[incr Insight]

An easy to use, easy to extend module system for Insight/GDB

Brandon Diamond
Brown University

1. Introduction

1.1. Motivation

As part of the departmental Operating Systems course and laboratory, computer science concentrators implement critical pieces of a toy operating system named **Weenix**. At nearly 45,000 lines of code, Weenix represents the first large-scale software engineering project that many students will contribute to. That a number of the concepts underlying Weenix are themselves particularly complex only serves to exacerbate student's difficulties with this project. In fact, many students quip that the course is named "CS169" because it takes that many hours of work each week just to keep afloat. Though these students are clearly exaggerating (at least to some degree), it is indeed true that a substantial time commitment is necessary if one has any hope of completing Weenix by the deadline.

Fortunately, as with any large software development project, various tools can be leveraged to allay the added complexity of dealing with tens of thousands of lines of source code. At present, however, Weenix makes use of just three of them: Cscope, GNU Make, and the GNU debugger (GDB). That the next generation of Weenix has been rewritten to run atop the Xen hypervisor makes the problem all the more urgent: Xen

guests must deal in a number of low level paravirtualization constructs that GDB isn't well suited, by default, to deal with. Debugging a system running on Xen requires a detailed understanding of the mechanisms underpinning the hypervisor; as students will only typically have an imperfect knowledge of Xen, debugging low-level kernel issues by hand will be extremely challenging. With more powerful tools, however, students gain the ability to interact with systems concepts on an intuitive level: it is much easier to understand the process list as it unfurls before you than it is to print out two or three dozen variables that, when taken together, might reveal a slight problem.

The benefits of code visualization and management tools are numerous and well explored; but which tools would help most with a system like Weenix? One of the reasons that operating systems form so compelling a field of study is that there is a great breadth of interesting problems to solve, each often lying at the junction of one or more diverse disciplines. As a result, there is no practical limit to the number of development aides that might help students to fully understand the concepts at play within Weenix; after all, depicting the full gamut of operating systems theory intuitively would require an unreasonable number of highly specialized utilities.

It seems reasonable, then, that a general framework providing for the implementation of custom utilities and visualizations would be far more helpful aide than any one specialized tool on its own. It is this observation that lies at the heart of the computer software produced as part of this thesis; [incr Insight] provides an intuitive and extremely flexible development environment allowing students to build,

test, and experiment with tools for exploring Weenix as it runs.

1.2. **The Xen Hypervisor**

Before proceeding, we shall touch upon a small amount of background information regarding the Xen hypervisor, a free software paravirtualized virtual machine monitor, developed by XenSource, inc. Much like other virtualization packages such as those offered by VMware and Parallels, Xen abstracts away many hardware specific details to allow the user to multiplex a single machine as any number of independent *virtual* machines. Software that achieves this task is known as a virtual machine monitor or, alternatively, a hypervisor. Typically, a hypervisor replaces the operating system kernel in r*ing 0*, moving the guest kernel to *ring 1* and user code to r*ing 3* ("rings" are the IA32 mechanism for achieving different system privilege levels; typically, there are four rings decreasing in privilege from ring 0 down to ring 3).

While many popular hypervisors perform full virtualization in the sense that guest operating systems running within a virtual machine need not be modified to support virtualization, Xen utilizes a technique called *paravirtualization* requiring guest operating systems to be modified in order to be virtualized using Xen. While this might at first sound like a disadvantage, there are a great many reasons why paravirtualization might make more sense than full virtualization. For one, embracing the presence of a hypervisor allows operating systems to interact with the virtual hardware on a more direct level—it is no longer necessary to transform data representation multiple times to achieve the illusion that the virtual machine has a

certain peripheral or device that the real machine does not; a paravirtualized operating system allows the guest to interact directly with virtualization primitives and inter-virtual machine channels, eliminating the need for phony device drivers and inefficient device emulation. Furthermore, paravirtualization eliminates the need to rely upon a number of fairly inefficient virtualization techniques such as binary rewriting that stem from an inherent inability to virtualize the x86 architecture directly: due to a handful of sensitive instructions that aren't consistently trapped by the processor, preserving the integrity of the virtual machine abstraction becomes exponentially more difficult. Paravirtualization replaces these sensitive instructions with explicit *hypercalls* that allow problematic instructions to be appropriately handled by the hypervisor.

Each virtual machine running under the Xen hypervisor is called a *domain* and is assigned a unique ID. Furthermore, exactly one domain is attributed special privileges and is named *Domain-0* or *Dom0*. All remaining domains are simply referred to by their unique IDs and may be referred to as user domains or *DomU.* This centralization of privilege forms a cornerstone of the Xen philosophy: the hypervisor itself delegates the task of running the physical machine to the Dom0 kernel. In this way, Xen's job is restricted to booting the physical machine and creating the virtual machine abstraction, keeping critical regions of code short and easily maintainable. The end result of this is that Dom0 serves as something of a user interface to the Xen hypervisor via a number of userland utilities including the *xend* daemon and the XenStore device. Thus, creating and managing user domains is performed from within what appears to be an ordinary Linux system; in reality, these interactions are implemented as hypercalls issued by a

paravirtualized version of the Linux kernel which is itself running as a Domain0 guest under the Xen hypervisor.

In this context, Weenix on Xen (Wox) runs as a DomU domain on a physical machine running a patched version of Linux as Dom0. Devices are implemented using standard Xen constructs for exchanging data between different virtual machines; the block device, for instance, is implemented using a split-driver pattern that allows Weenix to utilize an extremely simple abstract device. The other end of this split-driver is connected by Xen to a driver in Dom0 that effects the desired actions to the real disk. Fortunately, the extent to which Weenix has been changed to support Xen is largely abstracted from the student's view. Those details that do peek through, however, are particularly difficult to debug using GDB alone; [incr Insight] was designed to help illuminate these new dark corners.

1.3. **Overview**

[incr Insight] is an unofficially extension of the open source GDB frontend, Insight, itself maintained by Red Hat inc. [incr Insight] extends the original Insight functionality by providing a flexible and seamless interface for developing and using user defined "plugins" that have direct as well as managed access to GDB, and thus, the target program itself. Insight is an amalgamation of Tcl/Tk (Tool Command Language / GUI Toolkit) and the GDB codebase: rather than merely wrapping a running instance of GDB, Insight is built directly atop GDB itself. This allows Insight to have direct access to GDB's internal state and data structures, providing a great deal of flexibility that would

be otherwise lost.

