DISSERTATION

LINUX KERNEL HOT-PATCHER FOR i386

Submitted in partial fulfillment of the requirements of

M.S. Software Engineering Degree programme

By

S.Sriram
1998HZ70968

Under the supervision of

Sujata Gaddemane,
Technical Lead, Embedded Linux COE
Embedded & Internet Access Solutions

Dissertation work carried out at

Wipro Technologies, Bangalore

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
Pilani (Rajasthan) INDIA

March 2002

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

Dissertation Title : **Linux Kernel Hot-Patcher for i386**
Name of Supervisor : **Sujata Gaddemane**
Name of student : **S.Sriram**
ID No. of student : **1998HZ70968**

Abstract

"Linux Kernel Hot-Patcher" provides a mechanism to hot patch a Linux kernel on i386 based machines. Hotpatching is a process of making changes to the kernel of the operating system, and activate these changes without rebooting the system. Functions that are patched can be made active or inactive atomically as required by the user. These dynamically applied patches using the tool re-vector crucial kernel code to the patched code without interrupting the operation of applications. This greatly reduces the downtime to diagnose, test, analyze, and correct operating system kernel related problems.

The primary audience for this facility will be kernel development and sustaining engineers. This report describes the design, implementation and usage of the hot-patching facility.

This document assumes that you have some basic knowledge on Linux operating system programming and kernel in general.

List of Symbols and Abbreviations used

HOP	Hot Patcher
LKHP	Linux Kernel Hot Patcher
TLB	Translation Look aside Buffers
KERNEL_CS	Kernel Code Segment Offset
SMP	Symmetric Multi Processing
RPM	RedHat Package Manager
GCC	Gnu C compiler
GNU	GNU is not Unix
Binutils	A collection of binary tools.
I386	Intel processor architecture based machines.
nm	An utility, which lists symbols from object files.
Bash	GNU Bourne-Again SHell
ELF	Executable and Linkable Format
Ioctl	Interface to send commands to a device
root	Default super user on a Linux system.

List of Tables / Figures

Table of Contents

Introduction

Conventionally, changing an existing kernel function or adding a new kernel function in the Linux kernel requires re-building the Linux kernel image (to include the changes in the kernel functions) and then rebooting the system on the newly built kernel image.

Rebooting the system is a costly operation since the system is unavailable for use during that period. In addition, this involves shutdown of all existing applications. For end-users, system shutdown presents serious system availability issues. For kernel developers, it results in costly development/debug cycles.

Kernel development and sustenance engineers need to frequently deal with the core base system, not just the “extended” system. They also frequently need to deal with, and operate on, kernel modules at a much finer level i.e., at the function level. No Linux tools are available to address this at run time. ‘LKHP’ tries to fill this gap.

This project involves developing a hotpatching application for Linux on i386 based architectures. This tool is basically intended as a debugger for operating system kernel related bugs or enhancements.

To patch a live kernel using this tool, kernel programmers need to modify the required the C or Assembly source files, compile them and pass the object file to this tool to patch the kernel.

Chapter 1 Design of LKHP, describes the design followed and the rationale behind it to implement lkhp.

Chapter 2 Installation and User’s Guide provides information on using this tool to hotpatch the kernel. This also serves as a reference for the command line tool ‘lkhp’ supplied with the kit.

Chapter 3 Lists the issues and limitations of this tool.

Chapter 4 Contains details on work that will be taken up in future on lkhp.

Chapter 5 Lists the tools used.

Chapter 1

1. Design of the tool

The major part of the Linux kernel code is written in the C language. To effectively use the facilities provided by the kernel to system programmers, this tool should be written in C or assembly language. So the tool will be implemented in C language.
Bash Shell scripts will also be used wherever required.

