A Loadable Kernel Module Adaptor for the Virtual Choices Operating System

Tanya L. Crenshaw tcrensha@uiuc.edu
Archana Dutta dutta2@uiuc.edu
Scott Kircher kircher@uiuc.edu
Deepu Chandy Thomas dthomas1@uiuc.edu

May 6, 2004

Abstract

The Linux open source community has done well to create a wide array of drivers to support the multitude of devices available on the market. Virtual Choices, a research OS currently lacking in device driver support, would benefit from such a source base by utilizing a device driver adapter. We present an adapter, LKMA, which creates a translation of the Linux module into a module that can be loaded by our newly developed `ELFLoader` *in Virtual Choices. With this adapter, the Virtual Choices operating system can use Linux device drivers with minimal porting efforts by the system programmer.*

1 Introduction

One of the most tedious portions of operating system development is writing the numerous device drivers required by the wide variety of devices now availble in the market. Virtual Choices, a research operating system at the University of Illinois in Urbana-Champaign, has a limited number of these device drivers. Linux, on the other hand, is an open source operating system for which device drivers are being written and maintained by the computing population every day. So that Virtual Choices may take advantage of Linux's device driver source base, we offer a device driver adapter, Loadable Kernel Module Adaptor (LKMA).

The Virtual Choices Operating System is a research OS which demonstrates that an OS may be written almost entirely in an object-oriented fashion using the C++ programming language. As a result, everything in Virtual Choices is derived from an Object class. Moreover, Virtual Choices has a notion of a Dynamic Class in which classes can be loaded into the kernel at runtime. The Loadable Kernel Module (LKM) is somewhat analogous to the Dynamic Classes in that they too can be dynamically loaded into the kernel at runtime using the `insmod` command.

With these considerations in mind, the overall concept of the LKMA is shown in Figure 1. There are two ends, the Linux end and the Virtual Choices end. The Linux end simply consists of the LKMs, compiled on a native Sparc host, that will be adapted to choices. At the Virtual Choices end, taking advantage of the already existing infrastructure for the Dynamic classes, is a three-part design. The LKMs are adapted by load, link, and relocation methods into the Virtual Choices kernel. These adapted LKMs inherit from two classes. The first is the `LKMAdaptor` class which performs all of the common adaptations needed by all LKMs; `printf`, `memcopy`, and `bzero`, for example. The second is the derived child class that implements all of the driver specific adaptations. This class contains all of the adaptations a given category of device drivers–like Ethernet–need to interface to the Virtual Choices kernel.

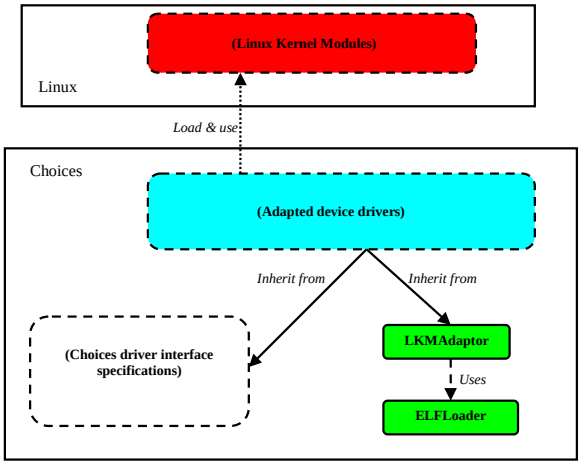

Figure 1. Fundamental LKMA Architecture

The rest of this document is organized as follows. Section 2 discusses the overall architecture of the LKMA. Section 3 details the ELF object format and the `ELFLoader` developed for the LKMA. Section 4 lists the other special considerations that had to be made in the development of LKMA. Section 5 describes the code added to the Virtual Choices source base and Section 6 provides an example of its use. Section 7 evaluates the success of the project and Section 8 describes the overall contributions made by LKMA. At the end of the document are biographies of each team member and his or her individual contributions to the development of LKMA.

2 Design

The LKMA architecture has two major parts. The first part is an ELF (Executable and Linking Format) loader and linker that is capable of reading compiled LKM modules in ELF object format and resolving both their defined

and undefined symbols against any provided symbol table. The second part is the `LKMAdaptor`, which encompasses the `LKMAdaptor` base class and all classes that inherit from it. This part is responsible for providing the actual adaptation between Linux calls and Virtual Choices calls. This section describes the architecture design of each of these parts.

