USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

Dynamic Probes and Generalised Kernel Hooks Interface for Linux

Richard J Moore - richardj_moore@uk.ibm.com, *IBM Corporation, Linux Technology Centre*

Abstract:

Dynamic Probes (Dprobes)[1] is a generic and pervasive system debugging facility that will operate under the most extreme software conditions such as debugging a deep rooted operating system problems in a live environment. For example, page-manager bugs in the kernel or perhaps user or system problems that will not re-create easily in either a lab or production environment. For such inaccessible problem scenarios Dprobes not only offers a technique for gathering diagnostic information but has a high probability of successful outcome without the need to build custom modules for debugging purposes.

Dprobes allows the insertion of fully automated breakpoints or probepoints, anywhere in the system and user space. Probepoints are global by definition, that is they are defined relative to a module not to a storage address. Each probepoint has an associated set of probe instructions that are interpreted when the probe fires. These instructions allow memory and CPU registers to be examined and altered using conditional logic. When the probe program terminates an external debugging facility may be optionally triggered - should it register for this purpose. For example:

A trace facility may augment its capability with a dynamic trace capability by using the Dprobes facility as a means of inserting tracepoints - dynamically, without any prior code modification.

A crash dump facility may use Dprobes as a means of invoking dumps conditionally when a specific set of circumstances occurs in a particular code path.

A debugger may use Dprobes as high-speed complex conditional breakpoint service.

This paper describes the architecture of Dynamic Probes and briefly discusses a couple of examples of its successful application.

In creating Dynamic Probes, we were challenged with the conflicts between:

Size of the kernel modification

Co-existence with other kernel enhancements, particularly debugging and instrumentation enhancements.

Maintaining concurrency with the latest kernel version.

Ease of development and continued enhancement of Dynamic Probes.

We alleviated these problems by developing a generalised interface for kernel modifications to use allowing them to exist as dynamically loadable kernel modules.

This interface: The **Generalised Kernel Hooks Interface (GKHI)** is described in the second part of this paper.

Historical Perspective

The idea for Dprobes was taken from a technology we previously developed on OS/2[2] originally for implementing tracepoints dynamically without requiring source code modification. This facility operated by making dynamic changes to the code of a loaded module to cause an interrupt at a tracepoint in the same way a debugger inserts a breakpoint. When the tracepoint handler received control it interpreted a small program associated with the tracepoint to collect data from processor registers and memory and built a trace record which was then passed to the system trace facility. Using this methodology there was no overhead when the tracepoints were not active and no requirement to modify a program to allow tracepoints. The trace program contained conditional logic and a limited amount of arithmetical and logical manipulation of data. We extended the programming language to allow Dynamic Trace to invoke user code in the form of device drivers and to exit to other debugging facilities for example:

Kernel Debugger
Application Debugger
System Dump
Application Dump.

Or indeed do nothing! The fact that we had conditional logic gave us a very powerful tool for monitoring a piece of code until conditions presented themselves that required user action. We were also able to accumulate information in the form of local variables, which could be used to retain data from one invocation of a tracepoint to another and be later accessed and display by a command utility.

The original OS/2[2] facility suffered from three design limitations:

1. It was deeply imbedded in large parts of the kernel processing and could not be modularised easily.
2. It was designed primarily as a tracing mechanism.
3. Some simplifications were made upon where and when tracepoints could be placed, for example interrupt-time tracepoints were originally disallowed. Also, because of the mechanism for implementing tracepoints (described later), they could not be placed on certain Intel[3] instructions.

In bringing this technology to Linux we have sought to address these problems in particular. We have attempted to divorce dynamic trace from trace and have generalised it's capability. We have called the Linux realisation of this technology: **Dynamic Probes** or **Dprobes** for short. We have correspondingly changed the term tracepoint to the more generalised **probepoint**.

Over and above the original OS/2[2] idea we have implemented Dprobes with the following characteristics:

It has a greatly extended and generalised probe program command set.

The implementation under Linux has been to make Dprobes an independent facility with a formal interface to allow other debugging facilities to gain control when a probepoint executes.

Dprobes is modular, existing as a set of command utilities and kernel modules. The modification to the kernel has been abstracted to a minimal set of changes by means of a technology we have introduced called **Generalised Kernel Hooks Interface (GKHI).**

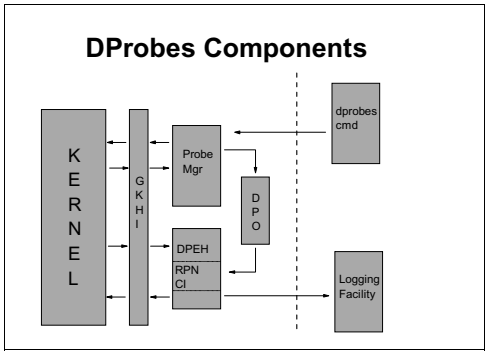
We have also sought to separate platform dependent code from independent code so that the effort in porting to other platforms is minimised.