1.1 Overview

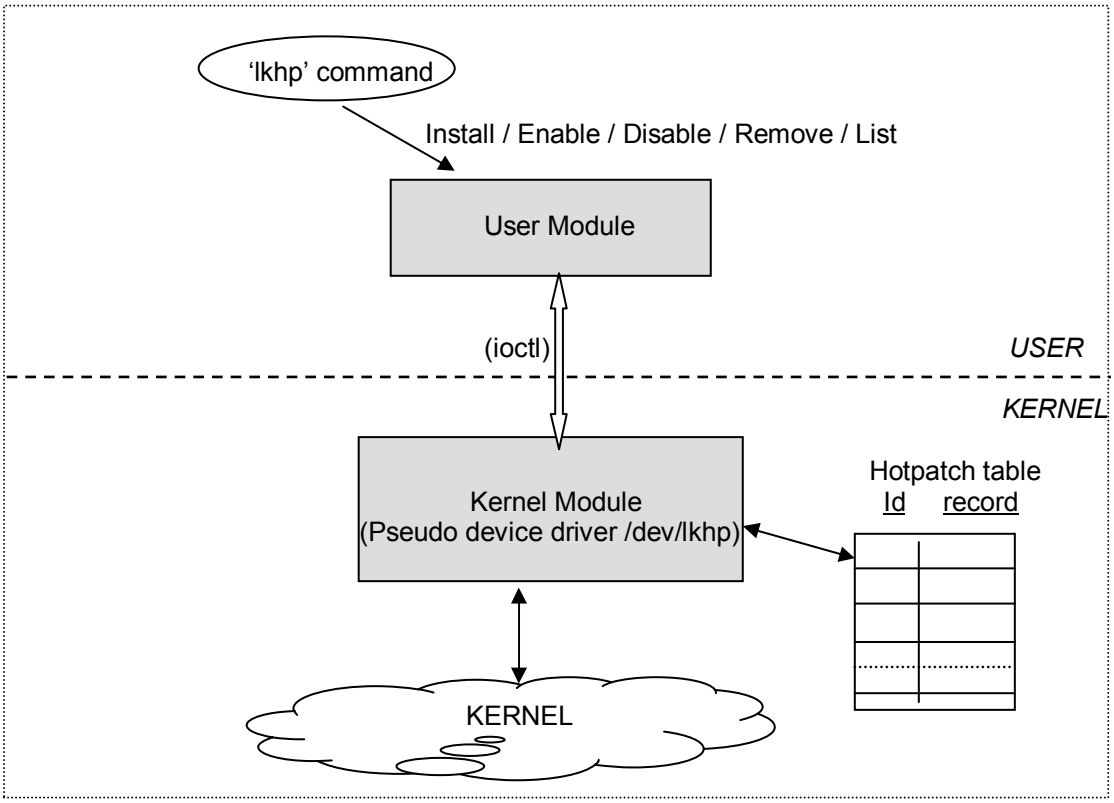
The design of this tool can be seen as containing components to do the following:

- Load or unload a hotpatch.
- Strap the module into kernel by relocating all the references, and installing it into the kernel. The relocation of the module will be done using the kernel object file.
- Patch/Unpatch the symbol in the kernel.

For patching, the part, which actually applies the patch should execute in kernel mode. So this will part will be implemented as a loadable kernel module. A command line utility will be provided to the user for performing hotpatching operations.

Basically, the design consists of two modules, a **User Module** and a **Kernel module**(pseudo device driver). User module installs the patch into memory as a loadable kernel module, and calls the kernel module to strap the patch into place. The kernel module then does the actual patching operation by replacing instructions in the kernel code segment.

Figure 1: Shows the high-level architecture of the tool.



Each hotpatch being loaded into the kernel will have a kernel module associated with it. The name of this hotpatch module will be the name of the symbol being patched.

The actual patching is done by inserting a long jump instruction(ljmp) as the first instruction at the address of the symbol being patched. This process, and how it works will be further explained in section 1.4.

To keep track of the state information of the hotpatches, a doubly linked list will be maintained. This list will exist in the kernel space, and will be maintained by the kernel module.

Each hotpatch applied will be identified using an id. This id is an unsigned integer, which will be automatically given by the tool.

1.2 Global Data Structures and Shared Data Functions

The following global data structures will be used across the implementation

1.2.1 /dev/lkhp device file

This device file is created for interaction between the user module and kernel module. The major number of this module will be dynamically obtained from the system. For security reasons, only superuser(root) will have permissions to open or operate on this file.

1.2.2 HotPatch structure

This Kernel data structure is used to track the hot-patched kernel symbols. This structure is the hotpatch definition structure. This stores the state information of the hotpatches applied on the kernel. Every hotpatched symbol has a record of type HotPatch. All the records of the hotpached symbols are maintained as a doubly linked list. Every record in this list has the following structure:

Figure 2: HotPatch Structure

```
typedef struct HotPatch
{
#if __KERNEL__
   struct list_head list;
#else
   void            dummy[2];           /* Padding in user space       */
#endif
   unsigned int    id;                 /* Hotpatch id                 */
   char            obj_file[64];       /* File used to load the patch  */
   char            symbol_name[64];    /* Symbol that got patched      */
   HopState        state;              /* ENABLED or DISABLED          */
   unsigned char   original_ins[7];    /* The original instruction     */
   unsigned char   patch_ins[7];       /* The patching instruction     */
   void            *hop_addr;          /* Address of replacement code  */
   void            *orig_addr;         /* Address of original code     */
   time_t          load_time;          /* Time this patch was loaded   */
   time_t          enable_time;        /* Time last enabled            */
   time_t          disable_time;       /* Time last disabled           */
} HotPatch;
```

```
TODO: Used str size 64 for object & Symbol name. Change to use a char * with len
```

1.2.3 HopState enum

This is a enum structure used in the kernel space to show the state of a hotpatch.

```
enum
{
       ENABLED,
```

```
        DISABLED,
ERROR
} HopState;
```

1.2.4 User Module, Kernel Module interface structures

These structures are used to shuttle data between the kernel module and user module. These structures will be passed as reference pointers from user space to kernel space. The kernel module, upon receiving it, will read the data from user space and write back the data needed by user space.

