The Inferno Operating System

Sean Dorward
Rob Pike
David Leo Presotto
Dennis M. Ritchie
Howard Trickey
Phil Winterbottom

Computing Science Research Center
Lucent Technologies, Bell Labs
Murray Hill, New Jersey
USA

ABSTRACT

Inferno is an operating system for creating and supporting distributed services. It was originally developed by the Computing Science Research Center of Bell Labs, the R&D arm of Lucent Technologies, and further developed by other groups in Lucent.

Inferno was designed specifically as a commercial product, both for licensing in the marketplace and for use within new Lucent offerings. It encapsulates many years of Bell Labs research in operating systems, languages, on-the-fly compilers, graphics, security, networking and portability.

Introduction

Inferno is intended 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.

The most visible new environments involve cable television, direct satellite broadcast, the Internet, and other networks. As the entertainment, telecommunications, and computing industries converge and interconnect, a variety of public data networks 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, economical operating system can insulate the various 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 its portability and versatility across several dimensions:

- Portability across processors: it currently runs on Intel, Sparc, MIPS, ARM, HP-PA, and PowerPC architectures and is readily portable to others.
- Portability across environments: it runs as a stand-alone operating system on small terminals, and also as a user application under Windows NT, Windows 95, Unix (Irix,

Originally appeared in the *Bell Labs Technical Journal*, Vol. 2, No. 1, Winter 1997, pp. 5-18.
Minor revisions have been made by Vita Nuova to reflect subsequent changes to Inferno.

Solaris, FreeBSD, Linux, AIX, HP/UX) and Plan 9. In all of these environments, Inferno applications see an identical interface.

- Distributed design: the identical environment is established at the users terminal and at the server, and each may import the resources (for example, the attached I/O devices or networks) of the other. 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: it runs useful applications stand-alone on machines with as little as 1 MB of memory, and does not require memory-mapping hardware.
- Portable applications: Inferno applications are written in the type-safe language Limbo, whose binary representation is identical over all platforms.
- Dynamic adaptability: applications may, depending on the hardware or other resources available, 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 providers are interested in purveying media and services: telephone network service providers, WWW servers, cable companies, merchants, various information providers. There are many connection technologies: ordinary telephone modems, ISDN, ATM, the Internet, analog broadcast or cable TV, cable modems, digital video on demand, and other interactive TV systems.

Applications more clearly related to Lucents 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 controls an IP switch/router for voice and data being developed by Lucents Bell Labs research and Network Systems organizations. An Inferno-based firewall (Signet) is being used to secure outside access to the Research Internet connection.

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

Inferno is parsimonious enough in its resource requirements to support interesting applications on todays 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: 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, but 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, such as 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 resources such 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.

Most typically, an Inferno-based service would consist of many relatively cheap terminals running Inferno as a native system, and a smaller number of large machines running Inferno 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 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 is not one-way: the users terminal (whether a network computer, TV set-top, 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 [1] 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 together into a single private hierarchical *name space*.
- A communication protocol, called *Styx*, is applied uniformly to access these resources, whether 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 a modem, an MPEG decoder, a network interface, or the TV screen) attached to a particular hardware box present themselves as small directories. These directories typically containing two files, data and ctl, which respectively perform actual device input/output and control operations. System services also live behind file names. For example, an Internet domain name server might be attached to an agreed-upon name (say /net/dns); after writing to this file 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. Thus, 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 TCP/IP, PPP, ATM or various modem transport protocols.

Internal Environment of Inferno Applications

Inferno applications are written in a new language called Limbo [2], which 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, strings, dynamic arrays, and simple abstract data types.

In addition, Limbo supplies several advanced constructs 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 easy to use and safe.

Limbo programs are built of *modules*, which are self-contained units with 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 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, and also 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 unavailable, the video-decoder module is never loaded.

Limbo is fully type-checked at 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 unused. For example, if an application task creates a graphics window and then terminates, the window automatically disappears the instant the last reference to it has gone 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 3-address machine, supplemented with a few specialized operations for handling some of the higher-level data types like arrays and strings. Garbage collection is handled below the level of the machine language; the scheduling of tasks is similarly hidden. When loaded into memory for execution, the byte-codes are expanded into a format more efficient for execution; there is also an optional on-the-fly compiler that turns a Dis instruction stream into native machine instructions for the appropriate real hardware. This can be done efficiently because Dis instructions match well with the instruction-set architecture of todays machines. The resulting code executes at a speed approaching that of compiled C.

