# Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

# The Novell Linux Kernel Debugger, NLKD

Clyde Griffin
*Novell, Inc.*
cgriffin@novell.com

Jan Beulich
*Novell, Inc.*
jbeulich@novell.com

## Abstract

In this paper we introduce the Novell Linux Kernel Debugger. After a brief introduction we will go into an in-depth discussion of NLKD's architecture. Following the architecture discussion we will cover some of the features supported by NLKD and its supported debug agents. We wrap up the discussion with some of the work items that still need to be done and follow with a brief conclusion.

## 1 Introduction

NLKD began its life as an R&D project in 1998 by Novell engineers Jan Beulich and Clyde Griffin. The effort to build a new debugger was driven by a need for a robust kernel debugger for future operating systems running on Intel's Itanium Processor Family.

The project was a success and soon there was demand for similar functionality on other hardware architectures. The debugger has since been ported for use on x86, x86-64, and EM64T.

Novell has never formally shipped this debugger as a stand-alone product or with any other Novell products or operating systems. To dispel any myths about its origin, it was never targeted for or used with Novell NetWare. It remained a research effort until the summer of 2004 when Novell engineering determined that the capabilities of this tool would be a boost to Linux development and support teams.

At the time of the publication of this paper, NLKD is functional on x86, x86-64, and EM64T SUSE Linux platforms. A port to IA64 Linux is pending.

### 1.1 Non-Goals

While we believe NLKD is one of the most stable and capable kernel debuggers available on Linux, we in no way want to force other developers to use this tool. We, like most developers on Linux, have our personal preferences and enjoy the freedom to use the right tool for the job at hand. To this end, NLKD is separated into layers, any of which could benefit existing debugging practices. At the lowest level, our exception handling framework could add stability and flexibility to existing Linux kernel debuggers. The Core Debug Engine can be controlled by add-on debug agents. As a final example, NLKD ships with a module that understands GDB's wire protocol, so that remote kernel debugging can be done with GDB or one of the many GDB interfaces.

### 1.2 Goals

Novell's primary interest in promoting NLKD is to provide a robust debugging experience for kernel development engineers and enable support organizations to provide quick response times on critical customer support issues. While Novell development may favor NLKD as its primary kernel debugger, Novell will continue to support other kernel debugger offerings as long as sufficient demand exists.

NLKD has been released under the GPL with Novell retaining the copyright for the original work. Novell plans to ship NLKD as part of the SUSE distribution and at the same time enable it for inclusion into the mainline Linux kernel.

## 2 NLKD Architecture

Like any kernel debugger, at the core of NLKD is a special purpose exception handler. However, unlike many exception handlers, kernel debuggers must be able to control the state of other processors in the system in order to ensure a stable debugging experience. The fact that all processors in the system can generate simultaneous exceptions complicates the issue and makes the solution even more interesting.

Getting all processors into a quiescent state for examination has been a common challenge for multiprocessor kernel debuggers. Sending these processors back to the run state with a variety of debug conditions attached can be even more challenging, especially when processors are in critical sections of kernel code or operating on the same set of instructions.

The architecture that we describe here deals with this complex set of issues in a unique way, providing the user with a stable debugging experience.

In the following discussion we introduce the major components comprising NLKD. These include the exception handling framework supporting NLKD, the Core Debug Engine (CDE), and the debug agents that plug into CDE. CDE is a complex piece, so we spend extra time discussing its state machine and breakpoint logic. Figure 1 depicts these components and their interactions.

So let's start with the exception handling framework.

### 2.1 Exception Handling Framework

The first task in providing a robust debugging experience is to get an exception handling framework in place that properly serializes exception handlers according to function and priority.

While NLKD does not define the exception handling framework, our research at Novell has led us to a solution that solves the problem in a simple and elegant way.

The first thing to recognize is that not all exception handlers are created equal. For some exceptions, all registered handlers must be called no matter what. The best example of this is the x86 NMI. Other handlers are best called serially and others round robin. We should also note that interrupt handlers sharing a single interrupt vector should be called round robin to avoid priority inversion or starvation. Some exception handlers are passive and do nothing but monitor events and these, too, must be called in the right order.