1.2.4.1 Installation of a patch

This structure is used while installing a hotpatch.

```
typedef struct
{
    HotPatch *hotPatch;
    ErrCode *err;
} InstallHotPatch_t;
```

Using this structure, the kernel address of the symbol being patched, and the new symbol address, which patches the kernel symbol will be sent from user space. The kernel module will write the error, if any occurs.

1.2.4.2 Enabling/Disabling/Removing a hotpatch

The following data structure will be used while enabling/disabling/removing a patch.

```
typedef struct
{
    int     id;
    ErrCode *err;
};
```

For these operations, the id of the hotpatch has to be sent to the kernel. Kernel module, then acts upon this id, and returns the error code in the ErrCode member.

1.2.4.3 Get information about a hotpatch

From user space, this structure is used to retrieve information from the kernel hotpatch table. The id of the hotpatch is sent, and the kernel module returns the hotpatch table entry in the reference pointer HotPatch *returnHotPatch.

```
typedef struct
{
    int      id;
    HotPatch *returnHotPatch;
    ErrCode *err;
} GetHotPatchInfo_t;
```

1.2.5 Ioctl commands for the driver

The user module described below uses lkhp driver to issue commands and retrieve information from the kernel module. It has a set of well-defined ioctl commands. Namely,

1.2.5.1 LKHP_INSTALL_HOT_PATCH

This command will be issued by user module to install a new hotpatch into the kernel.

1.2.5.2 LKHP_GET_HOTPATCH_INFO

Retrieve the hotpatch record from the kernel hotpatch table.

1.2.5.3 LKHP_REMOVE_HOTPATCH

Remove an installed hotpatch.

1.2.5.4 LKHP_ENABLE_HOTPATCH

Enable an already disabled hotpatch.

1.2.5.5 LKHP_DISABLE_HOTPATCH

Disable a hotpatch.

1.2.5.6 LKHP_NUMBER_OF_PATCHES

Returns the number of patches installed on the system.

1.2.5.7 LKHP_GET_ALL_PATCH_INFO

Copies all the records of the patches from kernel space.

1.2.6 Error and Warning codes

The following enum data type is used to refer to errors or warnings.

```
typedef enum
{
    noError = 0,

    /* Warnings */
    generalWarning = 100,

    /* Errors */
    generalError = 200,
    patchNotFoundError,
    symbolNotFoundError,
    outOfCPUMemoryError,
    illegalOperationError,
} ErrCode;
```

The errors and warnings are self-explanatory.

1.3 User Module

The user module does a set of functions that needs to be performed before patching a symbol. As the name suggests, the user module completely exists in the user space of the operating system.

This interacts with the kernel module(driver) through ioctls for performing hotpatch related operations.

The user module provides the following functions:

- Provides a command line utility called 'lkhp' to the user for administering a hotpatch. This utility can also be used to get information about the hotpatches installed on the system.
- Processes the object file for loading into kernel.
- Obtains the kernel address of the symbol to be replaced, and the address of the symbol, which replaces it.
- Pass on the patching information to the kernel component to replace the existing symbol in the kernel image.

1.3.1 Design details

The user module contains the following components

lkhp - A command line utility for performing operations from user space. This is a shell script to do all the preliminary checks and operations before operating on a patch.

hop - This is an executable binary, which does operations like making ioctls and querying the kernel symbol table for obtaining addresses. This should not be called directly. The lkhp utility calls this binary for operations.

strap – This utility is used to install the patch module into kernel. This utility is converted insmod code, which takes the symbol table from a file instead of kernel symbol table. This utility is needed because, if the patched symbol uses functions, which are not exported by the kernel, then the loading of the patch will fail because of undefined references.

1.3.2 Installing a hotpatch

For installing a patch, the module will be supplied with an ELF object file, which contains the replacement symbol. The following steps are then followed in user module while installing a hotpatch.

- Link the object file with the stub module, and generate a module.
- Install the module into the kernel using the 'strap' utility.
- Get the address of the symbol to be replaced, and the address of the new symbol.
- Make an ioctl call to the kernel pseudo driver module with 'INSTALL_HOT_PATCH" command. Pass the addresses of both the symbols to the driver as parameters.
- If the ioctl returned 0, then return normally. Otherwise print an error message.

1.3.3 Enabling/Disabling/Removing a hotpatch

A patch installed can be disabled or enabled. I.e. a hotpatch patch can be turned on/off.