2.0.1 Overview

Figure 2 depicts an overview of the LKMA architecture. Each relevant module (in Linux) or class (in Virtual Choices) is depicted as as a rounded box containing the name of the module or class. Relationships between entities are shown with labeled arrows. Virtual Choices uses a separate interface specification for each category of device. Thus, each category of device has a corresponding "adapted device driver" class that inherits both from the Virtual Choices interface specification and from `LKMAdaptor`. Through `LKMAdaptor`, the adapted device driver can then make use of an `ELFLoader` class to load, link, and relocate a loadable Loadable Kernel Module (LKM), to provide the actual implementation of the device driver. Both `LKMAdaptor` and its children can provide Linux kernel emulation functions, to provide the loaded LKM with the interface it expects from the Linux kernel. The adapted device drivers "implement" the driver interface that Virtual Choices expects, by making use of the functionality provided in the loaded LKM. A single adaptor class may be used with multiple LKMs, as long as sufficient Linux kernel emulation is provided. Ideally, there would need to be only one adaptor class for each Virtual Choices driver interface class, which would be able to make use of any LKM for that type of device.

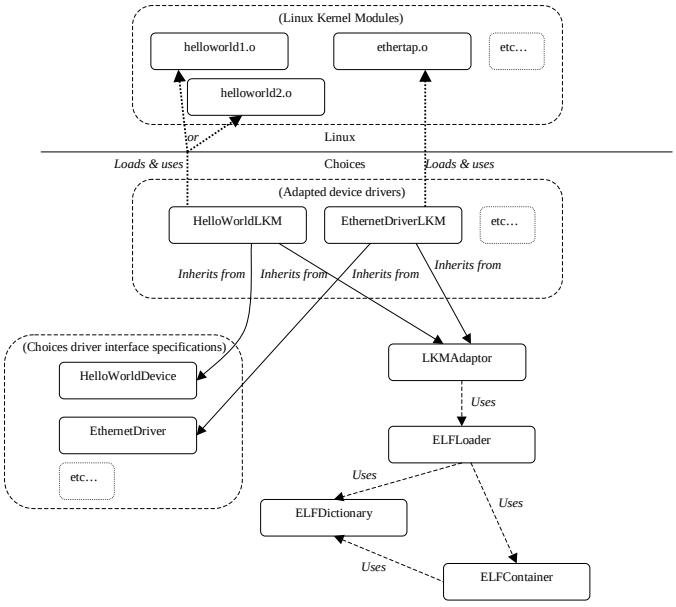

Figure 2. Detailed LKMA Architecture

Of the classes shown above, our project comprises:

- `ELFDictionary`,
- `ELFContainer`,
- `ELFLoader`,
- `LKMAdaptor`,
- `HelloWorldDevice`,
- `HelloWorldLKM`,
- `EthernetDriverLKM`.

We also had to make numerous changes to other parts of the Choices OS as discussed in the Section 4. These changes were necessary both to deal with legacy issues and to incorporate our `LKMAdaptor` architecture. Also, it should be noted that `HelloWorldDevice`, `HelloWorldLKM`, and `EthernetDriverLKM` are simply *examples* of using the `LKMAdaptor` architecture, rather than being a fundamental part of it.

2.0.2 Loading a Loadable Kernel Module

LKMs are generally compiled into ELF format. The first step in making use of an LKM is to load and link the ELF binary. This is similar to what occurs when an executable is linked from several source binary object files, except we perform this loading and linking at *run-time*, dynamically linking the ELF object into the Virtual Choices kernel. This is also, in fact, very much like what happens when an LKM is loaded into the Linux kernel, using insmod.

The Virtual Choices philosophy is that every file is treated itself as a file-system. Using this philosophy, the `ELFLoader` class makes use of `ELFContainer` and `ELFDictionary` to load the various sections of the ELF file. `ELFContainer` handles parsing the file and retrieving sections as they are requested by name. `ELFDictionary` provides a directory service into an ELF file. More details on these ELF classes are discussed in Section 3.

When an `ELFLoader` class is instantiated, its constructor must be passed the name of an ELF object file to load, as well as an export symbol table and an import symbol table. These symbol tables will be used to resolve undefined and defined symbols, respectively. From the ELF object file specified, `ELFLoader` reads the `text` (executable code), `data` (initialized data), `bss` (uninitialized data), `rodata` (non-string read-only data), and `rodata.str1.8` (string read-only data) sections. It also loads the relocation information for each of the above sections, as well as the symbol table and string table from the ELF file where the string table provides the names of the symbols. It then proceeds with the following steps:

1. Resolve undefined symbols.
2. Relocate symbols
3. Relocate references to symbols
4. Resolve defined symbols

Of the above steps only the fourth–resolve defined symbols–is specific to `ELFLoader`. The other three steps are part of any common link editor. We need the fourth step to allow adaptor classes the ability to call functions and make use of variables that are defined in the loaded ELF file. It should be noted that our `ELFLoader`, implementation

was heavily influenced by the existing Virtual Choices `COFFLoader` implementation which loads the old Common Object File Format object files. The four steps listed above are described below.

Resolving undefined symbols consists of searching through the provided export symbol table for names that match undefined symbols in the loaded ELF file's symbol table. When a match is found, the address provided by the export symbol table is placed in the loaded symbol table.

