# ♦ The Inferno™ Operating System

*Sean M. Dorward, Rob Pike, David Leo Presotto,*
*Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom*

*The Inferno™ operating system facilitates the creation and support of distributed services in the new and emerging world of network environments, such as those typified by CATV and direct satellite broadcasting systems, as well as the Internet. In addition, as the entertainment, telecommunications, and computing industries converge and interconnect, different types of data networks are arising, each one as potentially useful and profitable as the telephone network. However, unlike the telephone system, which started with standard terminals and signaling, these new networks are developing in a world of diverse terminals, network hardware, and protocols. Inferno is designed so that it can insulate the diverse providers of content and services from the equally varied transport and presentation platforms. The Inferno Business Unit of Lucent Technologies and the Computing Sciences Research Center of Bell Labs, the R&D arm of Lucent, designed it specifically as a commercial product. It is intended for licensing in the marketplace and for use in conjunction with new Lucent offerings. Inferno incorporates many years of Bell Labs research in operating systems, languages, on-the-fly compilers, graphics, security, networking, and portability in providing an effective and economical network operating system.*

## Introduction

The Inferno™ operating system[1] is designed to be used in a variety of network environments—for example, those supporting advanced telephones, hand-held devices, TV set-top boxes attached to cable or satellite systems, and inexpensive Internet computers—but also in conjunction with traditional computing systems.

Among the most visible new environments are CATV, direct satellite broadcasting, and the Internet. As the entertainment, telecommunications, and computing industries converge and interconnect, data networks in various forms are emerging, each potentially as useful and profitable as the telephone system. Unlike the telephone system, which started with standard terminals and signaling, these networks are developing in a world of diverse terminals, network hardware, and protocols. Only a well-designed and economical operating system can insulate the diverse providers of content and services from the equally varied transport and presentation platforms. Inferno is a network operating system for this new world.

Inferno's definitive strength lies in the following areas:

- *Portability across processors*. Currently, Inferno runs on Intel, SPARC,* MIPS, ARM, HP-PA, Power PC,* and AMD 29K* architectures and is readily portable to others.
- *Portability across environments*. Inferno runs as a stand-alone operating system on small terminals and also as a user application under the Windows NT,* Windows 95,* UNIX* (Irix,* Solaris,* Linux, AIX,* HP/UX,* NetBSD), and Plan 9™ systems. In all these environments, Inferno applications see an identical interface.
- *Distributed design*. The identical environment is established at both a user's terminal and a

**Panel 1. Abbreviations, Acronyms, and Terms**

| | |
|---|---|
| ATM | asynchronous transfer mode |
| CA | certifying authority |
| CATV | cable television |
| DESCBC | DES chain block coding |
| DES | Data Encryption Standard |
| DESECB | DES electronic code book |
| GIF | Graphics Interchange Format |
| I/O | input/output |
| IP | Internet Protocol |
| JPEG | Joint Photographic Experts Group |
| MPEG | Motion Picture Experts Group |
| OA&M | operations, administration, and maintenance |
| POTS | "plain old telephone service" |
| PPP | Point-to-Point Protocol |
| SHA | Secure Hash Algorithm |
| SSL | Secure Sockets Layer |
| STS | Station-to-Station |
| TCP/IP | Transmission Control Protocol/Internet Protocol |

server, and each environment may import the resources of the other (for example, the attached I/O devices or networks). Aided by the communications facilities of the run-time system, applications may be split easily (and even dynamically) between client and server.

- *Minimal hardware requirements*. Inferno runs useful applications as stand-alone programs on machines with as little as 1 MB of memory, and it does not require memory-mapping hardware.
- *Portable applications*. Inferno applications are written in the type-safe Limbo™ language, whose binary representation is identical over all platforms.
- *Dynamic adaptability*. Depending on the hardware or other resources available, applications may load different program modules to perform a specific function. For example, a video player application might use any of several different decoder modules.

Underlying the design of Inferno is a model of the diversity-of-application areas it intends to stimulate. Many suppliers are interested in purveying media and services—telephone network service providers, Web servers, cable companies, merchants, and various information services firms. Currently, many connection technologies are available—for example, ordinary telephone modems, ISDN, ATM, the Internet, analog broadcast TV or CATV, cable modems, digital video on demand, and other interactive TV systems.

Applications more clearly related to Lucent Technologies' current and planned product offerings include control of switches and routers and the associated operations system facilities needed to support them. For example, Inferno software will control an IP switch/router for voice and data being developed by Lucent's Bell Labs Research and Network Systems organizations. An Inferno-based firewall called Signet is being used to secure outside access to the research organization's Internet connection.

Finally, existing or potential hardware endpoints must be considered. Some are located in consumers' homes in the form of PCs, game consoles, and newer set-top boxes. Others are located inside the networks themselves in the form of nodes for billing, network monitoring, or provisioning. The higher ends of these spectra, such as fully interactive TV with video on demand, may be fascinating, but they have developed more slowly than expected. One reason is the cost of the set-top box—especially its memory requirements. Portable terminals are similarly constrained because of weight and cost considerations.

Inferno is parsimonious enough in its resource requirements to support interesting applications on today's hardware while being versatile enough to grow into the future. In particular, it enables developers to create applications that will work across a range of facilities. An example of such an application is an interactive shopping catalog that works in text mode over a POTS modem, shows still pictures (perhaps with audio) of the merchandise over ISDN, and includes video clips over digital cable.

Clearly, not everyone who deploys an Inferno-based solution will want to span the whole range of possibilities. However, the system architecture should be constrained only by the desired markets and the available interconnection and server technologies, not by the software.

