inferno®

Inferno: An Overview

OPERATING SYSTEM FOR NETWORK APPLICATION DEVELOPMENT

Introduction

Inferno is an operating system for creating and supporting distributed services and other networked applications. It was originally developed by the Computing Science Research Centre of Bell Labs, the R&D arm of Lucent Technologies and has been substantially further developed by Vita Nuova and Lucent.

Inferno encapsulates many years of Bell Labs research in operating systems, languages, on-the-fly compilers, graphics, security, networking and portability. It is intended to be used in a variety of network environments: home, office and mobile.

Inferno's definitive strength lies in its portability and versatility across several dimensions:

- Portability across processors: it currently runs on Intel, SPARC, MIPS, PowerPC and ARM (includingThumb). Work is currently underway on SH3/4.

- Portability across environments: it runs as a native operating system on small devices, and also as a user application under Windows, Linux and UNIX. In all these environments Inferno applications see an identical interface.

- Distributed design: the identical environment is established on each device, and each may import the resources of the other platforms.

- Dynamic adaptability: applications may, depending on the hardware or other resources available, load diferent program modules to perform a specific function, for example, a video player application might use any of several different decoder modules.

- Portable Applications: Inferno applications are written in the type-safe language Limbo, whose compiled representation is identical over all platforms.

Portability Across Processors

The Inferno kernel and device drivers are all written in C. The system comes with a cross compiler suite that was developed at Bell Labs. The compiler suite is extremely compact and yet still manages to produce code that is comparable in terms of size and efficiency with much larger compiler packages.The compiler for a particular architecture is typically around 12,000 lines of code.

Most of the common processor architectures are supported by the compiler including: StrongARM, x86, MIPS and PowerPC.

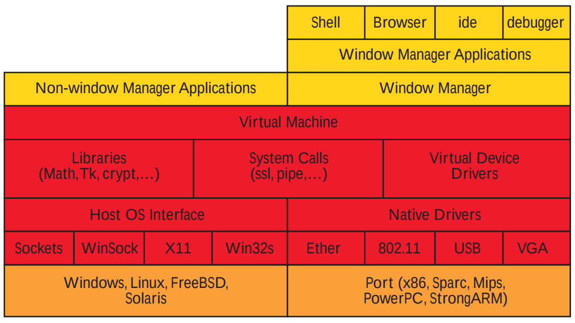
For ARM chips the compiler can generate either ARM or Thumb code. The compiler suite partitions the work between compiler and linker in a novel way that places more emphasis on optimization at the linking stage. In the case of the ARM, the linker can link mixed ARM and Thumb code into a single binary enabling the developer to choose between efficiency of size or performance for different components.

Porting Inferno to a new device based upon one of the above architectures is relatively simple compared to many other operating systems.The Inferno kernel has been designed such that there is a well defined interface between those parts of the system that are 'platform specific' and those that are 'platform independent'.The interface is small.

The 'platform specific' elements are essentially the interfaces (drivers) to the hardware. Device drivers themselves consist of a platform-specific and platform-independent part. The latter component resides within the portable kernel code and provides a uniform representation of the device to the Inferno applications above. For example, the 'draw-device' provides a device dependent interface to the display hardware.The non-portable part of the driver is responsible for the control of the physical device.

Having provided support for the hardware devices, the compilation of the remainder of the kernel is straightforward.

The block diagram below shows the relationship between the various components of the Inferno system and the physical hardware.


Shell
Browser
ide
debugger
Window Manager Applications
Non-window Manager Applications
Window Manager
Virtual Machine
Libraries (Math,Tk, crypt,…)
System Calls (ssl, pipe,…)
Virtual Device Drivers
Host OS Interface
Native Drivers
Sockets
WinSock
X11
Win32s
Ether
802.11
USB
VGA
Windows, Linux, FreeBSD, Solaris
Port (x86, Sparc, Mips, PowerPC, StrongARM)

Portability Across Environments

The traditional perception of an operating system is that it will take control of the whole of a computing device; this is true for most systems including Windows, Linux and most RTOSs. More recently, operating system developers have come to recognize that, whilst there are obvious advantages in providing new environments (particularly those that address distributed application development), it is not practical to replace existing operating systems in all cases. Inferno addresses this problem by providing 'hosted' versions of Inferno that make use of the services of an existing operating system in order to support the Inferno environment. As with all native Inferno ports, hosted versions of Inferno provide an identical environment to the application developer.
Not all operating systems can act as a host for Inferno, a minimum level of functionality must be provided. There is no Inferno port to DOS, for example, but there are versions for Windows 95/98/2000/NT, Linux, Solaris, Irix, FreeBSD, Plan 9 and others.