The next step is relocating symbols. Since the object file is now loaded into some location in memory, the section-offsets that specify the location of a symbol's data or function within the ELF file must be replaced with actual virtual memory addresses corresponding to where the symbol's data or function has been stored in memory. This is done easily by adding the memory location of the buffer that now holds the symbol's section to the symbol's location value.

After relocating symbols, any references to those symbols within any of the sections; especially the executable text section must be altered to reflect the new location of the symbol's data or function. This process is straightforward. One simply needs to follow the prescribed actions as specified by the loaded relocation info for each section. However reference relocation is processor-architecture dependent; our current implementation is for 32-bit Sun SPARC machines.

Finally, we resolve defined symbols. This is similar to resolving undefined symbols, except in reverse. Instead of placing the addresses of exported symbols into the loaded symbol table wherever there was an undefined symbol, we instead extract addresses from the loaded symbol table and place them into the imported symbol table wherever the symbol names match. This allows the adaptor classes to obtain the addresses of data and functions in the loaded ELF file.

At this point, the ELF file has been loaded into memory, and linked with the already running Virtual Choices kernel. It is now ready for use by a driver adaptor class.

2.0.3 Adapting a Loadable Kernel Module

The `LKMAdaptor` base class makes use of the `ELFLoader` class and provides the basic functionality of adapting Linux system calls into Virtual Choices system calls, as well as mapping Virtual Choices driver class member function calls into Loadable Kernel Module calls. At the most simplistic level, the problem is basically one of interface adaptation. Linux and Virtual Choices have different interfaces, and thus some adaptor needs to sit between them to translate.

Within the `LKMAdaptor` namespace exists a static array of "export tuples" of type `ELFLoader::ExportTuple`. This array, `LKMAdaptor::exported_symbols`, is simply a big table associating Linux system call symbol names with adaptor function pointers. For example `printk` is associated with the address of `adaptor_printk()`. Of course, every system call that an LKM can be expected to make to the Linux kernel must be implemented as an adaptor, which makes the corresponding calls in Virtual Choices. For example, `adaptor_printk()` calls `Console().formattedWrite()`.

It is not, however, necessary to place all the exported symbols in this global table. `LKMAdaptor` child classes can provide their own static adaptor functions for kernel calls that are specific to that kind of device. The global `exported_symbols` table is combined with any symbols a child class may wish to export. This aggregate `exported_symbols` table will be passed to an `ELFLoader` object, to provide the addresses to link undefined symbols in the loaded module against.

Exporting symbols that the loaded module expects is only part of the battle. It is also necessary to "import" symbols from the loaded module. That is, there are some functions provided in the loaded module that must be called by some other part of the operating system. If this were not the case, the module would be useless, because its code would never be executed. The canonical example of such a function is the standard `init_module()` function that every LKM is expected to provide. We also use this capability to "grab" and set LKM module parameters. Each class that inherits from `LKMAdaptor` may provide its own list of symbols it wishes to import. By default, the `init_module` and `cleanup_module` symbols will always be imported, as these are provided by every LKM.

Virtual Choices is, of course, an object-oriented operating system. As such, it expects all of its device drivers and services to be implemented as objects or collections of objects. Thus, to use a loaded LKM as a device driver or service in Virtual Choices, we must encapsulate the module within an object. Virtual Choices will talk to the "wrapper object" through the member function interface it understands, and the "wrapper object" will translate those invocations into calls into the loaded module code. This "wrapper object" is nothing more than an instance of a class that inherits from `LKMAdaptor` and expands it by providing the interface that Virtual Choices expects for the particular device or service in question. It also provides device specific Linux emulation routines that the loaded LKM expects and that are not already implemented in `LKMAdaptor`.

Generally, device drivers for a particular class of device are expected to have a certain interface. Depending on the details of the device, the implementation will vary, but the interface is fixed, so that the OS can actually talk to the device driver. Of course, this is the whole point of a device driver, to present a known interface to an unknown device. In Virtual Choices, this is handled by having abstract base classes that represent different types of devices. These base classes provide an interface that is implemented in the `MachineDependent` code, via inherited classes. In our case, we wish to use a loaded Linux kernel module as the implementation, while providing the interface that Virtual Choices can understand. Thus, we need to create a class that inherits from both the base class for the device in question, as well as the `LKMAdaptor` class. Luckily, C++ provides a multiple inheritance mechanism used to accomplish this.

Note that Linux also expects a standard interface from a device driver, and as such, the loaded module will provide a standard interface. Thus, while we need to create a new adaptor for each type of device, we do NOT need to create adaptors for every LKM we wish to load. A single Ethernet-specialized `LKMAdaptor` should be sufficient to allow loading any Linux Ethernet driver for any corresponding Ethernet device. Since we only need to translate the interface at the

Linux-Virtual Choices boundary, we do not have to worry about the actual implementation within the LKM, or the actual device being driven. Simply, it is only necessary to categorize the device properly.