For enabling/disabling/removing a hotpatch, the user module makes an ioctl to the kernel pseudo driver with the appropriate command. The commands are LKHP_REMOVE_HOTPATCH, LKHP_ENABLE_HOTPATCH and LKHP_DISABLE_HOTPATCH. It passes the hot patch id as an argument for the kernel to take action on the specified hotpatch.

While removing a hotpatch, the module also removes the patch module associated with it. i.e. whenever a patch is applied, the patch is linked with the dummy module and is installed into the kernel. The name of the module is same as the name of the symbol being patched. This patch module will be removed when the patch is being removed.

1.3.4 Query the hotpatch table

User module queries a hotpatch table entry by using ioctl commands LKHP_GET_HOTPATCH_INFO, LKHP_NUMBER_OF_PATCHES, and LKHP_GET_ALL_PATCH_INFO to the kernel pseudo driver. It passes hotpatch id for reference. The pseudo driver returns the hotpatch table entry requested in a reference pointer.

1.3.5 External Interfaces

1.3.5.1 Required

The followint interface are required for the user module to function properly

- The System.map file has been generated from the vmlinux object file during setup.
- Provided a function name, a function to get the symbol address. query_module().
- A module stub to integrate and install the object file into kernel as a module.
- /dev/lkhp device.

1.3.5.2 Provided

This module provides a utility called lkhp to the user. lkhp is a command line utility for administering and querying the hotpatching module. This will be installed in the /usr/sbin directory. This utility provides the following functions:

Install a hotpatch:

```
$ lkhp install –f <obj-file.o>  -s <symbol-name>
```

Remove a hotpatch:

```
$ lkhp remove  -i <hot_patch_id>
```

Enable/Disable a installed hotpatch.

```
$ lkhp disable/enable  -i <hot_patch_id>
```

List all patches or get information about a patch:

```
$ lkhp list [ -i <hot_patch_id>]
```

1.3.6 Assumptions

The following assumptions are made during the operations

- The operating system provides an ELF library to manipulate the object file.
- The object file used to patch the kernel is compiled without optimization. i.e. without –O Flag.
- The kernel symbol table is available, and is accessible from user space.

1.4 Kernel Module

The kernel part of the application is written as a pseudo-driver. Since drivers can be plugged-in to the kernel, and since driver ioctls provide a simple and flexible mechanism to extend the services, the kernel component has been implemented as a pseudo-driver.

The user module uses the functions of kernel module using ioctls. The device driver module provides the following functions to the user module:

- Installing a hotpatch.
- Enabling a hotpatch to make it active.
- Disabling an active hotpatch.
- Permanently removing a hotpatch.
- Query the hotpatch table and return information to the user space.

- Maintain the hotpatch table, which keeps track of the hotpatches installed.

1.4.1 Design Alternatives

1.4.1.1 Implementing as system calls

The kernel component, instead of being implemented as a pseudo driver, can be implemented using different system calls. But as the number of requests, which correspond to services in the kernel component grow, it may not be a feasible idea. Moreover, implementing driver ioctls provide a powerful as well as a simple and convenient mechanism for the user level programs to interact with the kernel components.

Moreover, system calls cannot be loaded as a module, and should exist in the kernel. So the LKHP tool should be given as a patch to the kernel, which is not desirable.

1.4.1.2 Patching using Interrupts

Right now, patching is done by writing a 'ljmp' instruction as the first instruction in the symbol being patched. But this can also being implemented using an interrupt based patching. Here, an interrupt is reserved for hotpatching operations. For patching, an 'int <hotpatch-int>' instruction is written at the start of the symbol. The interrupt handler for this<hotpatch-int> is registerd as our hotpatching module. So whenever this interrupt occurs, the module takes control and executes the hotpatch code, and returns the value back to the calling function.

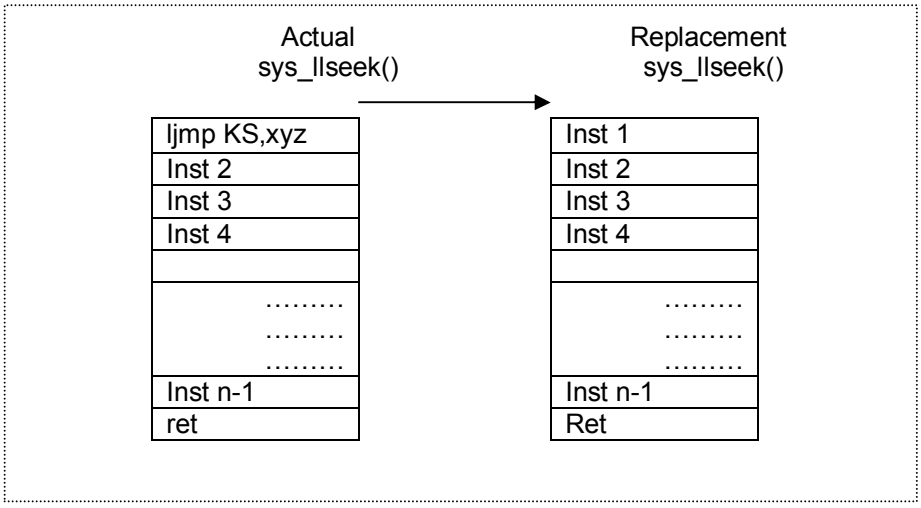
This approach was not taken because of the complexities involved in handling different types(prototypes) of symbols.