[incr Insight], which is itself built directly into Insight, takes the place of GDB in the traditional Weenix workflow. All prior effort tying GDB to the Weenix build infrastructure is preserved as Insight exports a superset of GDB's core functionality. Importantly, this means that none of the GDB scripts that had been carefully designed to allow for the remote debugging of Weenix on Xen (Wox) will need to be altered. This is fortunate as complicating this system further would only hinder the student's ability

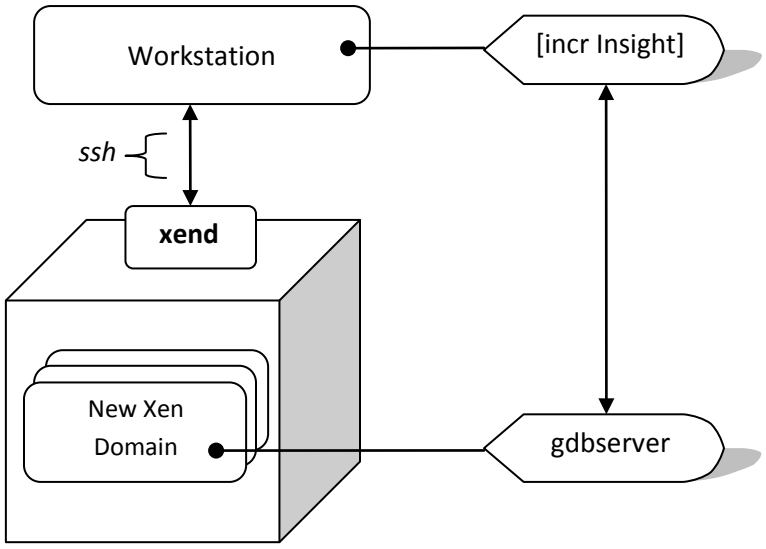

Figure 1: Debugging Weenix on Xen (Wox) remotely using [incr Insight]

to embrace the big picture.

Presently, launching Wox with [incr Insight] support is as simple as executing a single "`make gdb`" command. The Makefile constructs this target by issuing a call via ssh to the Xen server requesting that a new Xen domain be created. This also results in a fresh instance of gdbserver being started on a port number that is subsequently

conveyed back to the user's local machine. With the domain running and the gdbserver listening for a connection, the Makefile finally invokes [incr Insight] with the remote debugging target specified appropriately (figure 1).

Once [incr Insight] has successfully connected to the remote gdbserver, the user can begin debugging and exploring Weenix. All plugins are loaded automatically based on a per-user configuration file and are dynamically reloaded in the event of code modification (this feature allows for fast turnover during the implementation of the plugins themselves). Like Insight, plugins are implemented in Tcl, though the user is free to use any additional libraries as he or she sees fit. A number of the demo plugins, for instance, utilize extensions to Tk in order to draw more complex widgets than are shipped standard. Most if not all of the operational complexities of Insight and GDB are removed by means of a single root interaction object complete with an integrated help system. Importantly, users wishing merely to leverage the plugins that are already available need not concern themselves with the details of plugin implementation: plugins are launched by clicking the appropriate icon on the launch dock (a toolbar found at the top of the [incr Insight] root window) or by selecting the appropriate option from the "Weenix" drop-down menu.

Plugins can be designed in two major styles: as *static* plugins that are only updated as the user places his or her own break points, or as *dynamic* plugins that are updated while Weenix is running. A static plugin might be useful for visualizing state while the user steps through a program while a dynamic plugin allows the user to watch as data structures evolve over time. Importantly, dynamic plugins require no

additional knowledge to implement and can be realized simply by subscribing to locations within the target's code.

Finally, as a consequence of careful attention to usability, interacting with [incr Insight] as a GDB interface is extremely straightforward. All of the familiar debugging operations are available (stepping, continuing, disassembling, and so forth) with a great number of additional features built directly into the main window. Beyond these standard operations, a number of improvements to Insight were implemented as part of development of [incr Insight] most notably including an enhanced source view widget that performs syntax highlighting of source listings on the fly. As each of these features are integrated together with great consistency, [incr Insight] should come naturally to any student familiar with GDB and even more naturally to students familiar with other graphical debuggers such as those found in Eclipse as well as Microsoft's Visual Studio.

2. Implementation

2.1. **Tools and Background**

[incr Insight] is a code fork of Red Hat Insight, itself a version of GDB with an integrated Tcl interpreter and software interface to the underlying debugger core. The standard Insight GUI is implemented as a set of [incr Tcl] classes, where [incr Tcl] is a reasonably common object system that comes packaged with its own set of extensible widgets. GDB interaction is facilitated via a complex system of hooks and source modifications to GDB itself; a number of rough-edged Tcl procedures are exported by the C interface for use by the GUI. Most of these commands, however, are not

intended for general consumption and so a considerable deal of packaging is required to render these appropriate for a general audience. Notably, a large portion of developing [incr Insight] involved the isolation and packaging of these lower level commands into an intuitive and extensible API, documented below.

The source code modifications comprising [incr Insight] were realized as direct modifications to the some 25,000 lines of Tcl code implementing the standard Insight interface. A small amount of code wound up in the C source files implementing the GDB Tcl interface and an even smaller amount of code wound up in the extended GDB sources. These modifications were typically a result of bug hunting, though at least one feature was made feasible by modifying GDB's variable object implementation, itself.

```
Root    { $env(HOME)/insight }
plugins {
        refresh {
                {GUI refresh (bdiamond)}
                {Refresh the GUI (including plugins)}
                true
        }

        sharedinfo {
                {Shared info page (billg)}
                {View the Xen shared_info page}
                true
        }

        ptable {
                {Page table browser (stevejobs)}
                {Easily walk through the page tables}
                true
        }

        dataview {
                {Data structure visualizer (linus)}
                {Explore complex data structures visually}
                true
        }
```

Listing 1: Example configuration file

2.2. **Configuration**

[incr Insight] allows each user to customize his or her own selection of enabled plugins using a single configuration file stored in the user's home directory named "`.insight`". This file is implemented as a Tcl list structure, as depicted in listing 1.