3 ELF Module Format and Parsing in Virtual Choices

As previously mentioned, a portion of the LKMA was devoted to an `ELFLoader`. This section discusses the details of parsing the ELF module format in Virtual Choices.

Linux modules have traditionally been Executable and Linkable Format (ELF) [11] dynamic modules. Virtual Choices already contains a COFF object file parser; however the fact that the COFF format is largely obsolete in the UNIX world led us to implement a parser that provides a consistent file system interface for every section in the ELF module. The ELF object file format is shown in Figure 3.

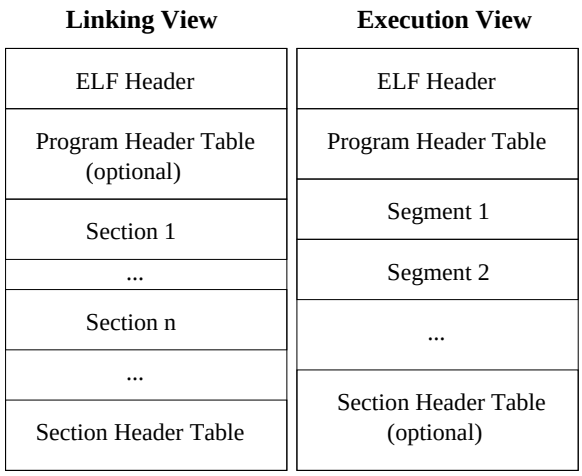

Figure 3. ELF Object File Format

Each ELF file is made up of one ELF header, followed by zero or more segments and zero or more sections. The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Each byte in the entire file is taken by no more than one section at a time, but there can be orphan bytes, which are not covered by a section. In the normal case of a UNIX executable one or more sections are enclosed in one segment. The segments and sections of the file are listed in a program header table and section header table respectively.

The program header table is absent in the case of Linux Kernel modules as they are relocatable in nature. Hence we derive most of the `ELFLoader` design from the section header table. Every section has an entry in the table; each entry gives information such as the section name, the section size etc. Each section in an ELF file can have an associated string table and symbol table in addition to the default string table the offset of which is addressable through the section header table. The default string table whose index is provided in the ELF header contains strings to represent section names. A given string is referenced as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise a string table's last byte is defined to hold a null character, ensuring null termination for all strings. The string table associated with the symbol table section is located by following the `sh_link` offset of that section which provides an index into section header. We currently support ELF-32 files though support for ELF-64 can be easily added.

Each section is presented as a file to the file system interface with well known id's provided for the Symbol Table, the default String Table, symbol String Table and the ELF header. The rest of the sections have the header, section information, and relocation information. A relocation section has the string `.rela` prepended to the name of the original section name.

The `ELFContainer` and the `ELFDictionary` class together implement the ELF file parsing in Virtual Choices. The `ELFContainer` class essentially provides matching of file id's or inodes to actual memory locations in the object file. The `ELFDictionary` class contains the default section names and provides function lookups that are in turn used up by the `ELFLoader` class for detailed querying.

4 Special Considerations

In addition to the LKMA itself, a number of considerations had to be made in this project. These included Virtual Choices legacy issues, updating the existing network interface, as well as device drivers considerations, both on the Linux side and the Virtual Choices side. These considerations are detailed in the sections below.

4.1 Virtual Choices Legacy Issues

Before actually adding the LKMA to Virtual Choices, a number of steps had to be taken to update the source base to something that was functional for present day compilers. The following provides a list of these issues but does no justice to the work involved in solving them.

1. **C++ legacy issues**: Many virtual functions were being hidden by type mismatches. No longer do `void function(unsigned int)` and `void function(int)` have the same type signature.
2. **Incorrectly implemented asynchronous closures**: `init` functions in the File System were not initializing the correct function at closure activation.
3. **Virtual Choices COFFLoader**: Virtual Choices original COFFLoader was used to load the Dynamic Classes in COFF format. This binary format used by Sparc had been replaced by the ELF format. An `ELFLoader` was developed based largely on the original COFFLoader design and is detailed in Section 3.
4. **Virtual Choices Ethernet Interface**: NIT was outdated, packet buffers no longer worked, and Virtual Choices had to be updated with the newer, more generic Data Link Packet Interface (DLPI). DLPI is addressed in further detail in Section 4.2.

4.2 Data Link Packet Interface

The DLPI API's [12] allow Ethernet packets to be sent and received from the user land level on Solaris 5.x. This is necessary for our Ethernet driver since it needs access to the underlying datalink interface.