## Inferno Interfaces

The role of the Inferno system is to *create* several standard interfaces for its applications:

- Applications use various resources internal to the system. These resources include a consistent virtual machine that runs the application programs together with library modules that perform services as simple as string manipulation through more sophisticated graphics services for dealing with text, pictures, higher-level toolkits, and video.
- Applications exist in an external environment containing such resources as data files that can be read and manipulated, together with objects that are named and manipulated like files but are more active. Devices (for example a hand-held remote control, an MPEG decoder, or a network interface) present themselves to the application as files.
- Standard protocols exist for communication within and between separate machines running Inferno so that applications can cooperate.

At the same time, Inferno *uses* interfaces supplied by an existing environment, either bare hardware or standard operating systems and protocols.

Typically, an Inferno-based service would consist of many relatively inexpensive terminals running Inferno as a native system and fewer large machines running it as a hosted system. On these server machines, Inferno might interface to databases, transaction systems, existing OA&M facilities, and other resources provided under the native operating system. The Inferno applications themselves would run either on the client or server machines or on both.

## External Environment of Inferno Applications

The purpose of most Inferno applications is to present information or media to the user. Thus, applications must locate the information sources in the network and construct a local representation of them. The information flow, however, is not one way. The user's terminal (whether it is a network computer, TV set-top box, PC, or videophone) is also an information source, and its devices represent resources to applications. Inferno draws heavily on the design of the Plan 9 operating system[2] in the way it presents resources to these applications.

The design has three principles:

- All resources are named and accessed like files in a forest of hierarchical file systems.
- The disjoint resource hierarchies provided by different services are joined into a single private and hierarchical *name space*.
- A communication protocol called Styx is applied uniformly to access the resources regardless of whether they are local or remote.

In practice, most applications see a fixed set of files organized as a directory tree. Some of the files contain ordinary data but others represent more active resources. Devices are represented as files, and device drivers (such as modems, MPEG decoders, network interfaces, or TV screens) attached to a particular hardware box present themselves as small directories. These directories typically contain two files, `data` and `ctl`, which respectively perform actual device I/O and control operations. System services also live behind filenames. For example, an Internet domain name server might be attached to an agreed-on name (say `/net/dns`). After writing to this file, which is a string representing a symbolic Internet domain name, a subsequent read from the file would return the corresponding numeric Internet address.

The glue that connects the separate parts of the resource name space together is the Styx protocol. Within an instance of Inferno, all the device drivers and other internal resources respond to the procedural version of Styx. The Inferno kernel implements a *mount driver* that transforms file system operations into remote procedure calls for transport over a network. On the other side of the connection, a server unwraps the Styx messages and implements them using resources local to it. Therefore, it is possible to import parts of the name space (and thus resources) from other machines.

To extend the example above, it is unlikely that a set-top box would store the code needed for an Internet domain name-server within itself. Instead, an Internet browser would import the `/net/dns` resource into its own name space from a server machine across a network.

The Styx protocol lies above and is independent of the communications transport layer. It is readily carried over the TCP/IP, PPP, ATM, or various modem transport protocols.

## Internal Environment of Inferno Applications

Inferno applications are written in Limbo,[3] a new language that was designed specifically for the Inferno environment. Its syntax is influenced by C and Pascal, and it supports the standard data types common to them together with several higher-level data types, such as lists, tuples (groups of values), strings, dynamic arrays, and simple abstract data types.

In addition, Limbo supplies several advanced constructs, which are carefully integrated into the Inferno virtual machine. In particular, a communication mechanism called a *channel* is used to connect different Limbo tasks on the same machine or across the network. A channel transports typed data in a machine-independent fashion so that complex data structures (including channels themselves) may be passed between Limbo tasks or attached to files in the name space for language-level communication between machines.

Multi-tasking is supported directly by the Limbo language. Independently scheduled threads of control may be spawned, and an `alt` statement is used to coordinate the channel communication between tasks (that is, `alt` is used to select one of several channels that are ready to communicate). By building channels and tasks into the language and its virtual machine, Inferno encourages a communication style that is safe and easy to use.

Limbo programs are built of *modules*, which are self-contained units having a well-defined interface containing functions (methods), abstract data types, and constants defined by the module and visible outside it. Modules are accessed dynamically—that is, when one module wishes to make use of another, it dynamically executes a `load` statement naming the desired module and it uses a returned handle to access the new module. When the module is no longer in use, its storage and code will be released. The flexibility of the modular structure contributes to the smallness of typical Inferno applications, as well as to their adaptability. For example, in the shopping catalog described above, the application's main module checks dynamically for the existence of the video resource. If it is not available, the video-decoder module is never loaded.

Limbo is fully type-checked at both compile and run time. For example, pointers—besides being more restricted than in C—are checked before being dereferenced, and the type-consistency of a dynamically loaded module is checked when it is loaded. Limbo programs run safely on a machine without memory-protection hardware. Moreover, all Limbo data and program objects are subject to a garbage collector built deeply into the Limbo run-time system. All system data objects are tracked by the virtual machine and freed as soon as they become idle. For example, if an application task creates a graphics window and then terminates, the window automatically disappears the instant the last reference to it goes away.

Limbo programs are compiled into byte codes representing instructions for a virtual machine called Dis™. The architecture of the arithmetic part of Dis is a simple three-address machine supplemented with a few specialized operations for handling some of the higher-level data types, such as arrays and strings. Garbage collection is handled below the level of the machine language. Task scheduling is similarly hidden. When loaded into memory for execution, the byte codes are expanded into a format more efficient for execution. In addition, an optional on-the-fly compiler turns a Dis instruction stream into native machine instructions for the appropriate real hardware. This can be done efficiently because Dis instructions closely match the instruction-set architecture of today's machines. The resulting code executes at a speed approaching that of compiled C.