At runtime, [incr Insight] parses the user's configuration file and attempts to load each plugin referenced therein. As it does so, [incr Insight] constructs internal data structures that facilitate the plug-in namespace system as well as other inward- and outward- facing subsystems. Based on the root plugin directory specified in the configuration file, [incr Insight] validates that each plugin path is accessible. Plugin paths are derived by concatenating the root path and each plugin's unique ID (the ID is a Tcl identifier specified just before the opening braces in the plugins section of the configuration file). Though plugins are free to utilize the plugin directory for any sort of source code or resource files, the plugin directory must minimally contain a single "`main.tcl`" file defining the plugin's mainline command, "`plugin_main`". If defined, the plugin's "`plugin_init`" command will be invoked to allow any source dependencies to be resolved and to initiate any required preprocessing or setup. While the plugins are being loaded, the plugin ID, title, and description along with any output produced by the "`plugin_init`" command are displayed to the terminal. Should a plugin fail to load, color-keyed errors will appear in the terminal, as well. Note that [incr Insight] will attempt to continue launching the main GUI even in the event of multiple plugin failures. Should the plugin become fixed, it can be loaded separately while [incr Insight] is already running.

A useful convenience afforded by the plugin loader is a dynamic reloading system for detecting changes in any of the installed and enabled plugins. Should the modification timestamp of a particular plugin ever become newer than the timestamp at the time the plugin was most recently loaded, [incr Insight] will automatically reload and reconfigure that plugin. Code for reloading all plugins in the event of a configuration file update has been implemented, though this is not enabled by default.

Once each plugin has been loaded, entries in the standard Insight tool panel are created with either the default plugin icon, or the contents of "`icon.gif`" if present in the plugin's path. Finally, a Weenix menu is populated at the top of the main Insight window to provide a second access path to the loaded plugins. Tooltips are created for each of the toolbar buttons that contain the plugin description as extracted from the main [incr Insight] configuration file.

2.3. **Plugin Requirements**

Great effort has been made to avoid imposing many requirements on the plugin author. At a minimum, the "`plugin_main`" command must be defined; the rest is entirely optional. The "`plugin_main`" command is invoked whenever the user selects the plugin from the toolbar or Weenix menu. Typically, this command is responsible for building the plugin's GUI to be updated later as the user steps through Weenix. Additional detail on this process will be discussed along with the standard [incr Insight] event loop, below.

There are only two additional "special" [incr Insight] plugin commands. The first,

“`plugin_init`”, was touched on a bit earlier and serves the sole purpose of performing any plugin-specific initialization that might be necessary. Should this command return false, the plugin will fail to load and an error will be reported to the user. The next special command, “`plugin_update`”, is invoked whenever the GUI is updated or Weenix’s state changes in a predictable way (i.e., changes to volatile memory by external programs do not trigger an update). Typically, the update function is responsible for updating the GUI to reflect current program state.

The rest of the plugin’s implementation is left entirely to the plugin author. A transparent module system is overlaid atop each plugin to ensure that each plugin is associated with its own local namespace without restricting access to the complete spectrum of functionality present within [incr Insight]’s various modules. This eliminates the possibility of conflicting with other installed plugins in [incr Insight].

Finally, various plugin-local functionality is encapsulated by the “`weenix plugin`” command; this command can be used for everything from establishing plugin local data, to constructing event callbacks to be executed within the context of that particular plugin’s execution environment; a full listing is provided in the references section, below. Access to the main GUI as well as to the debugger itself is achieved using different subcommands of “`weenix`”. The goal of these abstractions is to allow the plugin author to avoid dealing with the complexity of Insight (and GDB) implementation details, and to focus on crafting plugins and visualizations that are as lightweight and simple as possible. In fact, a successful plugin can be implemented with only a rudimentary understanding of Tcl (itself an extremely simple language) and small

bit of Tk, as shall be demonstrated below.

2.4. **Event Loop**

[incr Insight] ties into the main Insight event loop so as to ensure that plugins are refreshed appropriately as program state changes. Plugin authors enable notifications using the "`weenix plugin active`" command. This command causes [incr Insight] to invoke the calling plugin's "`plugin_update`" command at every GDB and Insight update cycle. Subscription can be disabled at any point using the same command; typically, a handler is installed on the plugin's top-level window's close event such that resources aren't wasted updating an inactive widget. Note that a plugin needn't define an update command if the plugin is not intended to receive update notifications. Similarly, there is no requirement that a plugin unsubscribe from notifications before the program terminates. Again, the author is free to use the event loop to his or her best advantage.

2.5. **Dynamic Plugins**

Some plugins may require more fine-grained access to internal state than is provided by the simple update event loop. For instance, since "`plugin_update`" is only ever invoked once per GDB state chance, transitioning from running to paused within [incr Insight] results in only a single update. Should a plugin designer wish to depict the evolution of data structures over the course of an uninterrupted run cycle, a different mechanism must be utilized.

A technique for achieving this result is realized in an easy-to-use notification

mechanism allowing plugins to subscribe to particular locations within the code, thereby causing GDB to temporarily suspend execution and granting the plugin an opportunity to update internal state. If the user did not indicate that he or she would like GDB to break on the same location as one or more plugins are subscribed to, [incr Insight] will automatically resume execution of Weenix after all subscribed plugins have had a chance to collect the necessary data.

As is the case with the coarse-grained event loop described above, only active plugins will ever be notified; this retains a consistent interface for enabling and disabling a plugin in its entirety. Access to this mechanism is exclusively provided via two weenix commands "`weenix plugin subscribe`" and "`weenix plugin unsubscribe`", each accepting a source file name and a line number to subscribe to.

2.6. **The Weenix Command**

The only necessary point of contact between a plugin and GDB or the standard Insight GUI is through the root of the [incr Insight] API: "`weenix`". This command contains a number of command modes which themselves provide a number of commands and subcommands. Usefully, the weenix command incorporates a form of light-weight interactive documentation: typing an invalid mode will cause the command to list all valid modes and subcommands; similarly, typing an invalid option or command will yield a brief summary of the requested operation as well as a list of all accepted options along with any default values. A useful technique for starting out with the weenix command is to open up the [incr Insight] console and, using the "`tk`" hook,

try out each of the weenix modes and subcommands interactively.