The Data Link Provider Interface specifies a STREAMS kernel-level implementation of the ISO Data Link Service Definition (ISO 8886) and Logical Link Control (ISO 8802/2) LLC. DLPI enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol. Specifically, the interface is intended to support X.25 LAPB, BX.25 level2, SDLC, ISDNLAPD, Ethernet, FDDI, Token Ring, Token Bus, BISYNC, and other datalink-level protocols. The interface specifies access to data link service providers, and does not define a specific protocol or protocol implementation. The DLPI driver that we interface with is the eri Fast Ethernet driver a STREAMS-based hardware driver supporting the DLPI interface accessible via `/dev/eri`.

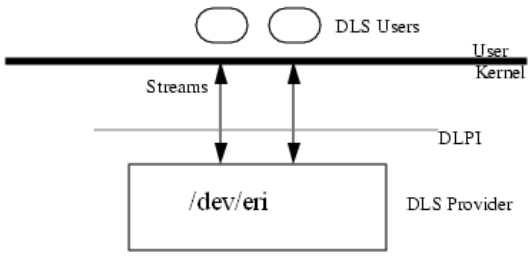

Figure 4. Data Link Packet Interface

We enable DLPI in the promiscuous mode to ensure that the Virtual Choices Ethernet driver also receives packets that not necessarily arrive for the host machine. This gives us flexibility to use a Virtual Choices specific MAC and IP Address. A user level daemon is currently used to arbitrate packet transfer for processes in the same host. This was done primarily to support sending of packets within the same host by multiple Virtual Choice instances. We also use a packet filter over DLPI in order to reduce the load on the number of packets handled by our interrupt handler/driver. The mode we operate in is defined as the "raw mode" as we directly receive the packet including the 14-byte Ethernet header. The packets that are sent out on the wire also need the correct 14-byte Ethernet header.

Packet reception

The EthernetLKM Driver waits for a signal to be received on the file descriptor associated with the DLPI Interface. The signal is verified to ensure that the received signal indicates a presence of a new data frame. The packet is read using DLPI calls, conversion into Linux friendly socket buffers is performed, and this socket buffer is passed on to the actual Linux interrupt routine which does packet processing on the data received. The Linux receive interrupt routine makes a call to it's upper layer which is emulated and a redirection is made to call the Virtual Choices `NetworkDriver::receive` function and completes the packet reception control flow.

Packet transmission

The `EthernetDriverLKM` presents a `TransmitPacket` function which is used by Virtual Choices to send a packet. In turn our driver sends the packet after performing the conversion routine detailed above to the Linux transmit routine. The Linux routines rely on the netlink API to transmit routines. We emulate the netlink API functions which call the DLPI transmit routines to send the packets. This was done since DLPI support is not provided in the UML Ethernet driver since UML has not been ported to Solaris.

4.3 Linux Device Drivers

Because the LKMA was added to Virtual Choices rather than its native Sparc counterpart, Virtual Choices, there was some difficulty in choosing the non-trivial drivers to adapt. For the solution, we turned to User-Mode Linux. Like Virtual Choices, User-Mode Linux is a user-level "kernel" which simply runs on top of Linux. This allows Linux developers a safe means to test portions of the kernel without fear of detrimentally affecting their actual Linux kernel executing on the native platform.

In a kernel, a driver is responsible for the following minimal set of functionality:

- **open**: turning on and initializing hardware,
- **write**: passing data to the hardware.
- **read**: extracting data from the hardware.
- **close**: turning off hardware.

However, in the user-level os model, the concept of a "driver" becomes something of a misnomer. At the user-level, there is no direct access to the hardware. Instead, a driver in a user-level kernel must simply interface with the lower-level kernel. Thus, in the selection of Linux drivers to adapt to Virtual-Choices, measures had to be taken to add these kinds of user-level to kernel-level interfaces.

4.3.1 Ethernet

User-Mode Linux uses a virtual network to transmit and receive Ethernet packets as it has no access to the host networking. To do so, it can use a number of available interfaces, one of which is the ethertap interface. Using this interface, UML sends its Ethernet frames to user space on `/dev/tap0` and expects Ethernet frames to be written back to it. As a result, the ethertap interface can be viewed as a simple Ethernet device which receives packets from user space rather than a network interface card (NIC). The driver, `ethertap.c`, is used to support this networking paradigm.

Linux provides another interface, netlink, which is used to transfer information between kernel modules and user-space processes. It provides bi-directional communication via a standard socket interface for user processes and an API for kernel modules. This interface is used heavily in the ethertap driver. For example, `ethertap_open()` is simply a driver call which opens a netlink interface and creates a packet queue.

Moreover, the ethertap driver relies on the socket buffer, `sk_buff`. These are the buffers which the Linux kernel uses to handle network packets. This datatype provides a very rich functionality to pass network packets up and down the network protocol stack. That is, `sk_buff` is used to add or remove the headers necessary at the ICMP, TCP, IP, and other layers.