Where Inferno is hosted on top of an existing operating system, one can consider Inferno to be a portable application development environment for the construction of distributed applications.

Distributed Design

One of the most difficult tasks facing programmers today is the design and development of distributed applications that will run across a heterogeneous network of computing devices. The difficulty of writing such applications is manifested in a number of ways:

- The development environments and programming technologies vary greatly from one device to another. Not all networks are IP, not all devices are on the Internet.

- Existing systems vary dramatically in the way they present their resources. Some devices may provide access through a specialized low level protocol (a digital camera say), others through remote procedure calls (RPC), and others through high level protocols not originally intended for the task (HTTP for example).

- Differing policies for security and authentication make claims of reliability hard to make.

Inferno addresses the first of these points by providing the same environment everywhere within the network, whether on an existing system in hosted mode or native on a new device.

The second issue Inferno addresses through a simple, unifying mechanism for the representation of resources in the network. This mechanism is based upon the import and export of a hierarchical 'namespace'. A namespace looks like a hierarchical filesystem, but it isn't. The names within the namespace can be accessed as if they were files using the file operations 'open', 'close', 'read' and 'write'. Namespaces can be composed into more complex hierarchies representing a collection of resources. Applications that operate on these namespaces are insulated from whether resources are, local or remote.

The import and export of namespaces is underpinned by a single, unifying file protocol called Styx. The Styx protocol is used for access to all resources whether local or remote. Styx can run over a variety of transport protocols (TCP/IP, ATM) and insulates Inferno applications from the type of network being used. Styx can run over simple link-protocols that connect small devices one to another, for instance, a Lego IR link.

Finally, since access to all resources is through the Styx protocol, Inferno has a single point at which to focus security. As a consequence security is an issue dealt with by the operating system and not by individual applications, as is so often the case.

Namespace Example

This example describes a home network that consists of the following devices:

- Windows PC
- Digital Camera
- Video Recorder
- PDA (Compaq iPAQ, for example)

The physical connection between the devices may be as follows:

- Wireless 802.11b network connecting the iPAQ and the Windows PC
- Ethernet network connecting the PC to the Video Recorder
- USB connection from the PC to the Digital Camera

Inferno is represented on these devices as follows:

Windows PC

Hosted version of Inferno providing an environment for distributed application development. The application that controls the other devices will run from this machine though does not have to.

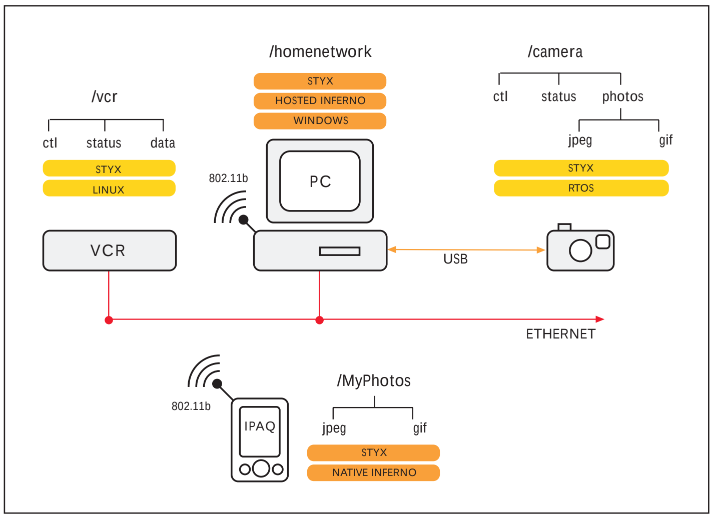
iPAQ

Native version of Inferno that controls the entire device including a driver for a PCMCIA Wireless card. The iPAQ contains a collection of useful PDA style applications and also a simple JPEG viewer for the user to look at their latest snaps.

Video Recorder and Digital Camera

Both of these devices could run Inferno as a native operating system, however, to futher illustrate the flexibility of Inferno we will assume that the Video Recorder is running a version of Linux and the Digital Camera an RTOS. Instead of replacing these incumbent operating systems we assume that a Styx protocol stack has been ported to each and operates as a process inside each of the hosted operating systems.