The probe insertion technique has been improved by delaying the physical insertion of probes in a page of an executable until the time that page is brought into memory on demand.

Probes can be applied to code that is under control of a debugger, without interference to the either the debugger or Dprobes.

Component Overview

Dprobes comprises the following major components whose interrelationships are shown in the figure below:



The Kernel part of Dprobes comprises the Dprobes Manager and the Dprobes Event Handler (DPEH). The Dprobes Manager is responsible for:

Accepting requests to register and deregister probes from the dprobes command line utility. It provides an API for this purpose.

Saving the probe definitions for each probed module in a per-module dynamic probe object[4]. This object comprises the following parts:

The set of probe programs for this module.

The local variable array.

Probe records for each probe defined for that module. Each probe record contains the location of the probe, maintained as file **inode-offset** pair. This provides a universal way of identifying the a probe regardless of whether the module is in memory or not, and if it is, where in memory different instances of the module are located. The probe record also contains a pointer to the probe program associated with it.

As discussed below under **The Breakpoint Mechanism**, probe insertion causes code to be modified in memory. Probes are inserted whenever a page within a probed module is loaded into memory. This is achieved by creating an alias virtual address to the physical address of the probe location . This allows us to insert probe in modules in other process contexts. We also cater for pages marked Copy-on-Write and pages of a shared module that might be loaded at different virtual addresses in different processes. When probes are inserted for the first time we register the **readpage** filter routine for the module. This allows us to be able to re-insert probes when discarded pages are reloaded in memory. This probe insertion technique avoids changing the page state and avoids breaking off multiple copies of swappable pages which would happen if we were merely to store into the virtual address.

Re-inserting probes when a page of code is brought back into memory after having been discarded.

Removal of a probe. As discussed below under **The Breakpoint Mechanism**, we use an instruction-replacement form of breakpoint, which requires us to restore the original instruction in a similar manner to insertion by means of an alias address.

The DPEH is responsible for handling a probe event notification. Event notification occurs when a probed location of a module executes. The DPEH does this by intercepting the system breakpoint and single-step exception handlers. This is described in more detail below. The DPEH identifies a probe event with its dynamic probe object by determining the **inode-offset** that corresponds to the event. It then invokes the **Probe Program Command Interpreter**.

The **Probe Program Command Interpreter** executes commands in the probe program lodged in the dynamic probe object. Should an exception occur then interpretation is quietly terminated with an error indication in the temporary log buffer, which is the temporary piece of storage allocated per processor. This buffer is made available to any external facilities that might register to be notified when the Interpreter exits via the **exit** RPN command. The usable size of the temporary log buffer is determined per probe object, with a fixed maximum size of 1K, which will become configurable.

The user communicates with the kernel components by means of the **dprobes** command line utility, which supports the following major functions:

Insert:

This reads a **dynamic probe definition file (DPDF),** which contains among other things, the module name to which the probe definitions apply, the local variable array size, and for each probe within the module its probe definition. The probe definition comprises the probe location, which may be expressed as a symbolic expression, a user identifier for the probe - a major-minor code pair and the text of the associated RPN probe program. The insert function parses the DPDF and passes a condensed form of this to the probe manager to be saved as a dynamic probe object.
Existing probe definitions for the same target module may be optionally replaced or merged with the new definitions.

Remove:

This will remove all probes from a module. Optionally a subset of the probes may be removed. Probe removal is the reverse of insertion.

Getvars:

This will extract local and global variables for one or more probe definitions. Optionally it may be used to reset the values to zero.

Query

This will display the state of registered probes.

BuildPPDF

This builds a **pre-built probe definition file (PPDF)**, which is essentially a file version of the package built by the **Insert** function. The value in providing this function is that probe definitions can be pre-built from a **makefile** and later inserted using the **dprobes** command's **insert** function. This would allow a module to be installed along with its PPDF so that is debug-ready, without being a special built with debugging code present. Pre-building is made possible because the probe location may be expressed symbolically using symbols from the module's symbol table.