Calls to the weenix command tend to resemble short, semi-grammatical sentences. For instance, "`weenix gui prompt ask {Are we there yet?}`" will raise a modal dialog prompting the user to click "yes" or "no." While somewhat verbose, discerning the meaning of a particular weenix command invocation ought to never be difficult; the weenix command was designed to provide complete access to the [incr Insight] API in a consistent and easy-to-use way.

The weenix command is divided into four standard modes: gdb, gui, plugin, and helper. The gdb mode provides comprehensive access to the underlying GDB session, while the gui mode allows plugins to alter the state of the main [incr Insight] GUI. The plugin mode, as touched on earlier, facilitates the transparent plugin namespace mechanism, whereas the helper mode organizes common functionality in a central place (for instance, a general expression iterator can be obtained using the "`weenix helper tree iterator`" command). For complete details on the weenix command, please consult the reference section, below.

2.7. **GDB Expressions**

Perhaps the most useful subcommands are those used to evaluate and interact with arbitrary GDB expressions. For maximum flexibility, there are at least six different commands available to this end; we shall explore the four most powerful of these in depth for the remainder of this subsection: raw, lazy, tree, and object. Each of these commands accepts a string containing an arbitrary GDB expression such as "`*curproc`"

or “PFN_TO_SEG(`p_table->thr_src[3]) + 0xcafebabe`”.

2.7.1. weenix gdb expression raw

The raw expression subcommand provides direct access to GDB’ s expression evaluation output; it is also the simplest GDB expression evaluator included in the [incr Insight] API. This command returns nearly the same string as would result from typing “p *expression*” directly into GDB. However, rather than returning the full unaltered output, this command strips away everything other than the *value* of the provided expression. As an example, history variable assignments such as “$1 = 53326” will never appear in the output of the raw evaluator; this string would be reduced to “53326” before being returned to the user.

2.7.2. weenix gdb expression lazy

The lazy expression evaluator is included to help optimize and simplify code that frequently accesses values in Weenix by way of the raw expression evaluator. Use of the lazy evaluator is suitable for evaluating expressions that will not change while execution of Weenix is paused; that is, volatile expressions should not be computed using the lazy evaluator. That said, the lazy evaluator is ideal for extracting symbolic values for use in recursive or iterative plugin commands; the value of the expression will be cached between invocations of the command and automatically refreshed whenever the debugger is stepped or an update event is signaled.

The lazy evaluator features something of a poor-man's generational garbage collector; expressions that aren't read after a certain number of updates will automatically be purged from the cache. Additionally, if an expression is referenced by multiple plugins, [incr Insight] will automatically merge the entry in the cache to avoid excessive computation or storage. The collector takes into account the number of references to a cached expression when determining whether or not to purge an entry.

```
(gdb) p *HYPERVISOR_shared_info

$5 = {vcpu_info = {{evtchn_upcall_pending = 0 '\0', evtchn_upcall_mask = 0 '\0',
evtchn_pending_sel = 0, arch = {cr2 = 134529024, pad = {0, 0, 0, 0, 0}}, time = {version = 10,
pad0 = 0, tsc_timestamp = 1076660350294586, system_time = 538253562870835, tsc_to_system_mul =
4294824532, tsc_shift = -1 '', pad1 = "\000\000"}}, {evtchn_upcall_pending = 0 '\0',
evtchn_upcall_mask = 1 '\001', evtchn_pending_sel = 0, arch = {cr2 = 0, pad = {0, 0, 0, 0,
0}}, time = {version = 0, pad0 = 0,

...

(gdb) tk weenix gui console echo [weenix gdb expression tree *HYPERVISOR_shared_info]

{array vcpu_info {list {array evtchn_upcall_pending {value 0 {'\0'}} evtchn_upcall_mask {value
0 {'\0'}} evtchn_pending_sel {value 0} arch {array cr2 {value 134529024} pad {list {value 0}
{value 0} {value 0} {value 0} {value 0}}} time {array version {value 10} pad0 {value 0}
tsc_timestamp {value 1076660350294586} system_time {value 538253562870835} tsc_to_system_mul
{value 4294824532} tsc_shift {value 1 ''} pad1 {value {"\000\000"}}}} {repeat 31 {array
evtchn_upcall_pending {value 0 {'\0'}} evtchn_upcall_mask {value 1 {'\001'}}
evtchn_pending_sel {value 0} arch {array cr2 {value 0} pad {list {value 0} {value 0} {value 0}
{value 0} {value 0}}} time {array version {value 0} pad0,

...
```

Listing 2: Parsing a complex GDB expression

2.7.3. weenix gdb expression tree

The tree evaluator provides the second most direct means of interacting with complex GDB expression output. Rather than returning the result of the output expression as a raw string, the tree evaluator utilize parses the output to distill the

string into a simple Tcl parse tree that can subsequently be iterated over directly or flattened for linear iteration using the tree iterator helper command described earlier. Listing 2 depicts the result of applying the parser to a complex GDB expression.

Though still fairly complex in appearance, the result of parsing the GDB expression is far more easily manipulated in Tcl; in particular, the parse tree forms a valid Tcl list, allowing the standard Tcl list manipulation functions to be applied in interpreting the expression tree. Moreover, each subexpression has been classified as an [associative] array, list, value, or repetition; should the included tree iterator be insufficient for a user's purposes, implementing a recursive exploration procedure based on this tree is straightforward, if not trivial.

2.7.4. `weenix gdb expression object`

The object evaluation command is both the most powerful and efficient way of interacting with GDB expressions. Expression objects (or *variable objects*, as they are known internally) present a simple pseudo object oriented interface for manipulating GDB expressions. Importantly, expression objects allow plugins to interact with program expressions as internal GDB objects: instead of parsing GDB output, expression objects wrap GDB's *actual* variable representations.

Executing an expression object command yields yet another command that can be used to explore the original expression. It is helpful to think of this derivative command as an object presenting an interface for querying various properties of a

particular expression as well as altering properties related to the expression (i.e., if an expression references a structure in memory, it is possible to alter fields within that structure).