Underlying Dis is the Inferno kernel, which contains 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) as well as the small file systems implemented internally.

Finally, the Inferno virtual machine implements several standard modules internally. These include Sys , which provides system calls and a small library of useful routines (e.g. creation of network connections, 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 of 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 [18], with a Limbo interface. A Math module encapsulates the procedures for numerical programming.

The 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. Depending on the environment in which Inferno itself is implemented, there are several versions of the Inferno kernel, Dis/Limbo interpreter, and device driver set.

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 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. For example, under Unix, the graphics library might be implemented using the X window system and the networking using the socket interface; under Windows, it uses the native Windows graphics and Winsock calls.

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

Security in Inferno

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 set up, the encryption is transparent to the application. Key exchange is provided through standard public-key mechanisms; after key exchange, message digesting and line encryption likewise 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, to communication resources, and to 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 is controlled and is an authenticated operation. 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, and these signatures may be checked by the system the modules are accessed.

Although Inferno provides a rich variety of authentication and security mechanisms, as detailed below, few application programs need to be aware of them or explicitly include coding 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 a part of the application 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 that 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 provided by Inferno include the SHA, MD4, and MD5 secure hashes; Elgamal public key signatures and signature verification [4]; RC4 encryption; DES encryption; and public key exchange based on the Diffie-Hellman scheme. The public key signatures use keys with moduli up to 4096 bits, 512 bits by default.

There is no generally accepted national or international authority 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. Thus an organization using Inferno (or any other security and key-distribution scheme) must design its system to suit its own needs, and in particular 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 certifying authority that signs a users 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 certifying authority (CA). The signature covers a secure hash (SHA, MD4, or MD5) of the name of the party, his 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 Station to Station protocol [5] to establish the identities of the two parties and to create a mutually known secret. This STS protocol uses the Diffie-Hellman algorithm [6] 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 having the parties exchange certificates and then digitally signing key parts of the protocol. To masquerade as another party an attacker would have to be able to forge that partys 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 can be neither 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 DES with either DES chain block coding (DESCBC) and electronic code book (DESECB).

Inferno uses the same encapsulation format as Netscapes Secure Sockets Layer [7]. 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 strength of the random numbers used for choosing keys, Diffie-Hellman parameters, initialization vectors, etc. Inferno achieves this in two steps: a slow (100 to 200 bit per second) random bit stream 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

Limbo is the application programming language for the Inferno operating system. 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, networked Inferno environment: a rich set of basic types, strong typing, garbage collection, concurrency, communications, and modules. Limbo may be interpreted or compiled just in time for efficient, portable execution.

This paper introduces the language by studying an example of a complete, 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 [14] program `view`, which displays graphical image files on the screen, one per window. This version sacrifices some functionality, generality, and error-checking but performs the basic job. The files may be in either GIF [12,13] or JPEG [19] format and must be converted before display, or they may already be 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 it. Each window is given a title bar across the top to identify it and hold the buttons to move and delete the window.