4.3.2 PS/2

As mentioned in earlier sections, adapting device drivers to Virtual Choices that directly access the hardware poses a challenge. We chose a simple PS/2 mouse driver to study how a hardware device driver can be adapted by our LKMAdapter. Given that this project's focus is that of the LKMA itself, we did not deem it necessary to design the interface required for such an effort. Though not implemented, this section makes some technical recommendations to adapting this type of device driver to Virtual Choices.

The standard PS/2 mouse supports the following inputs [9] : X-direction movement, Y-direction movement, left button, middle button, and right button. The mouse reads these inputs at a regular freqency and updates counters and flags to reflect movement and button states. The X-movement counter and the Y-movement counter along with the state of the three mouse buttons, are sent to the host in the form of a 3-byte movement data packet. The movement counters represent the amount of movement that has occurred since the last movment data packet was sent to the host. There are other factors like resolution and scaling that control the amount of movement, but those details does not concern us.

A PS/2 mouse can operate in "Reset", "Stream", "Remote" or "Wrap" mode. Reset mode is entered at power-up or upon receiving the "Reset" command. In stream mode, the mouse sends movement data when it detects movement or a change in state of one or more mouse buttons. The maximum rate at which this data reporting may occur is known as the sample rate. For simplicity, we consider only the stream mode which is the default mode of operation. This means that the simplest PS/2 mouse driver a) detects the mouse at start-up b) handles I/O interrupt at the "sample rate" to read in 3 bytes of movement data.

A Linux PS/2 mouse driver cannot be directly adapted to Virtual Choices since any interrupts generated by the mouse will be sent to the host kernel and will not be captured by VirtualChoices. Since User-mode Linux (UML) has similar handicaps regarding hardware access, we tried to adapt a PS/2 driver that works with UML. We found that UML also has unresolved issues in accessing physical devices. As discussed in [8], two things must be considered for hardware support in UML:

- **I/O memory acces**: Mapping I/O memory into a process virtual address space. This would give a driver access to the device's memory and registers.
- **Interrupts**: Device interrupts need to be forwarded to the user-mode driver. This could be done using a stub driver in the native kernel which can probe for the device at boot time. This driver would recognize the device and provide some mechanism for the real user-mode driver to gain access to it, such as an entry in /proc or /dev. The user-mode driver would open that file and make requests of the stub driver with calls to ioctl. The file descriptor would also provide the mechanism to forward interrupts from the stub driver to the user-mode driver. The stub driver's interrupt routine would raise a SIGIO on the file descriptor.

Thus, implementing a PS/2 mouse driver in UML requires:

1. A stub driver in the host kernel, `ps2mousestub`
2. A user mode driver for the UML kernel, `ps2mouse`.

The `ps2mousestub` is similar to a stub in remote procedure call. It presents the same interface to the host kernel as a real PS/2 mouse driver. However, instead of just interacting with the I/O subsystem, it acts as a mediator between the device and the `ps2mouse`. For example, when the mouse is clicked, `ps2mousestub` responds to the interrupt, and generates a SIGIO to `ps2mouse` which can then read the movement data. We infer that unlike `ps2mousestub` which must be loaded at startup, `ps2mouse` can be designed as an LKM.

Our LKMAdapter can adapt the user-mode driver, `ps2mouse` as with any LKM. Prior to adaptation, the PS/2 interface must be known to Virtual Choices. This requires creating a new class for Virtual Choices, `VCps2mouse`. Adapting `ps2mouse` then requires mapping its interface methods to corresponding methods in `VCps2mouse`.

To summarize, adapting a PS/2 mouse driver would require considerable work involving Linux and UML as we would have to first create `ps2mousestub` (for Linux) and `ps2mouse` (for UML). Given that our focus is on the actual adapter and the timeframe was limited, we left it at analysis level.

4.4 The `EthernetDriverLKM` Adaptor

Any non-trivial Linux driver or module will require many basic Linux kernel functions, such as kmalloc, kfree. These are fairly easy to emulate, as they have direct equivalents in the Choices kernel. However, in adapting the UML ethertap driver, their were some additional special considerations that had to be made. That is, there were some very non-trivial Linux services and interfaces with no direct equivalent in Choices. In particular, the fact that structures, or pointers to structures, are commonly used to pass data around means that our emulation functions have to know the same definitions of those structures as were used when the Linux module was compiled. When we reference object*rarrow*field, it must point to the same location in memory as the Linux module.

In order to write EthernetDriverLKM–the adaptor for User Mode Linux's ethertap driver–there are two major structures that have to be defined, and their supporting functions implemented. The first of these is Linux's net_device structure. This structure contains information about the network device being driven, but most importantly it is used to pass function pointers from the LKM to the kernel. The kernel then uses these function pointers to call the routines provided by the LKM. We necessarily emulate this behavior in EthernetDriverLKM, providing, among other things, a register_netdev() emulation function that receives the pointer to ethertap's net_device structure.