Each expression object accepts the following subcommands: “value,” “name”, “type”, “numChildren”, “format”, “delete”, “editable”, “update” and “children.” Implementing plugins to explore structures and complex data types using expression objects is a cinch: the dataview demo plugin is implemented using expression objects and pure Tcl/Tk in fewer than 300 lines of code.

3. Building the XenInfo Plugin

3.1. Getting Started

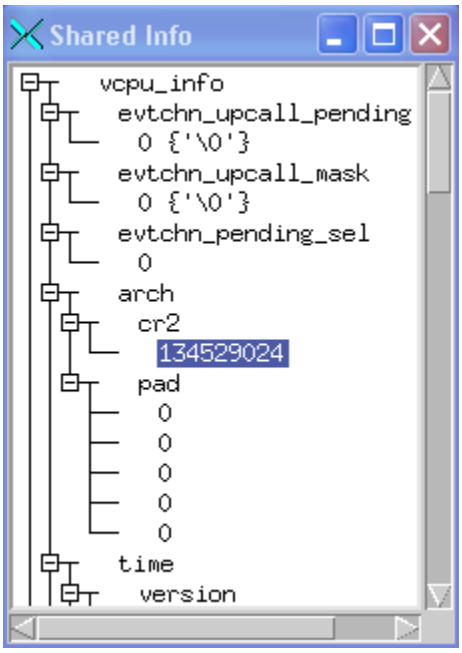

Figure 2: The finished XenInfo plugin

In the following section, a useful [incr Insight] plugin will be built from the ground up. The goal of this plugin will be to provide quick access to the Xen shared info page (the address of which is stored in the "`HYPERVISOR_shared_info`" symbol). This page provides a number of useful details as to internal state of the virtual machine as well as the hypervisor, itself. Figure 2 illustrates what the final plugin should look like.

To begin, create a new directory in your root [incr Insight] plugin directory named "xeninfo." Feel free to place a custom icon named "`icon.gif`" in the folder along with an empty "`main.tcl`" file. Next, place an entry in your main configuration file so that [incr Insight] will load your plugin next time that it's run:

```
xeninfo {
   {Shared info page (nobody)}
   {View the Xen shared_info page}
   true
}
```

With the configuration file set to go, navigate to your Weenix working directory and execute "make gdb". This will launch up the [incr Insight] GUI and attach to the remote Weenix domain. Once the GUI appears, ensure that a new button has been placed on the main toolbar and that the console output indicates that the xeninfo plugin was loaded successfully:

```
xeninfo:
        Shared info page (bdiamond)
        View the Xen shared_info page (Enabled)

         * Init: Success (ok)
```

As we proceed through the next several steps, note that Insight will automatically reload your plugin after every update; this will allow you to iteratively test your plugin and ensure that everything works as expected prior to every line being

written.

3.2. **Creating the GUI**

The first step towards creating the xeninfo plugin will be to implement its graphical user interface using Tk. As we'd like to display the shared info page in a tree and Tk doesn't provide a tree widget by default, we'll need to include a library that implements one: `BWidget`. To ensure that this library has been loaded by the time the plugin is first invoked, we place the following code in "`plugin_init`":

```
proc plugin_init {} {
   global auto_path
   lappend auto_path /home/nobody/libraries/BWidget-1.8.0
   package require BWidget

   return ok
}
```

With `BWidget` imported, it's time to prepare our GUI. As we'd only like the GUI to appear when either the XenInfo button clicked or the Weenix menu item is selected, we place our GUI layout code in the "`plugin_main`" command. As we saw earlier, this command is invoked exactly in the two cases we've specified.

```
 1:  proc plugin_main {} {
 2:    weenix plugin active true

 3:    set this [weenix plugin id]

 4:    toplevel .$this
 5:    wm title .$this "Shared Info"
 6:    wm protocol .$this WM_DELETE_WINDOW \
 7:       [weenix plugin callback {weenix plugin active false; destroy .$this }]

 8:    Tree .$this.t -bg white -takefocus 1 \
                  -xscrollcommand [list .$this.xscroll set]
                  -yscrollcommand [list .$this.yscroll set]

 9:    scrollbar .$this.xscroll -orient horizontal -command [list .$this.t xview]
10:    scrollbar .$this.yscroll -orient vertical -command [list .$this.t yview]

11:    grid .$this.t .$this.yscroll -sticky news
12:    grid .$this.xscroll -sticky ew
13:    grid rowconfigure .$this 0 -weight 1
14:    grid columnconfigure .$this 0 -weight 1

15:    weenix plugin set tree .$this.t
16:    plugin_draw .$this.t
17: }
```

Let's work through the above code line-by-line. In line 2, we're notifying [incr Insight] that we'd like to be notified every time state changes; this will cause [incr Insight] to invoke our "`plugin_update`" command once per event signal. In line 3, we obtain our plugin's unique identifier such that we can give our Tk root window a name that's guaranteed to be unique across all plugins that respect this convention. Lines 4-7 create a root window and ensure that [incr Insight] will no longer update the plugin once the root window has been closed. Note that the callback provided to the "`WM_DELETE_WINDOW`" event is wrapped in a "`weenix plugin callback`" invocation. This command returns its argument wrapped in such a way as to ensure that when it is evaluated, the callback procedure will have access to the XenInfo plugin's local environment. Forgetting to wrap a callback will likely cause Tcl to behave as though the callback function has not been defined.

Next, Lines 8-10 draw the tree itself (as well as horizontal and vertical scrollbars),

and lines 11-14 place the widgets into a grid. These commands are standard Tk and should be readily accessible after spending ten to fifteen minutes with a decent Tk tutorial or textbook. Line 15 places the tree widget in plugin-local storage such that it can be accessed by the "`plugin_update`" command whenever an update signal arises. Finally, the main command executes the "`plugin_draw`" command, causing the tree to be populated with data from within Weenix; note that it is not required that "`plugin_draw`" begin with "`plugin`".