The Source

Here is the complete Limbo source for our version of `view`, 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, it's worth skimming before moving to the next section, to get an idea of the style of the language. Control syntax derives from C [11], while declaration syntax comes from the Pascal family of languages [17]. Limbo borrows features from a number of languages (e.g., tuples on lines 45 and 48) and introduces a few new ones (e.g., explicit module loading on lines 90 and 92).

```
 1  implement View;
 2  include "sys.m";
 3      sys: Sys;
 4  include "draw.m";
 5      draw: Draw;
 6      Rect, Display, Image: import draw;
 7  include "bufio.m";
 8  include "imagefile.m";
 9  include "tk.m";
10      tk: Tk;
11  include "wmlib.m";
12      wmlib: Wmlib;
13  include "string.m";
14      str: String;
15  View: module
16  {
17      init: fn(ctxt: ref Draw->Context,
                 argv: list of string);
18  };
19  init(ctxt: ref Draw->Context,
         argv: list of string)
20  {
21      sys   = load Sys Sys->PATH;
22      draw  = load Draw Draw->PATH;
23      tk    = load Tk Tk->PATH;
24      wmlib = load Wmlib Wmlib->PATH;
25      str   = load String String->PATH;
26      wmlib->init();
27      imageremap := load Imageremap
                          Imageremap->PATH;
28      bufio := load Bufio Bufio->PATH;
29      argv = tl argv;
30      if(argv != nil
          && str->prefix("-x", hd argv))
31          argv = tl argv;
32      viewer := 0;
33      while(argv != nil){
34          file := hd argv;
35          argv = tl argv;
36          im := ctxt.display.open(file);
37          if(im == nil){
38              idec := filetype(file);
39              if(idec == nil)
40                  continue;
41              fd := bufio->open(file,
                              Bufio->OREAD);
42              if(fd == nil)
43                  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));
    (nil, file) = str->splitr(file, "/");
    (t, menubut) := wmlib->titlebar(ctxt.screen,
          " -x "+corner+" -y "+corner+
          " -bd 2 -relief raised",
           "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 .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));
}
```

```
85  filetype(file: string): RImagefile
86  {
87      if(len file>4
          && file[len file-4:]==".gif")
88          r := load RImagefile
                      RImagefile->READGIFPATH;
89      if(len file>4
          && file[len file-4:]==".jpg")
90          r = load RImagefile
                      RImagefile->READJPGPATH;
91      return r;
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; here line 1 states this file implements a module called View, whose declaration appears in the module declaration on lines 15–18. 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 only 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 all that is required for the module to be callable from the shell; all programs are constructed from modules, and some modules are directly callable by the shell 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.

Not all modules, of course, 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 2–14. Each module's interface is stored in a public include file that holds a definition of a module much like lines 15–18 of the View program. For example, here 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 defines a module type, called `Sys`, that has functions with familiar names like `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 the module. Line 3, for example, 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 the implementation of these types is by default 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. Also, it allows easy garbage collection of unused modules and allows 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 21-25 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 file name 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, but add a new wrinkle: they also *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 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) the remainder after the head, and `len` (length) the number of elements in the list.

Line 29 throws away the first element of argv, which is conventionally the name of the program being invoked by the shell, and lines 30–31 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 file name 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[20] 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 in the standard Inferno image format. If it fails, it will return nil and lines 38–50 will attempt to convert the file into the right form.

Decoding the file

Line 38 calls filetype to determine what format the file has. The simple version here, on lines 85–92, 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 looked at, 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. Since 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 file name and returns the implementation of a module to decode it.

A couple of other points about filetype. 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 file name. 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 [21] throughout) in the string.

Second, and more important, this version of filetype loads the decoder module anew every time it is called, which is clearly inefficient. It is 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.

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*: a pair of values. The first element of the pair is the image, while the second is an error string. If all goes well, the `err` will be `nil`; if there is a problem, 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 which 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 colormap associated with the file to the standard Inferno colormap. The result is an image ready to be displayed.

Creating a process

By line 52 in the main loop, 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 arguments the graphics context, the image to display, the file name, and a unique identification number used in placing the windows.

The new process shares with the calling process all variables except the stack. Shared memory can therefore be used to communicate between them; for synchronization, a more sophisticated mechanism is needed, a subject we will cover in the section on communications.

Starting Tk