Underlying Dis is the Inferno kernel, which contains both the interpreter and on-the-fly compiler, as well as memory management, scheduling, device drivers, protocol stacks, and the like. The kernel also contains the core of the file system (the name evaluator and the code that turns file system operations into remote procedure calls over communications links) and the small file systems implemented internally.

Finally, the Inferno virtual machine implements several standard modules internally. These modules include `Sys`, which provides system calls and a small library of useful routines (for example, creation of network connections and string manipulations). Module `Draw` is a basic graphics library that handles raster graphics, fonts, and windows. Module `Prefab` builds on `Draw` to provide structured complexes containing images and text inside windows. These elements may be scrolled, selected, and changed by the methods of `Prefab`. Module `Tk` is an all-new implementation of the Tk graphics toolkit[4] with a Limbo interface. A `Math` module encapsulates the procedures for numerical programming.

## Environment of the Inferno System

Inferno creates a standard environment for applications. Identical application programs can run under any instance of this environment—even in distributed fashion—and see the same resources. Several versions of the Inferno kernel, Dis/Limbo interpreter, and device driver set can be used depending on the environment within which Inferno itself is implemented.

When running as the native operating system, the kernel includes all the low-level glue (interrupt handlers, graphics, and other device drivers) needed to implement the abstractions presented to applications. For a hosted system—for example, one hosted under UNIX, Windows NT or Windows 95—Inferno runs as a set of ordinary processes. Instead of mapping its device-control functionality to real hardware, it adapts to the resources provided by the operating system under which it runs. Under UNIX, for instance, the graphics library might be implemented using the X Window System* and the networking using the socket interface. Under Windows,* the library uses the native Windows graphics and WinSock calls.

To the extent possible, Inferno is written in standard C language, and most of its components are independent of the many operating systems that can host it.

## Security Issues

Inferno provides security of communication, resource control, and system integrity. Each external communication channel may be transmitted in the clear, accompanied by message digests to prevent corruption, or encrypted to prevent corruption and interception. Once communication is established, channel encryption is transparent to the application. Key exchange is provided through standard public key mechanisms. After key exchange, message digesting and line encryption both use standard symmetric mechanisms.

Inferno is secure against erroneous or malicious applications and encourages safe collaboration between mutually suspicious service providers and clients. The resources available to applications appear exclusively in the name space of the application, and standard protection modes are available. This applies to data, communication resources, and the executable modules that constitute the applications. Security-sensitive resources of the system are accessible only by calling the modules that provide them. In particular, adding new files and servers to the name space—an authenticated operation—is controlled. For example, if the network resources are removed from an application's name space, then it is impossible for it to establish new network connections.

Object modules may be signed by trusted authorities who guarantee their validity and behavior. These signatures may be checked by the system the modules are accessing.

Although Inferno provides a rich variety of authentication and security mechanisms as detailed below, few application programs need to be aware of them or to include coding explicitly to make use of them. Most often, access to resources across a secure communications link is arranged in advance by the larger system in which the application operates. For example, when a client system uses a server system and connection authentication or link encryption is appropriate, the server resources will most naturally be supplied as part of the application's name space. The communications channel that carries the Styx protocol can be set to authenticate or encrypt. Thereafter, all use of the resource is automatically protected.

## Security Mechanisms

Authentication and digital signatures are performed using public key cryptography. Public keys are certified by Inferno-based or other certifying authorities who sign the public keys with their own private key. Inferno uses encryption for:

- Mutual authentication of communicating parties,
- Authentication of messages between these parties, and
- Encryption of messages between these parties.

The encryption algorithms Inferno provides include the SHA, MD4, and MD5 secure hashes; Elgamal public key signatures and signature verification;[5] RC4 encryption; DES encryption; and public key exchange based on the Diffie-Hellman scheme. The public key signatures use keys with moduli up to 4,096 bits (512 bits by default).

No generally accepted national or international authority exists for storing or generating public or private encryption keys. Thus, Inferno includes tools for using or implementing a trusted authority, but it does not itself provide the authority, which is an administrative function. An organization using Inferno (or any other security and key-distribution scheme) must design a system to suit its own needs. In particular, an organization must decide whom to trust as a certifying authority (CA). However, the Inferno design is sufficiently flexible and modular to accommodate the protocols likely to be attractive in practice.

The CA that signs a user's public key determines the size of the key and the public key algorithm used. Tools provided with Inferno use these signatures for authentication. Library interfaces are provided for Limbo programs to sign and verify signatures.

Generally, authentication is performed using public key cryptography. Parties register by having their public keys signed by the CA. The signature covers a secure hash (SHA, MD4, or MD5) of the name of the party, the party's public key, and an expiration time. The signature—which contains the name of the signer—along with the signed information, is termed a *certificate*.

When parties communicate, they use the STS protocol[6] to establish the identities of the two parties and to create a mutually known secret. The STS protocol uses the Diffie-Hellman algorithm[7] to create this shared secret. The protocol is protected against replay attacks by choosing new random parameters for each conversation. It is secured against man-in-the-middle attacks by requiring the parties to exchange certificates and then digitally signing key parts of the protocol. To masquerade as another party, an attacker must be able to forge that party's signature.