3.3. **Drawing the Tree**

The XenInfo "`plugin_update`" command is perhaps the simplest in the entire plugin, only serving to invoke the "`plugin_draw`" procedure (note that we are here using the "`weenix plugin get`" command to access plugin-local data):

```
proc plugin_update {} {
   plugin_draw [weenix plugin get tree]
}
```

The last command we need to write is the "`plugin_draw`" command which populates the tree widget using data extracted from GDB:

```
1: proc plugin_draw {tree} {
2:    set root [weenix gdb expression tree {*HYPERVISOR_shared_info}]
3:    set iterator [weenix helper tree iterator $root]

4:    $tree delete [$tree nodes root]

5:    foreach {id parent value} $iterator {
6:       $tree insert $id $parent $id -text $value
7:       $tree opentree $id
8:    }
9: }
```

In line 2, we evaluate the expression "`*HYPERVISOR_shared_info`" and obtain a parse tree containing the result. Rather than recursively walking this tree, we invoke the tree iterator helper procedure which returns a flattened representation of the tree ideally suited for direct placement into the tree widget (line 3). In line 4, we ensure that the tree widget is empty and we begin iterating over the parse tree in line 5. The iterator data structure returned by "`weenix helper tree iterator`" is a list in which every three elements represent a node's id, parent id, and value. The "`foreach`" Tcl command allows us to easily step through this list by threes, extracting the relevant fields as we go. In line 6, we insert the item into the `BWidget` tree and in line 7 we expand the node for immediate viewing.

With this code written, we have completed the XenInfo plugin in about 50 lines of simple Tcl. Saving the source file and refreshing the GUI (using the refresh plugin) will cause [incr Insight] to reload the XenInfo plugin; clicking the button in the taskbar or selecting the "XenInfo" item in the Weenix menu should open a new window containing the shared info tree depicted in figure 2.

4. Included Plugins

4.1. Overview

[incr Insight] currently includes five example plugins: **refresh**, **xeninfo**, **ptable**, **dataview**, and **disk**. Each of these plugins provides a useful view into Weenix internals as well as a concrete example of what can be achieved using the [incr Insight] plugin framework. Each example plugin will be presented with a short explanation as well as a

brief note on implementation.

4.2. **Plugins**

4.2.1. Refresh 

Refresh provides an easy way to reload all plugins and to ensure that each is at the latest version. While it is particularly useful when implementing custom plugins, refresh can also come in handy while inspect volatile data structures. Refresh is succinctly implemented using a single call to "`weenix gui update`".

4.2.2. XenInfo 

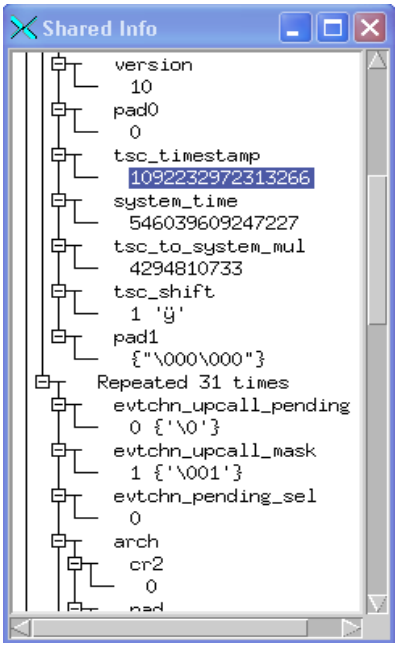

Figure 3: The XenInfo plugin

XenInfo provides instant access to Xen's shared info page. The implementation of the XenInfo plugin was the subject of section three, above.