To enable this flexibility, we defined a variety of exception handler types. They are: Exception Entry Notifiers, Registered Exception Handlers, Debug Exception Handler, Default Exception Handler, and Exception Exit Notifiers. Each of these handler types have strict semantics, such as how many handlers of each type may be registered, and whether all or just one

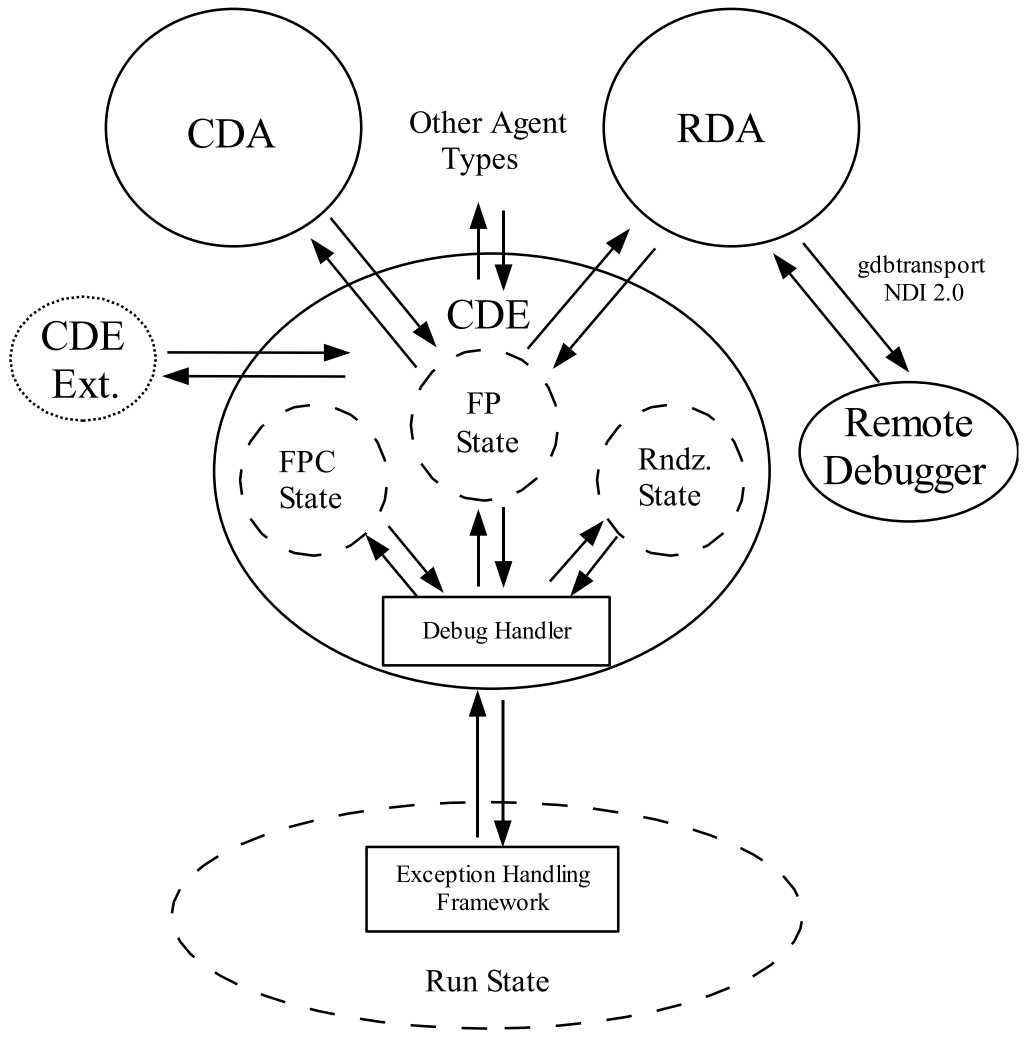

Figure 1: Architecture and state transitions

is called when an exception occurs. The various exception handler types are called in a well-defined order. Taken together, these rules ensure the system remains stable, and event counters remain correct, in all debugging situations.

The following sections describe the registration and calling conventions for each of these handler types. The handler types are listed in the order they are called.