The diagram below illustrates this simple network, detailing the namespace presented by each device.



The Application - Time Lapse Photography

The task we give ourselves is to write a distributed application to do time lapse photography. The photographs are to be taken at 10 second intervals. After each photograph is taken it is to be copied to a VCR for playback as a video. Each image is to be stored for posterity in some local store either on the hard disk or on the iPAQ.

Using other technologies this modestly complex application would require some effort to write. The programmer would have to deal with potentially four different development environments using different APIs and conventions on each. Furthermore, it would be difficult to insulate the application from where the resources it is using are located. With Inferno, however, this modest application could be implemented with virtually no programming at all. How is that done?

What follows is a step-by-step example using commands but, obviously, graphics would be used in a consumer interface.

Step One - mount the different name spaces

The 'namespace' for each device is first of all mounted under a mount point on the machine that is to run the application (in our case the PC). The mount point is `/n/remote`. Each mount specifies the type of network (TCP in this case), the IP address and the port. The network type can be left out, and in these examples the port is left off, defaulting to 'Styx' of course! At each of the IP addresses and ports specified there must be a process that serves the Styx protocol.

```
mount tcp!182.1.1.2 /n/remote/vcr
mount tcp!182.1.1.3 /n/remote/camera
```

Step Two - bind the namespaces together

These name spaces are then bound into the name space that our application expects to see `/homenetwork`.

```
bind -a /n/remote/vcr /homenetwork/vcr
bind -a /n/remote/camera /homenetwork/camera
bind -a '#Uc:/MyPhotos'/homenetwork/MyPhotos
```

The last bind command binds part of the Windows files system into the `/homenetwork` namespace. The convention is that the # introduces a reference to a local device, in this case the U specifies the host OS file system and the remainder of the text indicates a path within the host file system.

Step Three - run the application from the PC

In fact lets write the application using a shell script.

```
echo 'record single frame' > /homenetwork/vcr/ctl
echo 'picture type jpg' > /homenetwork/camera/ctl
while : ; do
        echo 'snap' > /homenetwork/camera/ctl
        photo='cat /homenetwork/status'
        cp /homenetwork/camera/photos/$photo.jpg /homenetwork/MyPhotos
        cp /homenetwork/camera/photos/$photo.jpg /homenetwork/vcr/data
        echo 'next frame' > /homenetwork/vcr/ctl
        echo 'delete $photo' > /homenetwork/camera/ctl
        sleep 10
done
echo 'record off' > /homenetwork/vcr/ctl
echo 'rewind' > /homenetwork/vcr/ctl
```

This script initializes the camera and VCR and then it enters a loop that involves taking a photo, copying it to the local store and the VCR and then pausing for 10 seconds before repeating the process again.

What you notice from this script is that the application operates on the namespace as if it were a collection of files. As far as the application is concerned the namespace is indistinguishable from physical files on the local machine. What you will also notice is that commands are written as text into these files, for example:

```
echo ‘snap’ > /homenetwork/camera/ctl
```

The commands do not have to be text but there are advantages:
- text is easy to read
- scripting can be used to implement programs

The camera and VCR need to interpret the commands sent and to interact with the physical hardware, but this would be true no matter what scheme were used. The advantage of the Inferno architecture is that the commands to control the device are separated from the protocol for communication. Extensions to the command set to control the device do not require modifications to the Styx communication protocol.

Step 4 - run the application from the IPAQ

This is a trivial change for Inferno. In fact the application remains exactly the same whilst the initial mount and binds are modified as follows:

```
mount 182.1.1.2 /n/remote/vcr
mount 182.1.1.3 /n/remote/camera
mount 182.1.1.3 /n/remote/ipaq
bind -a /n/remote/vcr /homenetwork/vcr
bind -a /n/remote/camera /homenetwork/camera
bind -a /n/remote/ipaq/MyPhotos /homenetwork/MyPhotos
```

As long as the application is presented with the same namespace it will operate on it regardless of where the resources are located.