4.2.3. Dataview

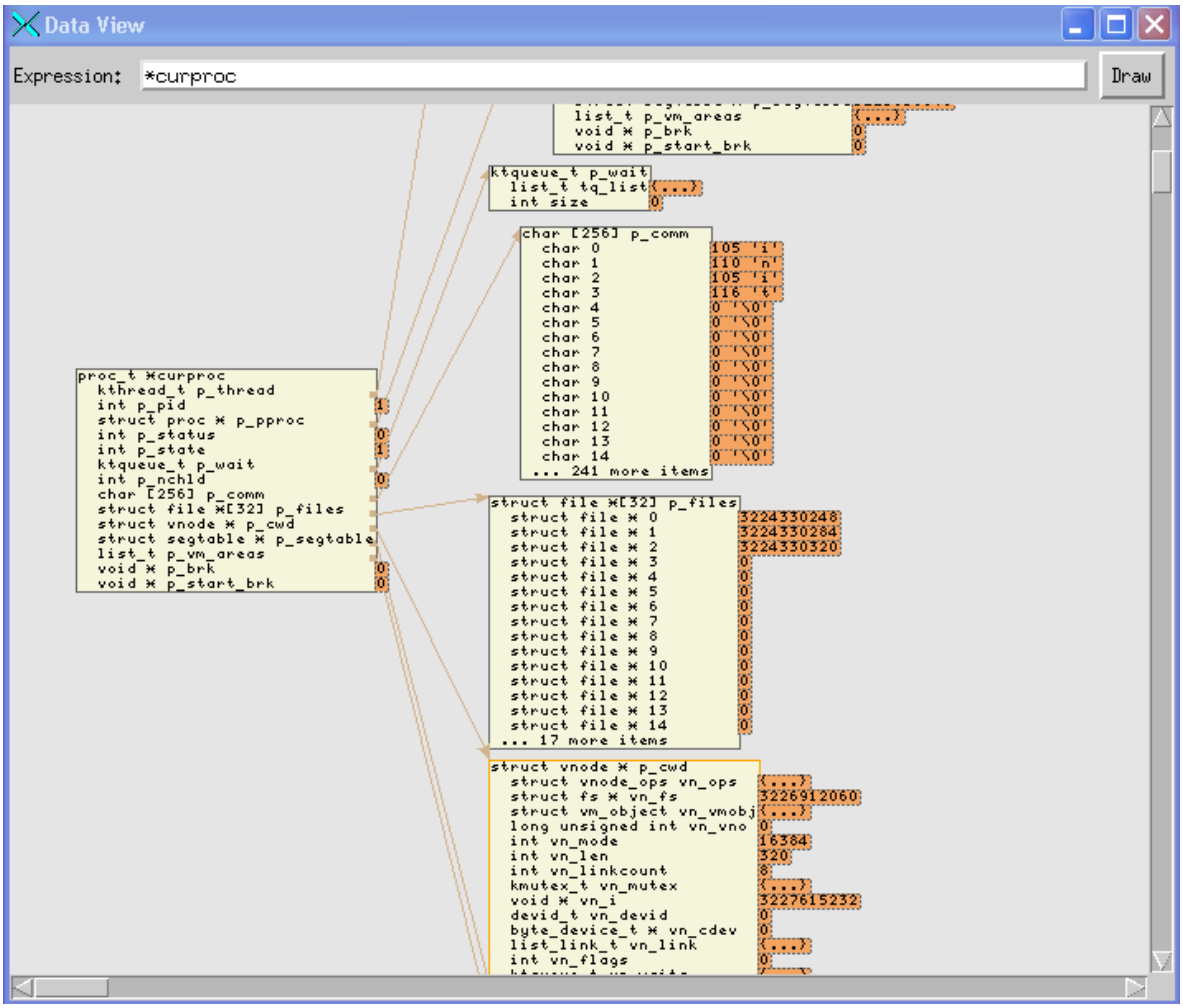

Figure 4: The dataview plugin

The dataview plugin allows the user to explore the data structures underlying Weenix visually. The user provides a depth and an optional list of labels to restrict the breadth of the graph, and the data structure is drawn to the screen. Each node can be rearranged so as to ensure that a useful view is obtained. Later versions of the plugin will also allow the user to navigate

particularly large structures by clicking on regions in a smaller map of the data.

The dataview is implemented using pure Tcl/Tk in just over 300 lines of code. No standard canvas widget provided all the flexibility that was required to build the graph rendering widget. A simple graph layout heuristic is used that could certainly use improvement but manages to yield decent results for tree-like graphs. A second version of the layout manager can be enabled by switching a flag in the code.

4.2.4. Disk

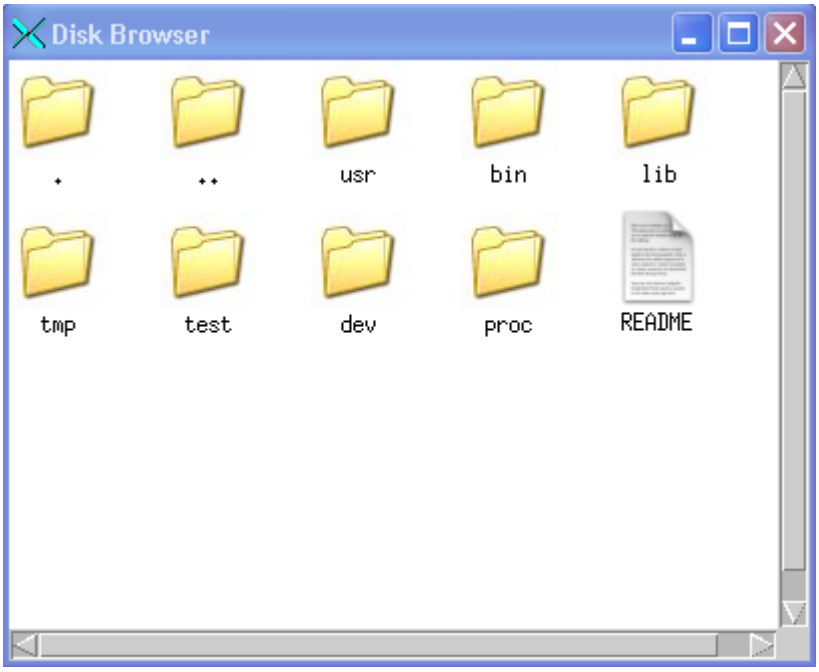

Figure 5: The disk plugin

The disk plugin allows the user to navigate the block device utilized by Weenix via an interface similar to Windows Explorer. Currently, the plugin is limited to browsing the file system, though extending the plugin to allow files to be viewed and modified would be a fairly trivial exercise (the commands for operating on the disk are already present within the plugin).

Like the dataview plugin, this plugin is implemented in pure Tcl/Tk; the file browsing widget is implemented entirely atop the standard canvas widget that comes with Tk. Directories are listed and types inferred by running the `fsmaker` utility in a child process.

4.2.5. Ptable

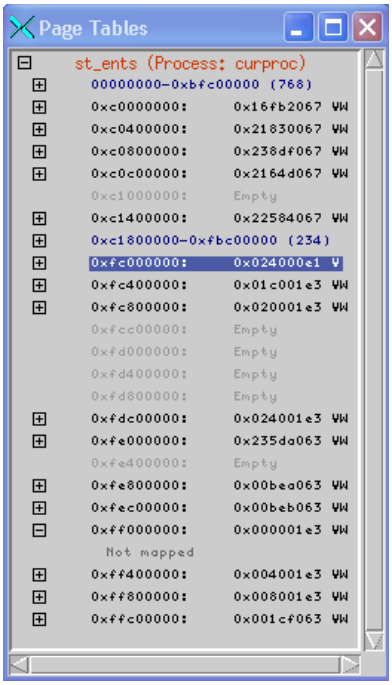

Figure 6: The ptable plugin

The ptable plugin allows the user to explore the L1 and L2 page tables associated with a particular process. Each top level child lists the segment address and flags associated with a particular segment table entry while each second level child lists the flags and mappings associated with a particular page table entry. Duplicate entries are compressed so as to make the view more useful.

At present, parsing the table entries for an active process can take up to two minutes; as a consequence, this plugin will be rewritten to load L1 table

entries only when a subtree is expanded. This plugin is implemented using the same tree widget as is described in section three.

5. API Reference

5.1. Overview

The following section aims to serve a quick reference for each of the modes, commands, and subcommands associated with the "`weenix`" command. A brief explanation is provided for each command as well as a list of required and optional arguments.

5.2. `weenix plugin`

Commands related to the plugin infrastructure.

5.2.1. `weenix plugin id`

Obtain the current plugin's unique identifier.

5.2.2. `weenix plugin callback` *`command`*

Wrap a plugin command for use as a callback.

5.2.3. `weenix plugin set` *`varname value`*

Set plugin state data; plugin reference is implicit.

5.2.4. `weenix plugin get` *`varname`*

Get plugin state data; plugin reference is implicit.