The function `view` uses the Inferno Tk graphics toolkit (a re-implementation for Limbo of Ousterhout's Tcl/Tk toolkit [18]) to place the image on the screen in a new window. 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 is similar to the use 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 77-84. The calls to `tk->cmd` evaluate the textual command in the context defined by the `Tk->Toplevel` variable t (created on line 57 and passed to posn); the result is a decimal integer, converted to binary by the explicit int conversion. On line 83, 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 Toplevel. This tuple is automatically promoted to the Rect type by the return statement.

Back in function view, line 58 uses a function from the higher-level String module to strip off the basename of the file name, 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 58 calls the window manager function wmlib->titlebar to establish a title bar on the window. The arguments are ctxt.screen, a data structure representing the window stack on the frame buffer, a string specifying the size and properties of the new window, the window s label, and the set of control buttons required. The + operator on strings performs concatenation. The window is labelled "View" and the file basename, with a control button to hide the window. Titlebars always include a control button to dismiss the window. (The size and properties argument is more commonly nil or the empty string, leaving the choice of position and style to the window manager.) The first value in the tuple returned by wmlib->titlebar is a reference to a top-level widget a window upon which the program will assemble its display.

Communications

The second value in the tuple returned 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 Hoare s CSP [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*: both processes must be ready to communicate before the data changes hands, and if one is not ready the other blocks until it is. Channels are a feature of the Limbo language: they have a declared type (chan of int, chan of list of string, etc.) and only data of the correct type may be sent. There is no restriction on what may be sent; one may even send a channel on a channel. Channels therefore 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 on line 69,

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

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 (e.g. channel<-=2). This use of menubut appears inside an alt (alternation) statement, a construct we ll discuss later.

Lines 60 and 61 create and register a new channel, event, to be used by the Tk module to report user interface events. Lines 62-66 use simple Tk operations to make the window in which the image may be drawn. Lines 63 and 64 bind events within this window to messages to be sent on the channel event. For example, line 63 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 67, 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 our scope here. 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 may 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 must each 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 can proceed, it chooses one randomly.

Thus the loop on lines 68-75 processes messages received by the two classes of actions. When the window is moved or resized, line 73 will receive a "resize" message due to the bindings on lines 63 and 64. The message is discarded but the action of receiving it triggers the repainting of the newly placed window on line 74. Similarly, messages triggered by buttons on the title bar send a message on menubut , and the value of that is 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 71 will return from view . Since view was the top-level function in this process, the process will exit, freeing all its resources. All memory, open file descriptors, windows, and other resources held by the process will be garbage collected when the return executes.

The Limbo garbage collector [16] uses a hybrid scheme that combines reference counting to reclaim memory the instant its last reference disappears with a real-time sweeping algorithm that 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 become unused; there is no pause until a sweeper discovers it. This property allows Inferno to run in smaller memory arenas than are required for efficient mark-and-sweep algorithms, as well as providing 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 Lacy et al.[22].

References

1. R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs, *J. Computing Systems* 8:3, Summer 1995, pp. 221-254.
2. S. Dorward, R. Pike, and P. Winterbottom. Programming in Limbo, *IEEE Compcon 97 Proceedings*, 1997.
3. J.K. Ousterhout. *Tcl and the Tk Toolkit*, Addison-Wesley, 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, 1985, pp. 10-18
5. B. Schneier, Applied Cryptography, Wiley, 1996, p. 516
6. D. Stinson, Cryptography, Theory and Practice, *CRC Press*, 1996, p. 271
7. K. Hickman and T. Elgamal, The SSL Protocol (V3.0), *IETF Internet-draft*
8. S. M. Bellovin and M. Merritt, Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attack, Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy, 1992, pp. 72-84
9. M. Blaze, J. Feigenbaum, J. Lacy, Decentralized Trust Management, *Proceedings 1996 IEEE Symposium on Security and Privacy*, May 1996
10. R. Rivest and B. Lampson, SDSI - A Simple Distributed Security Architecture, unpublished, *http://theory.lcs.mit.edu/~rivest/sdsi10.ps*
11. *American National Standard for Information Systems Programming Language C*, American National Standards Institute, X3.159-1989.
12. *GIF Graphics Interchange Format: A standard defining a mechanism for the storage and transmission of bitmap-based graphics information*, CompuServe Incorporated, Columbus, OH, 1987.
13. *GIF Graphics Interchange Format: Version 89a*, CompuServe Incorporated, Columbus, OH, 1990.
14. S. Dorward et al., Inferno, *IEEE Compcon 97 Proceedings*, 1997.
15. C. A. R. Hoare, Communicating Sequential Processes. *Comm. ACM* 21:8, pp. 666-677, 1978.
16. L. Huelsbergen, and P. Winterbottom, Very Concurrent Mark & Sweep Garbage Collection without Fine-Grain Synchronization, Submitted *International Conference of Functional Programming*, Amsterdam, 1997.
17. K. Jensen, and N. Wirth, *Pascal User Manual and Report*. Springer-Verlag, 1974.
18. John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
19. W. B. Pennebaker and J.L. Mitchell, *JPEG Still Image Data Compression*, Van Nostrand Reinhold, New York, 1992.
20. R.W. Scheifler, J. Gettys, and R. Newman, *X Window System*, Digital Press, 1988.
21. The Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison Wesley, 1996.
22. J.B. Lacy, D. P. Mitchell, and W. M. Schell, CryptoLib: Cryptography in Software, *UNIX Security Symposium IV Proceedings*, USENIX Association, 1993 pp. 1-17.