**Exception Entry Notifiers** An exception entry notifier is a passive handler that does not change the state of the stack frame. It is typically used for monitors that want to know when an exception has occurred and what type it is. Zero or more such handlers may be registered, and all will be called.

**Registered Exception Handlers** These exception handlers are dynamically registered at runtime. If any of these handlers claim the exception, then no other registered exception handlers, nor the debug handler, are called. Zero or more such handlers may be registered.

**Debug Exception Handler** The debug exception handler invokes the debugger. (This may be NLKD or any other kernel debugger.) At most, one such handler may exist. If no debugger was registered, the exception is passed on to the default exception handler.

**Default Exception Handler** The kernel's default exception handler is included at compile time. Depending upon the exception type, it may cause the kernel to panic if no other handlers have claimed the exception.

**Exception Exit Notifiers** This is a passive handler that does not change the state of the stack frame. It is typically used for monitors wanting to know that an exception has been completed and what type it was. Zero or more such handlers may be registered, and all will be called.

The overhead of such an exception handler framework is extremely lightweight. For example:

```
// Multiple handlers test/call loop
while (ptr1) {
    if (ptr1->handler() == HANDLED)
        break;
    ptr1 = ptr1->next;
}
// Single handler test/call
if (ptr2)
    ptr2->handler();
```

There is very little overhead in this scheme, yet great flexibility is achieved.

With the framework in place allowing for exception handlers to be prioritized according to purpose, and by allowing those handlers to be registered at run time, we have enabled the kernel to load NLKD at run time. Note that the usefulness of such an exception system extends beyond just NLKD, as it enables a whole class of debuggers and monitors to be loaded dynamically.

Our current implementation does not actually load the Core Debug Engine (CDE) at runtime. CDE is currently a compile time option. However, with CDE in place, the debug agents (which we will discuss later) are loadable/unloadable at run time. This allows a user to switch from no debugger to an on-box kernel debugger or a remote source level debugger by simply loading the appropriate modules.

There have been many customer support scenarios that require debugging, monitoring, or profiling on production boxes in live environments. This must happen without taking the box down or changing the environment by rebuilding the kernel to enable a debugger.

There is some argument that a loadable kernel debugger is a security risk. To some degree this is true, but only insomuch as the root user is a security risk. Since root is the only user that can load kernel modules, such security concerns are negated.

It could easily be argued that the benefit of being able to load and then unload a debugger on demand provides even greater security in situations where a debugger is needed, since we can easily restrict the actual time that the debugger is available.

Let us reiterate that in our current implementation, adding support for CDE is a compile time option, like KDB, not a runtime option. But with CDE in place, kernel modules to support local or remote debugging can easily be loaded. Without a corresponding debug agent attached, CDE is inactive.

Let's now turn our attention to the next layer in the stack, the Core Debug Engine (CDE).

### 2.2 Core Debug Engine (CDE)

Sitting on top of the exception handling framework is a debugger infrastructure piece we have named the Core Debug Engine. This layer of the debugger provides three main functions. First, all NLKD state machine logic is located within CDE. Second, CDE provides a framework against which debug agents load and assume responsibility for driving the state machine and for providing interaction with the user. Finally, CDE provides a means of extending the functionality of the debugger.

The state machine also provides the infrastructure supporting the breakpoint logic, which is a key component and distinguishing capability of NLKD.

We will now examine each of these in turn.

#### 2.2.1 CDE State Machine

NLKD divides the state of each processor in the system into four simple yet well-defined states. These states are: RUN state, FOCUS PROCESSOR state, RENDEZVOUS state, and FOCUS PROCESSOR COMMIT state.

**Run State** The RUN state is defined as the state in which the operating system is normally running. This is the time when the processor is in user and kernel modes, including the time spent in interruptions such as IO interrupts and processor exceptions. It does not include the debug exception handler where CDE will change the state from the RUN state to one of the other three defined states.

**Focus Processor State** When an exception occurs that results in a processor entering the debug exception handler and subsequently CDE, CDE determines whether this is the first processor to come into CDE. If it is the first processor, it becomes the focus processor and its state is changed from the RUN state to the FOCUS PROCESSOR state.

The focus processor controls the machine from this point on, until it yields to another processor or returns to the RUN state.