## Line Security

A network conversation can be secured against modification alone or against both modification and snooping. To secure against modification, Inferno can append a secure MD5 or SHA hash (called a digest),

hash(secret, message, messageid)

to each message. *Messageid* is a 32-bit number that starts at 0 and is incremented by one for each message sent. Thus, messages cannot be changed, removed, reordered, or inserted into the stream without knowing the secret or breaking the secure hash algorithm.

To secure against snooping, Inferno supports encryption of the complete conversation using either RC4 or the DES with either DES chain block coding (DESCBC) or the DES electronic code book (DESECB).

Inferno uses the same encapsulation format as Netscape's SSL protocol. It is possible to encapsulate a message stream in multiple encapsulations to provide varying degrees of security.

## Random Numbers

The strength of cryptographic algorithms depends in part on the strength of the random numbers used for choosing keys, Diffie-Hellman parameters, and initialization vectors. Inferno achieves this strength in two steps. First, a slow (100- to 200-b/s) random bitstream comes from sampling the low-order bits of a free-running counter whenever a clock ticks. The clock must be unsynchronized or at least poorly synchronized with the counter. This generator is then used to alter the state of a faster pseudo-random number generator. Both the slow and fast generators were tested on a number of architectures using self correlation, random walk, and repeatability tests.

## Introduction to Limbo

The application programming language for the Inferno operating system is Limbo. Although Limbo looks syntactically like C, it has a number of features that make it easier to use, safer, and more suited to the heterogeneous and networked Inferno environment. For instance, Limbo has a rich set of basic types, strong typing, garbage collection, concurrency, communications, and modules. It may be interpreted or compiled just in time for efficient and portable execution.

This paper introduces the language by studying an example of a complete and useful Limbo program. The program illustrates general programming, as well as aspects of concurrency, graphics, module loading, and other features of Limbo and Inferno.

## The Problem

Our example program is a stripped-down version of the Inferno program `view`, which displays graphical image files on the screen—one per window. This version sacrifices some functionality, generality, and error-checking but still performs the basic job. The files may be configured either in the GIF[8,9] or JPEG[10] format, and they must be converted before display; or they may already be encoded in the Inferno standard format that needs no conversion. `View` "sniffs" each file to determine what processing it requires, maps the colors if necessary, creates a new window, and copies the converted image to the window. Each window is given a title bar across the top to identify it and to store or hold the buttons that move and delete the window.

## The Source

The complete Limbo source for our version of `view` is shown in **Panel 2.** The source is annotated with line numbers for easy reference (Limbo, of course, does not use line numbers). Subsequent sections explain the workings of the program. Although the program is too large to absorb as a first example without some assistance, we recommend skimming the program before moving on to the next section to become familiar with the style of the language. Control syntax derives from C[11] while declaration syntax comes from the Pascal family of languages.[12] Limbo borrows features from a number of languages (for example, tuples on lines 45 and 48) and introduces a few new ones (such as explicit module loading on lines 90 and 92).

## Modules

Limbo programs are composed of modules that are loaded and linked at run time. Each Limbo source file is the implementation of a single module. In Panel 2, line 1 states that this file implements a module called `View` whose declaration appears in the `module` declaration on lines 15 through 18. The declaration states that the module has one publicly visible element—the function `init`. Other functions and variables defined in the file will be compiled into the module but they will only be accessible internally.

The function `init` has a type signature (argument and return types) that makes it callable from the Inferno shell, a convention not made explicit here. The type of `init` allows `View` to be invoked by typing, for example,

```
view *.jpg
```

at the Inferno command prompt to view all the JPEG files in a directory. This interface is the only requirement that enables the shell to call the module. All programs are constructed from modules, and the shell can directly call some modules because of their type. In fact, the shell invokes `View` by loading it and calling `init`—not, for example, through the services of a system `exec` function as in a traditional operating system.

Of course, not all modules implement shell commands. Modules are also used to construct libraries, services, and other program components. The module `View` uses the services of other modules for I/O, graphics, file format conversion, and string processing. These modules are identified on lines 2 through 14. Each module's interface is stored in a public include file that holds a definition of a module in much the same manner as lines 15 through 18 of the `View` program. For example, the following is an excerpt from the include file `sys.m`:

```
Sys: module
{
   PATH:  con   "$Sys";

   FD: adt   # File descriptor
   {
      fd:   int;
   };

   OREAD:   con 0;
   OWRITE:  con 1;
   ORDWR:   con 2;

   open:   fn(s: string, mode: int):
              ref FD;
   print:  fn(s: string, *): int;
   read:   fn(fd: ref FD,
              buf: array of byte,
              n: int): int;
   write:  fn(fd: ref FD,
              buf: array of byte,
              n: int): int;
};
```

This example defines a module type called `Sys` that has functions with such familiar names as `open` and `print`, constants like `OREAD` to specify the mode for opening a file, an aggregate type (`adt`) called `FD` returned by `open`, and a constant string called `PATH`.

After including the definition of each module, `View` declares variables to access each module. Line 3, for instance, declares the variable `sys` to have type `Sys`. It will be used to hold a reference to the implementation of the module. Line 6 imports a number of types from the `draw` (graphics) module to simplify their use. This line states that by default, the implementation of these types is to be that provided by the module referenced by the variable `draw`. Without such an `import` statement, calls to methods of these types would require explicit mention of the module providing the implementation.

Unlike most module languages, which resolve unbound references to modules automatically, Limbo requires explicit loading of module implementations. Although this requires more bookkeeping, it allows a program to have fine control over the loading (and unloading) of modules, an important property in the small-memory systems in which Inferno is intended to run. Additionally, it allows easy garbage collection of unused modules and permits multiple implementations to serve a single interface, a style of programming we will exploit in `View`.