1.4.2 Design Details

1.4.2.1 Installing a hotpatch

The user module sends the address of the symbol to be replaced and the address of the symbol, which replaces it. The kernel module installs the hotpatch by replacing the first few instructions of the symbol to be replaced by a jump instruction. I.e. if a function sys_llseek() needs to be replaced by a new sys_llseek(), then the module replaces the first few instructions of the function actual sys_llseek() with a jump instruction. This jump instruction jumps to the sys_llseek() function, which is the replacement function.

Figure 3 Explains the process followed.



Installation is done using the following steps

- Get the kernel page table entry of the address of the original function.
- Disable the interrupts.
- Reset the protection bit of that page. i.e. if it is write protected, make it writable.
- Flush the TLB.
- Read the first 7 bytes from the address being patched and save it in the hotpatch record.
- Lock the kernel. (Needed only if we are using SMP based systems)
- Write the ljmp KERNEL_CS, <address> as the first instruction in the symbol being patched.
- Flush I-cache and write back d-cache. (Not needed on intel based systems)
- Unlock the kernel.
- Reset the protection bit, if it was changed earlier.
- Flush the TLB.
- Enable interrupts.
- Update the kernel hotpatch table.
- Return success to the user module.

1.4.3 Enabling/Disabling a patch

For enabling a patch, the same process as in installing a patch is followed. Except that the addresses of the symbols will be referred from the hotpatch table

The following steps are followed while disabling a patch:

- Get the page structure of the address of the original function.
- Disable the interrupts.
- Reset the protection bit of that page. i.e. if it is write protected, make it writable.
- Flush the TLB.
- Lock the kernel. (Needed only if we are using SMP based systems)
- Replace the jump instruction with the original instruction that was stored in the hotpatch table.
- Flush I-cache and write back d-cache. (No need in intel based machines)
- Unlock the kernel
- Reset the protection bit, if it was changed earlier.
- Flush the TLB.
- Enable interrupts.
- Update the kernel hotpatch table.
- Return success to the user module.

1.4.4 Removing a patch

The following steps are followed to remove a hotpatch:

- The user module supplies the id of the hotpatch table that needs to be removed.
- If the hotpatch is already disabled, then
  - Remove the entry from the hotpatch table.
  - Return success to the user module
- Get the page table entry of the address of the original function.
- Disable the interrupts.
- Reset the protection bit of that page. i.e. if it is write protected, make it writable.
- Flush the TLB.
- Lock the kernel. (Needed only if we are using SMP based systems)
- Replace the jump instruction with the original instruction that was stored in the hotpatch table.
- Flush I-cache and write back d-cache.
- Unlock the kernel
- Reset the protection bit, if it was changed earlier.

- Flush the TLB.
- Enable interrupts.
- Remove the entry from the hotpatch table.
- Return success to the user module.

1.4.5 Global Data Structures and References

1.4.5.1 Hotpatch table

This is the table where entries are stored on the hotpatches applied to the kernel. Each patch applied has an entry in this table. This table is maintained as a doubly linked list.

1.4.6 External Interfaces

1.4.6.1 Required

The module uses system calls provided by the operating system for the following services:

- Given a kernel address, access the corresponding page structures. It also uses functions provided to arbitrarily manipulate the page protection bits of the pages specified.
- An header file to manipulate the linked lists(lists.h).
- A mechanism to communicate between the modules. This is needed because each patch module needs to export its module pointer to the LKHP module for incrementing or decrementing the module count.

1.4.6.2 Provided

The module provides an ioctl interface to the user module to access its services. This is the only interface provided by this module. The following ioctl commands are available to the user module
LKHP_INSTALL_HOT_PATCH
LKHP_GET_HOTPATCH_INFO
LKHP_REMOVE_HOTPATCH
LKHP_ENABLE_HOTPATCH
LKHP_DISABLE_HOTPATCH
LKHP_NUMBER_OF_PATCHES
LKHP_GET_ALL_PATCH_INFO

Each ioctl returns error code, if any in a reference pointer back to user space.

1.4.7 Assumptions

The kernel module assumes the following about the system that runs on

- The system is based on Intel Architecture.
- The operating system provides mechanisms to access the page structures, and arbitrary manipulation of the protection bits corresponding to these pages.