Once the focus processor has entered CDE, its first responsibility is to rendezvous all other processors in the system before debug operations are allowed by the registered debug agent. Rendezvous operations are typically accomplished by hardware specific methods and may be unique to each processor architecture. On x86, this is typically a cross-processor NMI.

After sending the rendezvous command to all other processors, the focus processor waits for all processors to respond to the request to rendezvous. As these processors come into CDE they are immediately sent to the RENDEZVOUS state where they remain until the focus processor yields control.

Once all processors are safely placed in the RENDEZVOUS state, the focus processor transfers control to the debug agent that was registered with CDE for subsequent control of the system.

**Rendezvous State** The RENDEZVOUS state is sort of a corral or holding pen for processors while a debug agent examines the processor currently in the FOCUS PROCESSOR state. Processors in the RENDEZVOUS state do nothing but await a command to change state or to deliver information about their state to the focus processor.

It should be noted at this point that processors could have entered the debugger for reasons other than being asked to rendezvous. This happens when there are exceptions occurring simultaneously on more than one processor. This is to be expected. A processor could, in fact, receive a rendezvous request just before entering CDE on is own accord. This can result in spurious rendezvous requests that are detected and handled by the state machine. Again, this is normal. These sorts of race conditions are gracefully handled by CDE, such that those processors end up in the RENDEZVOUS state just as any other processor does.

As stated above, a processor may end up in the RENDEZVOUS state when it has a valid exception condition that needs evaluation by the active debug agent. Before ever sending any processor back to the RUN state, CDE examines the reason for which all other processors have entered the debugger. This may result in the processor in the FOCUS PROCESSOR state moving to the RENDEZVOUS state and a processor in the RENDEZVOUS state becoming the focus processor for further examination.

This careful examination of each processor's exception status forces all pending exceptions to be evaluated by the debug agent before allowing any processor to continue execution. This further contributes to the stability of the debugger.

Once all processors have been examined, any processors that have been in the FOCUS PROCESSOR state are moved to the FOCUS PROCESSOR COMMIT state, which we will now discuss.

**Focus Processor Commit State** The logic in this state is potentially the most complex part of CDE. Processors that have been moved to this state may need to adjust the breakpoint state in order to resume execution without re-triggering the breakpoint that caused the debugger to be entered.

The FOCUS PROCESSOR COMMIT state is the state that ensures that no processor is run or is further examined by the debug agents until the conditions specified by CDE are met. This contributes greatly to the stability of the debugger.

### 2.2.2 Breakpoint Logic

A distinguishing feature of NLKD is its rich breakpoint capability. NLKD supports the notion of qualifying breakpoints. Breakpoints can be set to qualify when a number of conditions are met. These conditions are:

- execute/read/write
- address/symbol, optionally with a length
- agent-evaluated condition (e.g. expression)
- global/engine/process/thread
- rings 0, 1, 2, 3
- count

This allows for a number of restrictions to be placed on the breakpoint before it would actually be considered as "qualifying," resulting in the debug agent being invoked.

The number of supported read/write break points is restricted by hardware, while the number of supported execute breakpoints is limited by software and is currently a `#define` in the code. NLKD uses the debug exception, INT3 on x86, for execute breakpoints and the processor's watch/debug registers for read/write breakpoints.

The debug agents work in cooperation with CDE to provide breakpoint capabilities. The debug agents define the conditions that will trigger a debug event and CDE modifies the code with debug patterns (INT3 on x86) as necessary. When the breakpoint occurs, CDE determines if it actually qualifies before calling the debug agent.

CDE's breakpoint logic is one of the most powerful features of the tool and a distinguishing feature of NLKD. CDE's breakpoint logic combined with CDE's state machine sets the stage for a stable on-box or remote debugging experience.

#### 2.2.3 CDE APIs

CDE exports a number of useful APIs. These interfaces allow the rest of the system to interact with the debugger and allow debug agents to be extended with new functionality.

CDE supports an API to perform DWARF2 frame-pointer-less reliable stack unwinding, using the `-fasynchronous-unwind-tables` functionality available with gcc.

The programmatic interfaces to the debugger also include support for various debug events (such as assertions and explicit requests to enter the debugger) and the ability to register and unregister debugger extensions. Extensions can be either loadable binaries or statically linked modules.