Step 5 - run on the iPAQ store on PC

Again there are no code changes at all in the application, it does not even have to be recompiled. The namespace on the iPAQ will be composed as follows with the `/MyPhotos` directory coming from the PC this time.

```
mount 182.1.1.2 /n/remote/vcr
mount 182.1.1.3 /n/remote/camera
mount 182.1.1.4 /n/remote/pc

bind -a /n/remote/vcr /homenetwork/vcr
bind -a /n/remote/camera /homenetwork/camera
bind -a /n/remote/pc/MyPhotos /homenetwork/MyPhotos
```

Summary

Great simplicity is gained by applying the Inferno namespace metaphor consistently and aggressively to all resources in the network. In doing so many things that in other systems would be hard to achieve become trivial. Here are a further two, short examples:

Remote debugging

The Inferno graphical debugger allows the developer to debug a thread by opening the files located in the directory `/prog/<pid>` where `<pid>` is the process id for the thread. To debug a process on another device is trivial, all the user need do is mount the namespace of the remote device and then bind the remote `/prog` directory in place of the existing `/prog` directory. Here are the commands to use:

```
mount 182.1.1.3 /n/remote
bind  /n/remote/prog /prog
```

The graphical debugger is blissfully unaware of whether the thread to debug is local or remote.

Porting Inferno

Porting any operating system can be a laborious task as one incrementally adds support for the various hardware devices. It is not until one implements a driver for the screen, keyboard and pointer can one begin to interact with the device.

With Inferno one can reach this position much quicker. Once the basic CPU and network support has been completed other resources can be imported over the network. For example, you could import the screen, keyboard and pointer device from another machine:

```
mount 182.1.1.3 /n/remote
bind  /n/remote/dev/keyboard /dev/keboard
bind  /n/remote/dev/draw /dev/draw
bind  /n/remote/dev/pointer /dev/pointer
```

Having done so the Inferno kernel can run on the new device while the user interacts with it using the sceen, keyboard and mouse of any other device running Inferno in hosted or native mode.

Portable Applications

Inferno applications are written using the Limbo programming language. It is also possible to write scripts using a programmers shell. The Limbo compiler generates byte code (dis) which is interpreted by the Inferno virtual machine.
The byte code is exactly the same on all Inferno platforms and hence Limbo applications are absolutely portable, no buts, no ifs, across all native and hosted platforms.

The Limbo programming language is similar in many respects to C. C programmers will find the transition to Limbo a simple and painless process. And because Limbo is type-safe and includes automatic garbage collection the code is much easier to write, read and maintain than many other languages.

Limbo is a concurrent programming language, that is, it contains the constructs necessary to synchronize a collection of co-operating processes or threads, Inferno enables these threads to be executed on different devices and to communicate using Styx or the namespace metaphor. Not only does Limbo support the creation of these distributed applications it also provides the utilities to graphically debug them.

Comparison with Other Systems

There are a large number of operating systems for small devices and a smaller, but still significant, number of distribution protocols and an even smaller number of concurrent programming languages.

In this section Inferno is compared with products from each of these three categories. The products of many other companies overlap with the application of Inferno, however, no other technology is so complete in its scope. The table below summarizes the performance of other systems against a number of key features of distributed computing technologies:

	Erlang	Jini/Java	IBM MQSeries	XML/SOAP	General RTOS	Linux	Embedded Linux	Microsoft Net	Inferno	Corba
Virtual Machine	○	✓	○	○	○	○	○	✓	✓	○
C/C++Language Support	○	✓	✓	○	✓	✓	✓	✓	✓[1]	✓
Concurrent Programming Language	✓	✓	○	○	○	○	○	○	✓	○
Protocol for Distribution	○	○	✓	✓	○	○	○	○	✓	✓
Native OS	○	○	○	○	✓	✓	✓	✓	✓	○
Hosted Environment	✓	✓	✓	✓	○	○	○	○	✓	✓
Embedded Environment	○	✓[2]	○	○	✓	○	✓	○	✓	✓
Royalty Free Distribution	○	○	○	○	○	✓	✓[3]	✓	✓	✓
Source Code Visibility	✓	○	○	○	○	✓	✓	○	✓	○

1 Built-in modules and device drivers are written in C, but concurrent applications are usually implemented in Limbo

2 In practice embedded Java environments require significant resources. The European STB standard which uses Java has a minimum specification of 16Mb RAM.

3 Many embedded Linux offerings charge per-piece royalty fees for the OS and do not give access to the full source code.

www.vitanuova.com

vita nuova®