1.5 Setup Module

This module does the necessary preparations needed to setup the lkhp tool kit. The setup operation is performed each and every time a new kernel is being patched. This performs the following operations

- Create the System.map file using the vmlinux kernel object file supplied. This System.map file is used by the ‘strap’ utility to resolve the relocatable addresses in the patch module.
- Installs the LKHP.o kernel module, which contains all the kernel functions.
- Creates a device /dev/lkhp for the user space module to open. This device has read/write permissions only for the root.

The setup module is implemented as a ‘bash’ script.

Chapter 2

2. Source Code Structure

In order to help one to find his way through the files of the source code, the organization of the source is briefly described here.

All pathnames refer to the main directory where the source is unzipped.

2.1 main.c

This file contains the main LKHP module driver code. i.e. this is the main driver file, which registers/unregisters, provides the module file operations.
This contains the lkhp_ioctl() function, which does the initial ioctl specific functions.

2.2 lkhp.c

This file contains the core part of the tool. This has functions like

ErrCode InstallHotPatch(HotPatch *hotPatch)
ErrCode GetHotPatchInfo(int id, HotPatch **ReturnHotPatch)
ErrCode RemoveHotPatch(int id)
ErrCode EnableHotPatch(int id)
ErrCode DisableHotPatch(int id)
ErrCode RemoveHotPatch(int id)

The functionalities of the functions are pretty self explanatory from its name.

2.3 hop.c

This is a user space file. This file opens the /dev/lkhp file, and makes ioctls to the /dev/lkhp device. This also has code for querying the kernel symbol table for a symbol address.

2.4 dummy.c

This is a dummy kernel lodable module stub, which is used to link the patch object files with.

2.5 include/lkhp.h header

Common header file used in all the files mentioned above. This contains the ioctl commands, and data structures like HotPatch etc.,

2.6 lkhp script

This is the command line utility supplied to the user. This is entirely written in bash script. This performs basic command line checks and calls the ‘hop’ binary file with options supplied. This script also uses the ‘strap’ utility to install a module.

2.7 lkhp_setup

This is a bash script. This file contains the setup functions that need to be done before patching a kernel. This basically performs operations like loading the LKHP.o module, generating the System.map file for the 'strap' utility etc.,

2.8 lkhp_unload

As the name suggests, this unloads the hotpatch module LKHP. This does the cleanup activities like removing the /dev/lkhp device file from the system.

Chapter 3

3. Installation and User's Guide

Only users with super user privileges can install the LKHP package. Lkhp will be supplied as a binary RPM for i386 based machines. This package has been compiled on a Pentium III machine with RedHat Linux 7.1 with kernel version 2.4.2-2.

3.1 Prerequisites

The system being installed should have the following packages installed

- Kernel version 2.4.2 or above. This kernel should not be a preemptive kernel. Normally Linux kernel is not preemptive. But there are freely available patches, which when applied, makes the Linux kernel preemptive.
- GNU C compiler. (gcc).
- Binutils package. (nm)

3.2 Installing lkhp

For installing the RPM on the target machine, one should login as the superuser of the machine. i.e. only root user or users with root permissions can install the RPM.

Install the binary RPM supplied with the command.

```
$ rpm –i <lkhp-rpm-name>
```

This command will install the following files on the system

/usr/local/lkhp/LKHP.o
- The Kernel module, which loads the patching functions.

/usr/local/lkhp/hop
- An executable binary 'hop' to make the ioctls. This should not be called directly.

/usr/local/lkhp/strap
- Used to install the patch module into the kernel. This should not be called directly.

/usr/local/lkhp/dummy.c
- A module stub to compile and link with the patch file supplied.

/usr/sbin/lkhp
- A command line utility for performing hotpatching operations from user space.

/usr/sbin/lkhp_setup
- To setup the machine for hotpatching.

/usr/sbin/lkhp_unload
- Remove lkhp from memory. This unloads the module.

Of the above files installed, only the last three files should be used for hotpatching.

3.3 Setting up the machine for patching

After installing the RPM, setup needs to be done before patching. Since this is a debugging tool, it is assumed that you have a kernel source tree on the machine. The current kernel loaded, which will be patched, should have been compiled from this tree.

The compilation of the kernel will generate an object file 'vmlinux' in the root directory of the kernel source tree. This file is the object file of the kernel that is live on the machine. This object file should be supplied while the setting up the machine.

Run the /usr/sbin/lkhp_setup script for setting up the machine, and enter the path of the 'vmlinux' object file when asked for it.

```
$ lkhp_setup
```

The setup operation will generate a System.map file. This file will be stored in the /usr/local/lkhp directory for reference. This file will be used by the 'strap' utility to relocate symbols in the patch module.

The setup needs to be done whenever the kernel being patched is changed.