Declaring a module variable, such as `sys`, is not sufficient to access a module. An implementation must also be loaded and bound to the variable. Lines 21 through 25 load the implementations of the standard modules used by `View`. The load operator—for example,

```
sys = load Sys Sys->PATH;
```

takes a type (`Sys`), the filename of the implementation (`Sys->PATH`), and loads it into memory. If the implementation matches the specified type, a reference to the implementation is returned and stored in the variable (`sys`). If not, the constant `nil` will be returned to indicate an error. Conventionally, the `PATH` constant defined by a module names the default implementation. Because `Sys` is a built-in module provided by the system, it has a special form of name. Other modules' `PATH` variables name files containing actual code—for example, `Wmlib->PATH` is `"/dis/lib/wmlib.dis"`. Note, though, that the name of the implementation of the module in a `load` statement can be any string.

Line 26 initializes the `wmlib` module by invoking its `init` function (unrelated to the `init` of `View`). Note the use of the -> operator to access the member function of the module. The next two lines load modules and also introduce some new notation—they *declare* and *initialize* the module variables storing the reference. Limbo declarations have the general form

*var : type = value;*

If the type is missing, it is taken to be the type of the value. So, for example,

```
bufio := load Bufio Bufio->PATH;
```

on line 28 declares a variable of type `Bufio` and initializes it to the result of the `load` expression.

## The Main Loop

The `init` function takes two parameters: a graphics context (`ctxt`) for the program, and a list of

**Panel 2. An Example of a Limbo Program**

```
implement View;
include “sys.m”;
    sys:Sys;
include “draw.m”;
    draw: Draw;
Rect, Display, Image: import draw;
include “bufio.m”;
include “imagefile.m”;
include “tk.m”;
    tk: Tk;
 include  “wmlib.m”;
    wmlib: Wmlib;
 include “lib.m”;
    str: String;
 View: module
 {
    init: fn(ctxt: ref Draw->Context,
             argv: list of string);
 };
 init(ctxt: ref Draw->Context,
      argv: list of string)
 {
    sys   = load Sys Sys->PATH;
    draw  = load Draw Draw->PATH;
    tk    = load Tk Tk->PATH;
    wmlib = load Wmlib Wmlib->PATH;
    str   = load String String->PATH;
    wmlib->init();
    imageremap := load Imageremap
                       Imageremap->PATH;
    bufio := load Bufio Bufio->PATH;

    argv = tl argv;
    if(argv != nil
       && str->prefix(“-x “, hd argv))
       argv = tl argv;

    viewer := 0;
    while(argv != nil){
       file := hd argv;
       argv = tl argv;

       im := ctxt.display.open(file);
       if(im == nil){
          idec := filetype(file);
          if(idec == nil)
             continue;

          fd := bufio->open(file,
                         Bufio->OREAD);
          if(fd == nil)
             continue;

          idec->init(bufio);
          (ri, err) := idec->read(fd);
          if(ri == nil)
             continue;

         (im, err) = imageremap->remap(
                 ri, ctxt.display, 1);
          if(im == nil)
             continue;
       }

       spawn view(ctxt, im, file,
                  viewer++);
    }
 }
view(ctxt: ref Draw->Context,
     im: ref Image, file: string,
     viewer: int)
{
   corner := string(25+20*(viewer%5));
   t := tk->toplevel(ctxt.screen,
         “ -x “+corner+” -y “+corner+
         “ -bd 2 -relief raised”);

   (nil, file) = str->splitr(file,
         “/”);
   menubut := wmlib->titlebar(t,
         “View: “+file, Wmlib->Hide);

   event := chan of string;
   tk->namechan(t, event, “event”);
   tk->cmd(t, “frame .im -height “ +
              string im.r.dy() +
              “ -width “ +
              string im.r.dx());
   tk->cmd(t, “bind . <Configure> “+
              “{send event resize}”);
   tk->cmd(t, “bind . <Map> “+
              “{send event resize}”);
   tk->cmd(t, “pack .Wm_t -fill x”);
   tk->cmd(t, “pack .im -side bottom”+
             “ -fill both -expand 1”);
   tk->cmd(t, “update”);

   t.image.draw(posn(t), im,
     ctxt.display.ones, im.r.min);
   for(;;) alt{
   menu := <-menubut =>
      if(menu == “exit”)
         return;
      wmlib->titlectl(t, menu);
   <-event =>
      t.image.draw(posn(t), im,
         ctxt.display.ones, im.r.min);
   }
}

posn(t: ref Tk->Toplevel): Rect
{
   minx := int tk->cmd(t,
               “.im cget -actx”);
   miny := int tk->cmd(t,
               “.im cget -acty”);
   maxx := minx + int tk->cmd(t,
               “.im cget -actwidth”);
   maxy := miny + int tk->cmd(t,
               “.im cget -actheight”);

  return ((minx, miny), (maxx, maxy));
}

filetype(file: string): RImagefile
{
   if(len file>4
     && file[len file-4:]==”.gif”)
      r := load RImagefile
              RImagefile->READGIFPATH;
   if(len file>4
     && file[len file-4:]==”.jpg”)
      r = load RImagefile
              RImagefile->READJPGPATH;
   return r;
}
```

command-line argument strings (`argv`). `Argv` is a `list of string`. Strings are a built-in type in Limbo, and lists are a built-in form of constructor. Lists have several operations defined: `hd` (head) returns the first element in the list, `tl` (tail) returns the remainder after the head, and `len` (length) returns the number of elements in the list.