APIs also exist to support pluggable debug agents, which we will discuss in the next section.

#### 2.2.4 Debug Agents

In earlier discussions, we briefly introduced the notion of debug agents. Debug agents plug into CDE and provide some sort of interface to the user. Debug agents can be loadable kernel modules or statically linked into the kernel.

NLKD provides two debug agents: the Console Debug Agent (CDA) for on-box kernel debugging, and the Remote Debug Agent (RDA) for remote debugging including remote source level debugging.

Other debug agents can be written and plugged into CDE's framework, thus benefiting from the state logic provided by CDE.

It should be noted that CDE only allows one agent to be active at a time. However, a new agent can be loaded on the fly and replace the currently active one. This scenario commonly happens when a server is being debugged on-site (using CDA), but is then enabled for debugging by a remote support team (using RDA). This is possible by simply unloading CDA and loading RDA.

**Console Debug Agent (CDA)** CDA is NLKD's on-box kernel debugger component. It accepts keyboard input and interacts with the screen to allow users to do on-box kernel debugging.

**Remote Debug Agent (RDA)** RDA is an agent that sits on-box and communicates with a remote debug client. RDA would typically be used by users who want to do remote source level debugging.

**Other Agent Types** It should be noted that NLKD's architecture does not limit itself to the use of these two agents. CDE allows for other agent types to plug in and take advantage of the environment provided by CDE.

NLKD's agents support the ability to apply certain settings to the debug environment before the debugger is initialized. Some examples are a request to break at the earliest possible moment during system boot, or setting screen color preferences for CDA. These configuration settings are held in a file made available early in the boot process, but only if the agent is built into the kernel.

### 2.3 Architecture Summary

At this point we have introduced the exception handling framework and NLKD's architecture, including CDE with its state machine, debug agents, breakpoint logic, and finally NLKD's ability to be extended.

Further discussion of the NLKD will follow but will not be presented as an architectural discussion. The remainder of this discussion will focus on features provided by NLKD and the debug agents CDA and RDA.

## 3 Console Debug Agent (CDA) Features

This section discusses the features of the Console Debug Agent. We should note that this section lists features, but it is not meant to be a user's guide. To see the full user's guide for NLKD, go to `http://forge.novell.com` and then search for "NLKD".

### 3.1 User Interface Overview

CDA supports on-box debugging. Interaction with the debugger is via the keyboard and screen.

#### 3.1.1 Keyboard

Input from PS2 keyboards and the 8042 keyboard controller is currently supported. The debugger can be invoked by a special keystroke when CDA is loaded.

#### 3.1.2 Screen IO

CDA can operate in text or graphics mode. The mode CDA uses is determined by the mode that the kernel switched to during boot.

Since CDA has the ability to display data in graphics mode, we also have the ability to enter the debugger directly from graphics mode at run time. This is extremely useful but requires that the screen resolution and color depth of the user graphics environment match the screen resolution and color depth of the kernel console environment.

#### 3.1.3 Screen Layout

CDA supports both command line and window-pane based debugging. The debugging screen is divided into six window panes as shown in Figure 2. One of these panes hosts a command line. All panes are resizable.

Each pane's features can be accessed via a number of keystroke combinations. These keystrokes are documented in the user's guide, and are also available by pressing F1 while in the debugger. Help can also be obtained by typing `h` in the command line pane.

**Code Pane** The code pane shows instruction disassembly. There are a variety of format specifier commands that can alter the way the information is displayed. Currently CDA supports the Intel assembly format.

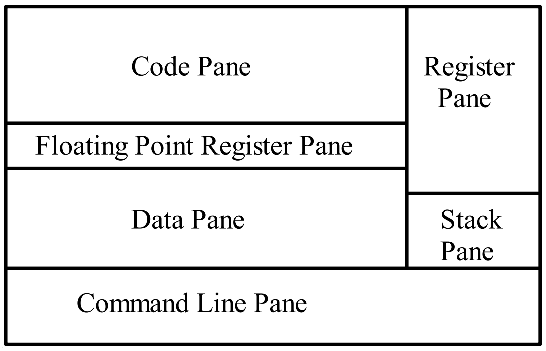

Figure 2: Screen layout of CDA

**Data Pane** The data pane supports the display and modification of logical and physical memory addresses, including IO ports and PCI config space. Data can be displayed in a variety of formats.