The second major Linux structure that EthernetDriverLKM has to understand is the socket buffer structure known as sk_buff. There are many supporting functions that make use of the sk_buff structure but luckily most of these are inline, and thus are compiled into the LKM, and don't have to be emulated. Nevertheless, sk_buff presented a challenge to our adaptor implementation. The socket buffer structure is used in Linux to pass network packets up and down the protocol stack, since each layer may need to add or remove packet headers, the socket buffer is designed to easily accommodate expansion and contraction at both ends. Of course, Virtual Choices has its own protocol stack (though they are called conduits), and thus the special capabilities of sk_buff are not useful to us. However, it means we have to go through a conversion process, translating Virtual Choices NetworkBuffer structures into Linux sk_buff structures, and back again whenever EthernetDriverLKM needs to get data from, or put data into, ethertap.

To do transmission and reception, ethertap makes use of the Linux netlink API. To do reception, it passes a function pointer to netlink_kernel_create(), which sets up an interrupt handler, to call the reception function whenever a packet is received. Clearly, we needed to emulate this function, using Virtual Choices interrupt handling. When ethertap transmits a packet, it calls netlink_broadcast(). We also had to emulate this function. The Data Link Packet Interface (DLPI) was used to perform the actual reception and transmission, since neither Virtual Choices nor the UML ethertap module has access to actual hardware.

There were, of course, numerous other minor functions and structures that had to be dealt with, but the above mentioned items were the most challenging, and must fundamental to getting ethertap to work with Virtual Choices.

5 Code Documentation

The CVS repository for the Virtual Choices source base used for this project is located at:

/usr/dcs/csil-projects/cs423/cs423g3/cvsroot.

The following files represent the actual implementation of the LKMA. These files were created by our team unless otherwise indicated.

- vcnew/FileSystems/
  - BSDContainer.cc. Modified to incorporate ELFContainer. BSD file system class.
  - BSDInode.cc. Modified to incorporate ELFContainer. BSD I-node file system class.
  - ELFContainer.cc. ELF file parser.
  - ELFDictionary.cc. Directory indexing class for ELF files.
  - ELFLoader.cc. ELF binary run-time linking and relocation.
  - FileSystemInterface.cc. Modified to incorporate ELFContainer and ELFDictionary. File system requests are handled via this interface.
- vcnew/LKMAdaptors/
  - HelloWorldLKM.cc. Example LKM adaptor class.
  - EthernetDriverLKM.cc. Adaptor for ethertap.o, UML ethernet LKM.
  - LKMAdaptor.cc. LKMA base class. Provides the basic adaptor functions and infrastructure.
- vcnew/MachineDependent/VChoices/
  - dlwrap.c. provides SUN's implementation of DLPI API routines. DLPI has a request acknowledge scheme for both data transfer and configuration changes and this file abstracts this REQ/ACK mechanism away from the systems programmer
  - VCEthernet.cc. Modified the Virtual Choices Ethernet driver to use the DLPI implementation.
  - VCVirtualEthernetDaemon.cc. Provides packet transmit/receive functionality for Virtual Choices instances on the same host.
- vcnew/Networks/Ethernet/
  - EthernetAdministrator.cc. Modified to perform packet transmission using the DLPI implementation.

6 An Example

Adapting an LKM using the LKMA requires the following steps.

1. Compile the LKM into ELF 32-bit format on a Sparc host, lkm.o.
2. Copy the lkm.o file onto the Virtual Choices file system.
3. Select the driver interface class appropriate to the driver. For example, EthernetDriverLKM for ethertap.c.
4. Load the lkm.o and adapt it utilizing the appropriate driver interface class.

What follows is an actual Virtual Choices session demonstrating these steps. In the following case, the EthernetDriverLKM is adapted, so the VCVirtualEthernetDaemon must be run previous to running Choices.

```
% cd vcnew/Configure/System/VChoices/
% VCEthernetDaemon &
Virtual ethernet daemon starting up.....
% Choices

Choices [Version 0.9 beta]
All rights reserved.
Information at http://choices.cs.uiuc.edu

This product includes software developed by
the University of California, Berkeley
and its contributors.

(additional output removed for brevity)

Enter path of binary executable application file:
cpon ethertap_demo.o

(debug output removed for brevity)

Enter path of binary executable application file:
start EthernetDriverLKM ethertap.o

(debug output removed for brevity)
(packets are transmitted and received)
```

7 LKMA Evaluation

At the onset of LKMA development, we forecasted three points as a measure of the adaptor's success. We evaluate the success of this project using these three points.