In Panel 2, line 29 throws away the first element of `argv`, which is the conventional name of the program being invoked by the shell; lines 30 and 31 ignore a geometry argument passed by the window system. The loop from lines 33 to 53 processes each file named in the remaining arguments. When `argv` is a `nil` list, the loop is complete. Line 34 picks off the next filename, and line 35 updates the list.

Line 36 is the first method call we have seen:

```
    im := ctxt.display.open(file);
```

The parameter `ctxt` is an `adt` that contains all the relevant information for the program to access its graphics environment. One of its elements called `display` represents the connection to the frame buffer on which the program may write. The `adt display` (whose type is imported on line 6) has a member function `open` that reads a named image file into the memory associated with the frame buffer, returning a reference to the new image. (In X[13] terminology, `display` represents a connection to the server and `open` reads a pixmap from a file and instantiates it on that server.)

The `display.open` method succeeds only if the file exists and is configured in the standard Inferno image format. If it fails, it will return `nil`, and lines 38 through 50 will attempt to convert the file into the right form.

## Decoding the File

Line 38 in Panel 2 calls `filetype` to determine what format the file has. The simple version shown on lines 87 through 94 just looks at the file suffix to determine the type. A realistic implementation would work harder but even this version illustrates the utility of program-controlled loading of modules.

The decoding interface for an image file format is specified by the module type `RImagefile`. However, unlike the other modules we have examined, `RImagefile` has a number of implementations. If the file is a GIF file, `filetype` returns the implementation of `RImagefile` that decodes GIFs. If it is a JPEG file, `filetype` returns an implementation that decodes JPEGs. In either case, the `read` method has the same interface. Because reference variables like `r` are implicitly initialized to `nil`, that is what `filetype` will return if it does not recognize the image format. Thus, `filetype` accepts a filename and returns the implementation of a module to decode it.

Two other aspects of `filetype` are worth mentioning. First, the expression `file [ len file-4:]` is a *slice* of the string file. It creates a string holding the last four characters of the filename. The colon separates the starting and ending indices of the slice. The missing second index defaults to the end of the string. As with lists, `len` returns the number of characters (not bytes; Limbo uses Unicode[14] throughout) in the string.

Second and more importantly, this version of `filetype` loads the decoder module anew every time it is called, which is clearly inefficient. It's easy to do better, though. Just store the module in a global, as in this fragment:

```
readjpg: RImagefile;
filetype(...)...
{
    if(isjpg()){
        if(readjpg == nil)
            readjpg = load RImagefile
                RImagefile->READJPGPATH;
        return readjpg;
    }
}
```

The program can form its own policies on loading and unloading modules based on time/space or other tradeoffs. The system does not impose its own policies.

Returning to the main loop, after the type of the file has been discovered, line 41 opens the file for I/O using the buffered I/O package. Line 44 calls the `init` function of the decoder module, passing it the instance of the buffered I/O module being used (if we were caching decoder modules, this call to `init` would be done only when the decoder is first loaded.) Finally, the Limbo-characteristic line 45 reads in the file:

```
        (ri, err) := idec->read(fd);
```

The `read` method of the decoder does the hard job of cracking the image format, which is beyond the scope of this paper. The result is a *tuple* or pair of values. The first element of the pair is the image while the second element is an error string. If all goes well, the `err` will be `nil`. If a problem surfaces, however, `err` may be printed by the application to report what went wrong. The interesting property of this style of error reporting—common to Limbo programs—is that an error can be returned even if the decoding was successful (that is, even if `ri` is non-`nil`). For example, the error may be recoverable. In this case, it is worth returning the result but also worth reporting that an error did occur, leaving the application to decide whether to display the error or ignore it. (`View` ignores it, for brevity.)

In a similar manner, line 48 remaps the colors from the incoming color map associated with the file to the standard Inferno color map. The result is an image ready to be displayed.

## Creating a Process

By line 52 in the main loop (see Panel 2), we have an image ready in the variable `im` and use the Limbo primitive `spawn` to create a new process to display that image on the screen. `Spawn` operates on a function call, creating a new process to execute that function. The process doing the spawning—here the main loop—continues immediately while the new process begins execution in the specified function with the specified parameters. Thus, line 52 begins a new process in the function `view` with these arguments: the graphics context, the image to display, the filename, and a unique identification number used in placing the windows.

The new process shares with the calling process all variables except the stack. Therefore, shared memory can be used to communicate between them. For synchronization, a more sophisticated mechanism is needed, a subject we will cover in the "Communications" section.

## Starting Tk

The function `view` uses the Inferno Tk graphics toolkit (a reimplementation for Limbo of the Tcl/Tk toolkit[3]) to place the image on the screen in a new window. In Panel 2, line 57 computes the position of the corner of the window using the viewer number to stagger the positions of successive windows. The `string` keyword is a conversion. In this example, the conversion does an automatic translation from an integer expression into a decimal representation of the number. Thus, `corner` is a string variable, a form more useful in the calls to the Tk library.

The Inferno Tk implementation uses Limbo as its controlling language. Rather than building a rich procedural interface, the interface passes strings to a generic Tk command processor, which returns strings as results. This process is similar to the use of Tk within Tcl but with most of the control flow, arithmetic, and so on written in Limbo.

A good introduction to the style is the function `posn` on lines 79 through 86. The calls to `tk->cmd` evaluate the textual command in the context defined by the `Tk->Toplevel` variable `t` (created on line 58 and passed to `posn`). The result is a decimal integer, which the explicit `int` conversion converts to binary. On line 85, all the coordinates of the rectangle are known, and the function returns a nested tuple defining the rectangular position of the `.im` component of the top level. This tuple is automatically promoted to the `Rect` type by the return statement.