**Register Pane** The register pane supports the display and modification of processor registers. Registers with special bitmap definitions are decoded.

On some architectures, IA64 for example, there are too many registers to view all at once in the register pane. Hence, support exists to scroll up or down through the registers in the register pane.

**Stack Pane / Predicate Pane (IA64)** A special pane exists for displaying the stack pointed to by the processor's stack pointer. Since on IA64 the register stack engine is used instead of a normal stack, CDA uses this pane to display the IA64 predicate registers instead of stack information.

Code or data browsing can be initiated directly from the stack.

**Floating Point Register Pane** The floating point register pane supports the display and modification of floating point registers. The data can be displayed in a variety of formats. Since kernel debugging rarely requires access to floating point registers, this pane is normally hidden.

**Command Line Pane** The command line pane supports a command line parser that allows access to most of the capabilities found in other CDA panes. This pane can assume the size of the entire screen, and it can also be entirely hidden.

The command line pane exports APIs so that other modules can further extend the debugger.

## 3.2 CDA User Interface Features

### 3.2.1 Viewing Program Screens

The screen that was active when the debugger was invoked can be viewed from the debugger. Viewing both text and graphics (such as an X session) is supported.

### 3.2.2 Processor Selection

Support to switch processors and view information specific to that processor is supported. Some information is processor specific such as the registers, per processor data, etc. CDA also supports viewing the results of such commands as the CPUID instruction on x86.

### 3.2.3 Command Invocation

There are a number of pane-sensitive hot keys and pane-sensitive context menus available for command execution. Additionally, there is a global context menu for commands common to all panes.

### 3.2.4 Expression Evaluation

Support for expression evaluation exists. The expressions use mostly C-style operators, operating on symbolic or numeric addresses.

### 3.2.5 Browsing

The code, data, and stack panes support the ability to browse. For example, in the code pane we can browse (follow) branch instructions, including call and return statements.

In the data pane we can follow data as either code pointers or data pointers. The same is true for the register and stack panes.

Functionality exists to follow a pointer, to go to the previously visited location, or to go to the place of origin where we started.

Of course, we can always browse to a specific code or data address.

### 3.2.6 Stepping

CDA supports typical processor stepping capabilities.

- Single Step
- Branch Step
- Step Over
- Step Out
- Continue Execution (Go)

### 3.2.7 Symbols

Provided that symbols are available from the kernel, support for symbolic debugging is supported throughout the debugger.

### 3.2.8 Smart Register Mode

A mode exists to make it easier to watch only the registers that change as the code is stepped through. This is particularly useful on architectures like IA64 that have many more registers that we can display at once.

### 3.2.9 Aliases

Aliases are supported in the code and register panes. For example, on IA64 a register named `cr12` may also be a the stack pointer. With aliasing off, the name `cr12` will be displayed everywhere. With aliasing on, `sp` will be displayed.

### 3.2.10 Special Registers

Support exists for viewing (and in some cases, setting) special processor and platform registers. On x86, these are:

- CPUID
- Debug Registers
- Data Translation Registers
- Last Branch
- MSR
- MTTR

### 3.2.11 Debugger Events

NLKD has a level of interaction with the host operating system, enabling certain operating system events or code to invoke the debugger.

APIs exist to generate debug messages from source code. When a message event is hit, CDA displays the message on the screen. The user then enters a keystroke to resume execution.

CDA also can notify the user when a kernel module is loaded or unloaded, as well as when a thread is created or destroyed.

The user may enable and disable each event notification type at run-time.

On a per processor basis, CDA can display the most recent event that caused the debugger to be invoked.

#### 3.2.12 OS Structures

CDA understands certain Linux kernel data structures. In particular, CDA can list all threads in the system. It also can list all kernel modules that are currently loaded, and information about them.

#### 3.2.13 Linked List Browsing

A useful feature in the data pane is the ability for the user to inform the debugger which offsets from the current address are forward and backward pointers. This features enables users to easily browse singly and doubly linked lists.