1. **The ability** of the adapter to translate an empty driver that the Virtual Choices OS can successfully load. The first driver adapted to Virtual Choices, as presented in the midsemester demo, was a simple "Hello World" driver, `test.o`. This driver had the two fundamental calls, `init_module` and `cleanup_module`. The LKMA successfully adapted this simple driver and it was demonstrated to work in Virtual Choices.
2. **The functionality** of the Kernel Linux Modules. The two drivers that were adapted for this project, `test.o` and `ethertap.o` were both shown to function in Virtual Choices. It was immediately apparent that the adapted `test.o` worked given its simplicity. "Hello World" was displayed on the screen. Ensuring that the `ethertap.o` adaptation worked was not as trivial. Using the packet sniffer ethereal, it was determined that the Ethernet-level packets were successfully being transmitted by the `EthernetDriverLKM`. Significant effort was made to resuscitate the TCP/IP stack–the Conduit class–in Virtual Choices as well as add an ICMP layer, but this was not successful due to time contraints.
3. **The number** of Loadable Kernel Modules adapted; be it they are actually adapted, or valuable insights are provided to doing so. In the early visions of this project, it was projected that a number of adapted drivers would be delivered at the semester's end. However, due to the significant work involved in getting Virtual Choices running, not much time was allowed for this endeavor. However, the two drivers, `test.o` and `ethertap.o`, were an effective demonstration of the capabilities of the LKMA. Furthermore, the reader may find valuable insights into adapting a mouse driver to Virtual Choices in Section 4.3.2.

8 Contributions

This project made four significant contributions to Virtual Choices. First, the Virtual Choices source base was updated to work with current day C++ compilers. Though this contribution was only ancillary to the main goals of LKMA development, it was certainly necessary. However, this contribution represents much of this semester's work for this project.

The second contribution was the `ELFLoader` class. For the LKMA project, it replaced the legacy `COFFLoader` class used by Virtual Choices to load its dynamic classes. However, the COFF binary format is no longer a standard for Sparc machines and Linux has long used the ELF format. The `ELFLoader` improves Virtual Choices by enabling it to

The third contribution was updating the Virtual Choices ethernet interface from NIC to DLPI. This allows Virtual Choices to use its own `VCEthernetDriver` to transmit Ethernet packets as well as for the adapted `EthernetDriverLKM` to do the same.

The final contribution was the addition of the LKMA to the Virtual Choices. The LKMA consists of an `ELFLoader` and a set of classes used to adapt the device drivers. The majority of the goals for the LKMA were met, saving the multitude of drivers actually adapted. This signifies that a framework now exists in which the rich resource of Linux drivers can now be adapted to function in Virtual Choices, thereby increasing the functionality of this research OS.

Biographies

Tanya L. Crenshaw is a Ph.D. student at the University of Illinois at Urbana-Champaign. She is currently a member of the Real-Time Systems Laboratory group working under Dr. Marco Caccamo. Her contributions to this project included her position as group leader, documentation organization, extensive Linux and User-Mode Linux administration for creating LKMs on the Sparc Platform, dissecting the Virtual Choices Conduit system for use of a TCP/IP stack.

Archana Dutta is a masters student at the University of Illinois at Urbana-Champaign currently residing in Connecticut and working for Pfizer. Her contributions to this project are building the crosstoolchain to generate sparc32 object files on sparc64, analyzing how a PS/2 driver can be adapted to Virtual Choices and, modifying the Conduit framework in Virtual Choices to transmit/receive ICMP messages.

Scott Kircher is a Ph.D. student at the University of Illinois at Urbana-Champaign. He is a member of Dr. Michael Garland's Multiresolution Mesh Processing graphics research group. Among his contributions to the project

were the linking and relocation portion of `ELFLoader`, the `LKMAdaptor` architecture design, many of the actual adaptor functions, and the Virtual Choices file-system hacks necessary to get the project off the ground.

Deepu Chandy Thomas Deepu Chandy Thomas is a final year masters student working with Prof. Marco Caccamo in the Real-Time Systems Laboratory. His contributions towards the project include a ELF file parser that is implemented in ELFContainer and ELFDictionary classes, the Virtual Choices Ethernet driver and Linux API emulation code for Choices.

References

[1] Campbell, R.; Islam, N.; Raila, D.; Madany, P. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*. September 19 93, Vol. 36, No 9, pp. 117-126

[2] Goel, S.; Duchamp, D. Linux Device Driver Emulation in Mach, 1996. *USENIX Annual Technical Conference*.

[3] Raila, David. *Getting Started with Choices*.

[4] Russo, V. Campbell, R. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. Oopsla. 1989.

[5] Intel. Tool interface standard portable formats specification (version 1.1), October 1993. Intel order number 241597

[6] Infecting loadable kernel modules. http://phrack.org/show.php?p=61&a=10

[7] Linux Loadable Kernel Module HOWTO and insmod.c source code

[8] The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/

[9] PS/2 Mouse/Keyboard Protocol. http://panda.cs.ndsu.nodak.edu/

[10] Choices Source Code. COFFLoader module, Class module, Kernel module.

[11] Executable and Linkable Format [ELF] - Tool Interface standards Portable Formats Specification, Version 1.1

[12] How to Use DLPI: Neal Nuckolls June 30, 1992. SUN Internet Engineering