Back in function `view`, line 58 calls `tk->toplevel` to create the window on the display. The arguments are `ctxt.screen`, a data structure representing the window stack on the frame buffer, and a string specifying the size and properties of the new window. The `+` operator on strings performs concatenation. The return value from `tk->toplevel` is a reference to a top-level widget—a window—on which the program will assemble its display.

Line 59 uses a function from the higher-level `String` module to strip off the basename of the filename for use in the banner of the window. Note that one component of the tuple is nil; the value of this component is discarded. Line 60 calls the window manager function `wmlib->titlebar` to establish a title bar on the window labeled "`View:`"

and on the file basename with a control button to hide the window. Title bars always include a control button to dismiss the window.

## Communications

The return value from `wmlib->titlebar` is a built-in Limbo type called a *channel* (`chan` is the keyword). A channel is a communications mechanism in the manner of communicating sequential processes.[15] Two processes that wish to communicate do so using a shared channel. Data sent on the channel by one process may be received by another process. The communication is *synchronous*—that is, both processes must be ready to communicate before the data changes hands. If one process is not ready, the other blocks until it is. Channels are a feature of the Limbo language. They have a declared type ( for example, `chan of int` and `chan of list of string`), and only data of the correct type may be sent. No restriction limits what may be sent. One may even send a channel on a channel. Therefore, channels serve both to communicate and to synchronize.

Channels are used throughout Inferno to provide interfaces to system functions. The threading and communications primitives in Limbo are not designed to implement efficient multicomputer algorithms but rather to provide an elegant way to build active interfaces to devices and other programs.

One example is the `menubut` channel returned by `wmlib->titlebar`, a channel of textual commands sent by the window manager. The expression

```
menu := <-menubut
```

on line 71 in Panel 2 receives the next message on the channel and assigns it to the variable `menu`. The communications operator, `<-`, receives a datum when prefixed to channel and transmits a datum when combined with an assignment operator (for example, `channel<-=2`). This use of `menubut` appears inside an `alt` (alternation) statement, a construct we will discuss later.

Lines 61 and 62 create and register a new channel, `event`, to be used by the Tk module to report user interface events. Lines 63 through 68 use simple Tk operations to make the window in which the image may be drawn. Lines 64 and 65 bind events within this window to messages to be sent on the channel `event`. For example, line 64 defines that when the configuration of the window is changed—presumably by actions of the window manager—the string "`resize`" is to be transmitted on `event` for interpretation by the application. This translation of events into messages on explicit channels is fundamental to the Limbo style of programming.

## Displaying the Image

The payoff occurs on line 69 in Panel 2, which steps outside the Tk model to draw the image `im` directly on the window:

```
t.image.draw(posn(t), im,
  ctxt.display.ones, im.r.min);
```

`Posn` calculates where on the screen the image is to go. The `draw` method is the fundamental graphics operation in Inferno whose design is outside the scope of this discussion. In this statement, it just copies the pixels from `im` to the window's own image, `t.image`. The argument `ctxt.display.ones` is a mask that selects every pixel.

## Multi-Way Communications

Once the image is on the screen, `view` waits for any changes in the status of the window. Two things might happen: either the buttons on the title bar may be used, in which case a message will appear on `menubut`, or a configuration or mapping operation will apply to the window, in which case a message will appear on `event`.

The Limbo `alt` statement provides control when more than one communication may proceed. Analogous to a `case` statement, the `alt` evaluates a set of expressions and executes the statements associated with the correct expression. Unlike a `case`, though, the expressions in an `alt` each must be a communication, and the `alt` will execute the statements associated with the communication that can first proceed. If none can proceed, the `alt` waits until one can. If more than one statement can proceed, it chooses one randomly.

Thus, the loop on lines 70 through 77 in Panel 2 processes messages received by the two classes of actions. When the window is moved or resized, line 75 will receive a "`resize`" message due to the bindings on lines 64 and 65. The message is discarded but the action of receiving it triggers the repainting of the newly placed window on line 76. Similarly, messages triggered by buttons on the title bar send a message on `menubut`. The value of `menubut` is then examined to see if it is "`exit`", which should be handled locally, or anything else, which can be passed on to the underlying library.

## Cleanup

If the exit button is pushed, line 73 in Panel 2 will return from `view`. Because `view` was the top-level function in this process, the process will exit and free all its resources. All memory, open file descriptors, windows, and other resources the process holds will be collected as garbage when the return executes.

The Limbo garbage collector uses a hybrid scheme that combines reference counting with a real-time sweeping algorithm. Reference counting allows reclamation of memory the instant its last reference disappears; the sweeping algorithm runs as an idle-time process to reclaim unreferenced circular structures. The instant-free property means that system resources like file descriptors and windows can be tied to the collector for recovery as soon as they are no longer used. This property allows Inferno to run in smaller memory arenas than those required for efficient mark-and-sweep algorithms, and it also provides an extra level of programmer convenience.

## Summary

Inferno supplies a rich environment for constructing distributed applications that are portable—in fact, identical—even when running on widely divergent underlying hardware. Its unique advantage over other solutions is that it encompasses not only a virtual machine but also a complete virtual operating system, including network facilities.

### Acknowledgment

The cryptographic elements of Inferno owe much to the cryptographic library of Jack Lacy, Don Mitchell, and William Schell.[16]

### *Trademarks

AIX and Power PC are registered trademarks of International Business Machines Corp.

AMD 29K is a registered trademark of Advanced Micro Devices, Inc.