#### 3.2.14 Set Display Format

CDA is very flexible with how data is displayed.

**Code Pane** The code pane allows a variety of formating options.

- Show Address
- Show Aliases
- Show NOPs (Mainly used for IA64 templates.)
- Set Opcode Display Width
- Display Symbolic Information
- Show Templates/Bundles (IA64)

**Data Pane** Data sizing, format, and radix can be set. CDA provides support for displaying or operating on logical and physical memory.

**Floating Point Pane** Data sizing and format can be set.

#### 3.2.15 Search

CDA can search for a set of bytes in memory. The search can start at any location in memory, and can be restricted to a range. Both forward and backward searches are supported.

#### 3.2.16 List PCI Devices

On architectures that support PCI buses, CDA has knowledge of the PCI config space and allows browsing of the config space of all the devices.

#### 3.2.17 System Reboot

NLKD can perform a warm and/or cold boot. Cold boot is dependent upon hardware support.

## 4 Remote Debug Agent (RDA) Features

RDA provides a means whereby a remote debugger can communicate with NLKD breakpoint logic and drive NLKD's abilities to support debugging.

As expected, there are a number of verbs and events supported by RDA that enable remote source level debuggers to drive the system. Some examples include setting and clearing breakpoints, controlling processors' execution, and reading and writing registers and memory.

### 4.1 Protocol Support

RDA currently supports the gdbtransport protocol.

Novell has an additional protocol, Novell Debug Interface (NDI) 2.0, which we hope to introduce into a new remote debug agent. NDI provides advantages in three areas:

- NDI has support for describing multiprocessor configurations.
- When debugging architectures with a large register set, NDI allows the debugger to transfer only a portion of the full register set. This is especially important when debugging over slow serial lines.
- NDI fully supports control registers, model specific registers, and indirect or architecture-specific registers.

### 4.2 Wire Protocols

RDA currently supports RS232 serial port connections to the debugger.

## 5 Work To Do

Currently, a number of things still need to be worked on. We welcome help from interested and capable persons in these areas.

### 5.1 USB Keyboards

Support for other keyboard types, mainly USB, will be added to CDA.

### 5.2 AT&T Assembler Format

CDA currently supports the Intel assembler format. We would like to add support for the AT&T assembler format.

### 5.3 Additional Command Line Parsers

There is a native command line interface provided by CDA. We would also like to see a KDB command line parser and a GDB command line parser for those familiar with these debuggers and who prefer command line debugging to pane-based debugging.

### 5.4 Novell Debug Interface 2.0

We plan to create a new debug agent supporting the NDI 2.0 protocol. We also need support for NDI in a remote source level debugger.

### 5.5 Additional Wire Protocols

We would like to support additional wire protocols for remote debugging, such as LAN, IPMI/BMC LAN, and USB.

### 5.6 Additional Architectures

We would like to finish the NLKD port to IA64 Linux, and port it to Power PC Linux.

## 6 Conclusion

We recognize that no one tool fits all, and that user familiarity with a tool often dictates what is used in spite of the existence of tools that may offer more features and capabilities. In introducing this tool, we make no claim to superiority over existing tools. Each and every user will make that decision themselves.

As we stated earlier, our goal is to provide developers and support engineers a robust kernel development tool enabling them to be successful in their respective roles. We believe we have introduced a debug architecture that needs no apology. At the same time, we welcome input as to how to improve upon this foundation.

Our hope is that kernel debugging with this architecture will become standard and that the capabilities of NLKD will find broad acceptance with the goal of creating a better Linux for everyone.

## 7 Acknowledgments

We would like to express thanks to Charles Coffing and Jana Griffin for proofreading this paper, and to Charles for typesetting.

## 8 References

Data Center Linux: DCL Goals and Capabilities Version 1.1
`http://www.osdl.org/lab_activities/data_center_linux/DCL_Goals_Capabilities_1.1.pdf`

Novell Linux Kernel Debugger (NLKD)
`http://forge.novell.com`

Built-in Kernel Debugger (KDB)
`http://oss.sgi.com/projects/kdb/`

GNU Project Debugger (GDB)
`http://www.gnu.org/software/gdb/gdb.html`