5.2.5. `weenix plugin active` [*`boolean`*]

Determine if plugin is receiving update notifications; plugin reference is implicit.

5.2.6. `weenix plugin state`

Show all plugin state data; plugin reference is implicit.

5.2.7. `weenix plugin path`

Return the path to the plugin's working directory.

5.2.8. `weenix plugin subscribe`

Set auto-continue breakpoints for updating state.

5.2.9. `weenix plugin unsubscribe`

Disable notifications and potentially clear auto-continue break point.

5.3. `weenix gui`

Commands related to updating and working with the [incr Insight] GUI.

5.3.1. `weenix gui varchanged` *`varname value`*

Notify GUI of a GDB variable change.

5.3.2. `weenix gui breakchanged` *`breakpoint state`*

Notify GUI of a GDB breakpoint change.

5.3.3. `weenix gui tracechanged` *`tracepoint state`*

Notify GUI of a GDB tracepoint change.

5.3.4. `weenix gui busy`

Visually indicate that the debugger is busy.

5.3.5. `weenix gui update`

Update all widgets and step event loop.

5.3.6. `weenix gui idle`

Unlock GUI for user interaction.

5.3.7. `weenix gui quit`

Safely quit the debugger.

5.3.8. `weenix gui forcequit`

Quit the debugger immediately.

5.3.9. `weenix gui signal` *`name [longname]`*

Alert the user that a signal has arrived.

5.3.10. `weenix gui reset`

Reinitialize GUI.

5.3.11. `weenix gui prompt`

Show graphical prompts or messages.

5.3.11.1. `weenix gui prompt ask` *`message [default:yes]`*

Ask the user a yes or no question.

5.3.11.2. `weenix gui prompt warn` *`message`*

Show the user a warning message.

5.3.11.3. `weenix gui prompt suggest` *`id message`*

Show the user a suggestion that can be subsequently ignored.

5.3.12. `weenix gui console`

Output to the graphical GDB console.

5.3.12.1. `weenix gui console echo` *`message`*

Output a message to the console.

5.3.12.2. `weenix gui console log` *`message`*

Output a log message to the console.

5.3.12.3. `weenix gui console target` *`message`*

Output a message to the console appearing to come from target.

5.3.12.4. `weenix gui console targeterr` *`message`*

Output an error message to the console appearing to come from target.

5.3.12.5. `weenix gui console error` *`message`*

Output an error to the console.

5.3.12.6. `weenix gui console read`

Prompt the user via the console.

5.4. `weenix gdb`

Commands for interacting with GDB.

5.4.1. `weenix gdb tty`

Obtain program's output TTY for reading and writing.

5.4.2. `weenix gdb open` *`[path:prompt]`*

Open a binary for debugging.

5.4.3. `weenix gdb close`

Close the current binary.

5.4.4. `weenix gdb step`

Execute a GDB step command.

5.4.5. `weenix gdb next`

Execute a GDB next command.

5.4.6. `weenix gdb finish`

Execute a GDB finish command.

5.4.7. `weenix gdb continue`

Execute a GDB continue command.

5.4.8. `weenix gdb nexti`

Execute a GDB next instruction command.

5.4.9. `weenix gdb stepi`

Execute a GDB step instruction command.

5.4.10. `weenix gdb script` *`path`*

Load a GDB script into the current session.

5.4.11. `weenix gdb funcs` *`sourcefile`*

List the functions in a loaded file.

5.4.12. `weenix gdb search` *`mode regexp [files:all] [static:false] [filename:false]`*

Search for a regular expression across all loaded files.

5.4.13. `weenix gdb stop`

Try to stop the debugger.

5.4.14. `weenix gdb printlimit` *`[set:query]`*

Set the GDB output limit; useful in conjunction with the raw expression evaluator.

5.4.15. `weenix gdb run`

Start debugging current executable (remotely if a remote target has been specified).

5.4.16. `weenix gdb target` *`target [prompt:true]`*

Set target based on prompt or provided string.

5.4.17. `weenix gdb command` *`command`*

Execute arbitrary GDB commands.

5.4.18. `weenix gdb setmemory` *`addr hexstring len`*

Modify memory in bulk.

5.4.19. `weenix gdb location`

Obtain a summary of GDB state.

5.4.20. `weenix gdb source` *`sourcefile`*

Obtain a list of executable lines for a loaded source file.

5.4.21. `weenix gdb sources`

Obtain a list of loaded source files.

5.4.22. `weenix gdb registers`

Obtain the current register state.

5.4.23. `weenix gdb breakpoint` *`breakpoint`*

Obtain info about a breakpoint.

5.4.24. `weenix gdb breakpoints`

Obtain a list of all breakpoints.

5.4.25. `weenix gdb expression`

Explore GDB expressions using a variety of mechanisms.

5.4.25.1. `weenix gdb expression raw` *`expr`*

Obtain the value of a single GDB expression without parsing to Tcl.

5.4.25.2. `weenix gdb expression lazy` *`expr`*

Obtain a lazy expression object that queries gdb only on updates; do not use with volatile expressions.

5.4.25.3. `weenix gdb expression tree` *`expr`*

Obtain the value of a single GDB expression as a Tcl parse tree.

5.4.25.4. `weenix gdb expression object` *`expr`*

Obtain a Tcl expression object for a given expression.

5.5. `weenix helper`

Common utility functions.

5.5.1. `weenix helper tree`

Parse tree utility functions.

5.5.1.1. `weenix helper tree expression` *`raw`*

Obtain the value of raw GDB output as a Tcl parse tree.

5.5.1.2. `weenix helper tree iterator` *`tree`*

Obtain a flattened parse tree that is easy to iterate over.

5.5.1.3. `weenix helper tree count` *`tree`*

Returns the number of nodes in a parse tree.

6. Acknowledgements

I would like to acknowledge Prof. Thomas W. Doeppner for guiding me throughout the perilous honors thesis process (and beyond!); Prof. John Jannotti for happily accepting my last minute plea for a reader; Prof. John F. Hughes for providing far too many extensions than one ought to (the same goes for Prof. Barbara Meier!); Keith Seitz for helping me to debug GDB; Daniel Kuebrich for putting up with my all-too-frequent disappearances; and my family, for giving me the opportunity to study at Brown in the first place.