3.4 Patching the kernel

After the setup is over, the kernel is ready for hotpatching. Follow the below steps to patch the kernel:

In the kernel source tree, make the changes to the kernel symbol to be patched.
e.g. Say we need to patch a symbol xyz(), which is in file fork.c, then make the necessary changes in the fork.c file.

Compile the file that has been modified and obtain the object file.
e.g. Compilation of the modified fork.c will resulting fork.o file.

Patch the kernel using the 'lkhp install' command. The install option in the lkhp utility takes two arguments, an object file containing the patch symbol and the symbol name being patched.
e.g. $ lkhp install –f fork.o –l xyz

If the utility is able to relocate all the symbols in the object file, then the patch will be successfully installed on the system.

3.4.1 Word of Caution

- The symbols in the patch object file being used will first be locally relocated. i.e. if we are patching a function called do_fork(), and this function makes a call to a function called init_fork(), which is available in the same file, then the new patched function of do_fork() will use the local copy of init_fork(), if it is locally available. So it is always advisable to have only the symbol being patched in the patch object file being used to patch the kernel.

- If the symbol that is used to patch makes any calls to static functions, then these static functions should be present in the local file also. I.e. if xyz() function is getting patched, and it calls a function abc(), which is a
  ```
  static int abc();
  ```
  function, then the function abc() should be present in the patch file.

- The patch object file being compiled should not be compiled with the –O option. If the patch is compiled with –O option, then the calls to functions defined in the same file may result in the code of the function getting inserted into the place where the call was made.

- Patching cannot modify the structure of global kernel data structures. For example, if we have a structure

```
struct abc
{
  int a;
  char b;
};
```

  It is not possible to add a new member to this structure. If we try to add a new member, and try to pass he reference pointer back, to the kernel may crash because the length of the data structures are calculated at compile time.

Chapter 4

4. Issues and limitations

a) Only exported symbols can be hotpatched.

b) Only one patch can be applied at a time. I.e. more than one symbol cannot be patched at a time.

c) There cannot exist more than one patch for the same symbol existing at a given time. i.e. a symbol cannot be patched more than once. However the old patch can be removed and the new patch can be installed.

d) The prototype of the function being patched should be same as the function that is used to patch the symbol.

e) It is not possible to patch symbols exported by loadable kernel modules.

f) The tool is not SMP compliant. This cannot be run on a multi-processor machine.

g) If there are static variables inside a patched function, it is not possible to retain the previous values. These values will be assigned to its initial value and will be used.

h) The system behavior is undefined when a symbol being patched is already being executed by another process. Same applies for enabling or disabling a hotpatch.

i) Once rebooted, the hotpatches applied on the kernel will get lost. This is not a severe limitation, since whenever the system is rebooted, it is reasonable to load the new modified kernel.

j) This tool has been tested only on RedHat Linux 7.1 running kernel version 2.4.2-2.

k) This cannot be used on a system, which has a preemptive kernel. Normally Linux kernel is not preemptive, but there are some patches, which will make the kernel preemptive. For example, MontaVista has a preemptive kernel patch, which is downloadable.

l) The issues of NMIs(Non-Maskable Interrupts) are not addressed in this tool. Study needs to be done further in this area. However, this is not really an issue, since NMIs are normally hardware interrupts and were mainly used to reboot the system.

Chapter 5

5. Wish list

a) This tool has been intended primarily as a debugging tool for kernel developers. But once all the issues involved in kernel hotpatching has been solved, this should also be used for patching mission critical servers, which need patching.

b) Contact the open source community and propose this solution for hotpatching. This could have been done at the beginning of this project. But I did not do it because of the inherent delay this will bring into the development process. This should be done as soon as the BITS viva/demo is complete.

c) HotPatch even unexported symbols. This can be done with very few modifications. This tool dictates the availability of kernel object file(vmlinux). So the address of unexported symbols can easily be obtained from this kernel object file.

d) Instead of using 'strap' to install the patch module, use ELF libraries to relocate the object file supplied. i.e. do away with the 'strap' utility.

e) Right now the tool is not SMP compliant. I.e. this tool cannot run on a multi-processor machine. Address the synchronization issues involved in this, and make lkhp SMP compliant.

f) Port this to other hardware platforms like SPARC.

g) Atomically apply patches to more than one symbol at a time.

h) Add a man page to the 'lkhp' command.

Chapter 6

6. Tools used

The following are the tools used while developing this tool.

1.6 Tools used

These are the tools used to develop or debug LKHP.

nm - Lists symbols from object files.

Objdump - Displays information from object files.

cscope - cscope is an interactive, screen-oriented tool that allows the user to browse through C source files for specified elements of code.

Vim - Vim is a text editor that is upwards compatible to Vi. It can be used to edit any ASCII text. It is especially use-ful for editing programs.