HP/UX is a registered trademark of Hewlett-Packard Inc.

Irix is a trademark of Silicon Graphics, Inc.

Solaris is a registered trademark of Sun Microsystems.

SPARC is a registered trademark of SPARC International.

UNIX is a registered trademark of Novell.

Windows, Windows NT, and Windows 95 are registered trademarks of Microsoft Corp.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

### References

1. R. Pike, D. L. Presotto, S. M. Dorward, B. Flandrena, K. Thompson, H. W. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Journal of Computing Systems*, Vol. 8, No. 3, Summer 1995, pp. 221-254.
2. S. M. Dorward, R. Pike, and P. Winterbottom, "Programming in Limbo," *Proceedings of the IEEE Computer Conference (COMPCON)*, San Jose, California, 1997.
3. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, New York, 1994.
4. T. Elgamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *Advances in Cryptography: Proceedings of CRYPTO 84*, Springer-Verlag, New York, 1985, pp. 10-18.
5. B. Schneier, *Applied Cryptography*, Chapter 22, John Wiley, New York, 1996, p. 516.
6. D. Stinson, *Cryptography, Theory and Practice*, CRC Press, Cleveland, Ohio, 1996, p. 271.
7. S. M. Dorward, R. Pike, D. M. Ritchie, H. W. Trickey, and P. Winterbottom, "Inferno," *Proceedings of the IEEE Computer Conference (COMPCON)*, San Jose, California, Feb. 1997.
8. *GIF Graphics Interchange Format: A Standard Defining a Mechanism for the Storage and Transmission of Bitmap-Based Graphics Information*, CompuServe Inc., Columbus, Ohio, 1987.
9. *GIF Graphics Interchange Format: Version 89a*, CompuServe Inc., Columbus, Ohio, 1990.
10. W. B. Pennebaker and J. L. Mitchell, *JPEG Still-Image Data Compression*, Van Nostrand Reinhold, New York, 1992.
11. *Programming Languages - C*, International Standards Organization (ISO), revision and redesignation of American National Standards Institute (ANSI) X3.159-1989, Amendment 1, 1990.
12. K. Jensen and N. Wirth, *Pascal—User Manual and Report*, Springer-Verlag, New York, 1974.
13. R. W. Scheifler, J. Gettys, and R. Newman,

*X Window System*, Digital Press, Bedford, Massachusetts, 1988.
14. The Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison-Wesley, New York, 1996.
15. C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the Association for Computing Machinery (ACM)*, Vol. 21, No. 8, 1978, pp. 666-677.
16. J. B. Lacy, D. P. Mitchell, and W. M. Schell, "CryptoLib: Cryptography in Software," *Proceedings of the UNIX Security Symposium IV*, USENIX, Santa Clara, California, 1993, pp. 1-17.

## Further Reading

– *Inferno—A Complete Platform to Develop and Deploy Intelligent Devices in Any Networked Environment*, Bell Labs Computing Sciences Research Center, Murray Hill, New Jersey, 1997. http://www.lucent.com/inferno

***(Manuscript approved March 1997)***

*SEAN M. DORWARD is a member of technical staff in the Computing Structures Research Department at Bell Labs in Murray Hill, New Jersey. He has worked in the areas of protocol verification, network authentication, compiler technology, and audio compression. Currently, his chief responsibility is the Inferno operating system. He also conducts research on programming languages and compilers, as well as on audio and video algorithms. Mr. Dorward received a B.S. degree in computer science from Princeton University in New Jersey.*

*ROB PIKE is a distinguished member of technical staff in the Computing Sciences Research Department at Bell Labs in Murray Hill, New Jersey. In 1981, he wrote the first bitmap window system for UNIX and has since written ten additional systems. Mr. Pike was a principal designer and implementer of both the Plan 9 and Inferno operating systems. In addition, he designed a gamma-ray telescope, co-designed the Bilt terminal, and co-authored* The UNIX Programming Environment. *He has never written a program that uses cursor addressing.*

*DAVID LEO PRESOTTO is a member of technical staff in the Computing Structures Research Department at Bell Labs in Murray Hill, New Jersey. He is responsible for research into and development of a technology known as electronic glue. Mr. Presotto received a Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley.*

*DENNIS M. RITCHIE is head of the Systems Software Research Department at Bell Labs in Murray Hill, New Jersey. He joined Bell Labs after receiving graduate and undergraduate degrees from Harvard University in Cambridge, Massachusetts. He is a co-developer of the UNIX operating system and is the primary designer of C language in which UNIX and many other systems are written. A Bell Labs Fellow and a member of the U. S. National Academy of Engineering, Mr. Ritchie has received several other honors, including the ACM Turing award, the IEEE Piore, Hamming, and Pioneer awards, and the NEC C&C Foundation award. He continues to work in the areas of operating systems and languages.*

*HOWARD W. TRICKEY is a member of technical staff in the Computing Architectures Research Department at Bell Labs in Murray Hill, New Jersey. His work involves research and development of the Inferno operating system. Mr. Trickey holds a B.A.Sc. degree in science and an M.A.Sc. degree in electrical engineering from the University of Toronto in Canada, and a Ph.D. in computer science from Stanford University in Palo Alto, California.*

*PHILIP WINTERBOTTOM is a member of technical staff in the Computing Sciences Research Department at Bell Labs in Murray Hill, New Jersey. He works in the areas of compilers, languages, operating systems, and networking hardware. Before coming to Bell Labs, he attended Kings College in London, England, then continued on to the City University of London where he was a Lloyds Research Fellow building parallel computers.* ◆