KB), so they consume very little memory; at the same time they greatly simplify the structure of the program: the management of the buttons, sliders, etc. does not contaminate the structure of the main loop of the program, which looks like this:

```
for(;;) {
     switch(alt(alts)) {
     case Efisin:
          ...  // Mouse event received
     case Tab:
          ...  // GPS/Info/Plan/Mode tab
     case Chartout:
          ...  // Map update request sent
     case Chartin:
          ...  // Updated map received
     case Gpsin:
          ...  // GPS fix received
     case Zoom:
          ...  // Zoom request
     case Quit:
          threadexitsall(nil);
     }
}
```

Even though there are 5 processes and 30 threads, there is not a single lock or mutex in the application code. The thread library itself, however, does need locks; but the complexity of that code has been amortized over dozens of applications.

Channels are protected against simultaneous access from different procs and this makes them eminently suitable for all sorts of synchronization between threads and procs. A simple example is buffer management in an environment with one or more producers and consumers: Create two buffered channels *empty* and *full* and initialize them as follows:

```
empty = chancreate(sizeof(Buf *), Nbuf);
full = chancreate(sizeof(Buf *), Nbuf);

for (i = 0; i < Nbuf; i++)
     sendp(empty, &buf[i]);
```

When a producer needs an empty buffer, it reads (*recvp*()) its address from *empty*. The producer blocks if there are no more empty buffers. It could also read the address of an empty buffer using *nbrecvp*, the non-blocking version of the receive call and, for example, give an error message when *nil* is returned. The producer fills the buffer and sends its address to *full*. Consumers get their buffers from *full* and return the empty ones on code*empty.*The

File servers typically need as many threads as they have simultaneous client requests to process. One possibility is to create a new thread for each incoming request and to have those threads exit when they've returned their reply. It is more efficient to use a pool of idle threads. These threads, when they are available to accept new work, will receive from a channel on which work is posted. The central work dispatcher sends work on this channel using non-blocking send. If the send fails, there are no idle workers and a new worker can be created to take on the job.

We now have a few years of experience programming with channels and threads and it has become clear that this has been much easier than any of the other techniques we have used. But, as with all things, channels, threads and procs are not a panacea. A disciplined programming style contributes more to correct and debuggable software than anything else. You may have noticed we program in C, not in C++ or Java. Object-orientation is a property of disciplined programming, programming languages only provide a bit of syntactic sugar. Languages cannot force bad programmers to write good code.

The Plan 9 thread library was created long after the Plan 9 kernel interface had been defined. As a result, some things in the thread library are downright clumsy. It would be nice if the operating system had support for multiple (small!) stack segments. Now, we *malloc*(2) stacks and we find every now and then that we allocated too little and we get obscure crashes. Note (*signal* in Unix terms) handling is also complicated. One day, perhaps, we'll bite some of these bullets and build some of the thread support into the operating system kernel.

A final remark is that there is a Linux version of the thread library that will need some polishing before it is once again completely usable, but it is there for those who are interested. Write to `sape@plan9.bell-labs.com`.