Gawk - pattern scanning and processing language

Bash - Bash is an sh-compatible command language interpreter that executes commands read from the standard input or from a file.

Conclusion

LKHP mechanism provides an effective controlled method of patching functions on a live kernel. This facility may also be used as a patching mechanism for mission critical servers to maintain a high availability of service throughout the application of the patch until the next scheduled downtime, whereupon a more complete patch may be applied if necessary. However the tool should become ultra stable to be operated on a mission critical live server.

This facility can also be used by developers of device drivers to switch between functions that implement different algorithms so that a comparison of various approaches can be made on the fly.

The main objective behind developing this tool was to get an insight of Linux kernel programming. As it turned out, the objective was well met. This project involved more of study than actual design and implementation. It has given me a lot of knowledge on Linux and its organization. To do this project, had to study the following areas

- Linux Kernel module programming and loading.
- Paging in Linux.
- Linux kernel architecture.
- Segmentation in Linux.

Directions for future work

- LKHP may need further work to optimize kernel memory utilization, for instance, the 'strap' utility can be replaced with an patch ELF processing module for relocating the symbols. This will avoid the memory allocated for the .data segment while using 'strap' utility.

- The LKHP utility can be enhanced to patch even unexported symbols. As of now we query the kernel symbol table to obtain the target symbol address. If we change this part to refer the kernel vmlinux object file for obtaining the target symbol address, this should be possible.

- A solution to ensure that the symbol is not in use when it is being operated upon. i.e. when patching a symbol, it may be possible that another process is in the middle of executing the symbol, but for some reason relinquished the CPU. In such a scenario, the result is undefined. This can be avoided by using a tracking mechanism to track the execution of the symbol being patched.

- LKHP initializes the values of the static variables declared inside a function. This can be avoided by searching the kernel object file for these references. The static variables are shown in the <symbol_name>.<number> format. Work needs to be done to find out the reason behind this naming convention, and use it to initialize the static variables.

- Port LKHP to other hardware platforms. This should be pretty straight forward, since the code has been written mainly in C, and very less architecture specific code has been used. As I see it, the following should be addressed for porting the application to other platforms:
  - Are the instruction cache and data cache in the target hardware intelligent enough to detect a change immediately?. If not, then these caches should be flushed after patching the kernel code segment.
  - The opcode for long jump, if exists in the target hardware platform will be different. So this opcode should be used for patching.
  - Is it a little endian or big endian system?. Intel is based on little endian.

Appendix – A

'lkhp' command reference

```
NAME
      /usr/sbin/lkhp – Linux Kernel Hot Patcher


SYNOPSIS
      lkhp  [-h] [--help] [install|remove|list|enable|disable]
            [-f objfile] [--objfilename=objfile] [-s symbol]
            [--symbolname=symbol] [-i id] [--patchid=id]
            [-v] [--version]

Description

      Tool for administering the hotpatching mechanism on the kernel.  Install,
      remove, enable, disable and display information about a hotpatch.  This
      tool takes a command as its first argument. The command should be
      'install' or 'remove' or 'enable' or 'disable' or 'list'.
      Before using this tool, the LKHP.o module, which provides the kernel
      hotpatching operations should be loaded to the kernel.

Options

      install – Installs a hotpatch.
                Valid options [-f][-s].
                [-f objfile] – Object file containing the patch.
                [-s symbol]  – Symbol to be patched.
                e.g.
                   $ lkhp install –f fork.o –s do_fork

      remove  - Remove a hotpatch permanently from the kernel.
                Valid option [-i].
                e.g.
                   $ lkhp remove –i 3

      list    - List the information about the hotpatches in the kernel.
                Valid option [-i].
                e.g.
                   $ lkhp list
                   $ lkhp list –i 3

      enable  - Activate a disabled hotpatch.
                Valid option [-i].
                e.g.
                   $ lkhp enable –i 3

      disable – Disable an active hotpatch.
                Valid options [-i].
                e.g.
                  $ lkhp disable –i 3

Returns

      0 on success, -1 otherwise.
```

References

1. Book:
"Linux Device Drivers", Alessandro Rubini & Jonathan Corbet, O'Reilly, June 2001.

2. Book:
"Understanding the Linux Kernel", Daniel P.Bovet & Marco Cesati, O'Reilly, January 2001.

3. Journal paper:
"Implementing Loadable Kernel Modules for Linux", Matt Welsh, Dr.Dobb's Journal, May 1995.
URL: http://www.ddj.com/print/documentID=13776

4. A Tutorial Paper:
"Brennan's Guide to Inline Assembly", Brennan Underwood, Document Version 1.1.2.
URL: http://www.cs.princeton.edu/courses/archive/fall99/cs318/Files/djgpp.html

5. Manual:
"Intel Instruction Set Reference Manual", Intel Corporation, 1999.