The **dprobes** command supports preprocessor directives supported by the **GNU gcc** compiler. If present, dprobes will invoke **gcc** to resolve preprocessor directives and direct the output to a temporary file against which the probe definitions are parsed. This facilitates:

Substitution into the DPDF from the command line
Macro definitions
Conditional preprocessing

The Breakpoint Mechanism

At the heart of dynamic probes lies the **probepoint** which is a breakpoint - the same as that implemented by a debugger - with a few implementation differences:

Because Dprobes is in one sense an automated kernel debugger we do not wish a breakpoint to interrupt execution temporarily. Instead it gives control to the **Dprobes Event Handler (DPEH)**, which under normal circumstances will return control to the program without user intervention. Because the breakpoint is automated and does not really break execution, we refer to it as a **probepoint**.

As is usual for breakpoints we intercept code before the probed instruction executes. This might seem like an otiose statement given that the beginning of one instruction is the end of another - but not so when the mechanism for implementing the breakpoint is examined. There are in general two mechanisms by which to implement a breakpoint:

Instruction replacement
Watchpoint.

Instruction replacement as the name implies requires that the probed instruction is replaced by another that gives control to the DPEH. Before the DPEH returns control to the probed program we execute the original instruction. Use of instruction replacement provides a pre-execution breakpoint on all hardware platforms.

The watchpoint is a hardware assisted mechanism for interrupting execution in order to give control to some monitoring application, such as the DPEH. Unfortunately watchpoint implementation is not consistent across all processors. On Intel[3] 32-bit architecture there are four debugging registers provided for watchpoint implementation, a severe limitation in itself. Intel[3] watchpoints interrupt on instruction fetch. In contrast S/390[5] watchpoints operate over a continuous range and interrupt on completion or partial completion of the execution of an instruction.

Since there's no easy abstraction of watchpoints across processor platforms we use the *Instruction Replacement* technique.

The DPEH is discussed more fully later however it does play an essential role in the breakpoint mechanism. Its main components are:

Execute *Probe Program* commands
Single-step original instruction
Commit probe buffer
Return to probed program.

On entry to the DPEH, application, system and processor state can be examined as one would do from a kernel debugger but by means of commands in the associated RPN Probe Program. This requires that the DPEH operates at a supervisor level of privilege. Therefore to enter the DPEH from any privilege level we require the breakpoint to be implemented using an instruction that will cause an interrupt. Under Intel[3] the **INT3** instruction suffices and is designed for this purpose. Under S/390[5] the **SVC255** instruction provides an equivalent function. Each processor platform that distinguishes privilege levels will offer an

instruction equivalent to these that will allow controlled access into a supervisor privilege level of execution.

The second stage of the DPEH is to single-step the original instruction followed by a commit phase. (Committal applies to the temporary log buffer, when passing this to an external facility, for example a tracing facility or the default **Syslog** facility).

Two questions arise:

1. Why single-step?
2. Why commit after single-stepping?

Because the DPEH is privileged we cannot easily execute a copy of the original instruction in-line in the context of the DPEH since that would:

a. Grant to the original instruction and hence the probed program a privilege level that may not authorised
b. Alter the execution outcome of some instructions, e.g. POPF on Intel[3].

The DPEH allows the Probe Program to save data in a temporary buffer before committal on completion of the single-step. In this way the DPEH provides a tracing or logging mechanism. Some instructions might be interrupted several times and restarted before they complete (I am including here the possibility of recoverable exceptions, for example page-faults). We would not wish to record multiple events for each re-execution. This requires the DPEH to commit log data after the original instruction has completed execution.

Single-step allows the DPEH to execute the original instruction in its intended context and gain control on completion.

In the current implementation under Intel[3] 32-bit architecture (IA32), the single-step phase of the DPEH comprises restoring the original instruction, returning from the DPEH using the **IRET** instruction with the **TRACE** flag set in the **EFLAGS** register. If the single-stepped instruction completes without interruption or exception control is returned to the DPEH via the single-step system exception handler and logged information is committed. The **INT3** is restored and the DPEH returns to the probed program.

If the single-stepped original instruction terminates with an exception (other than single-step), the log buffer is discarded and the **INT3** is restored. This requires that the DPEH be given control from all system exception handlers. (In the current implementation under IA32 we have disallowed probes on **INTn** instructions since these always end in an exception and would require an unnecessary intrusion into the system exception handlers for exceptions 0x20 - 0xff).

Implementation details of the probepoint, particularly the single-step are clearly processor dependent. However, mechanisms for single-stepping instructions under program control exist on all modern processor architectures. The single-step mechanism is therefore customised per architecture. To ease porting to other platforms, the single-step is made modular and therefore easily replaceable. The single-step implementation under Intel[3] more or less forces us to do this with interrupts disabled, since there is no easy way to save state across the single-step should we re-enter the breakpoint exception handler, consequently we single-step with interrupts disabled and make appropriate adjustments for instructions such as **PUSHF**, **CLI** and **STI**. In addition we take steps to avoid recursion due to probepoints being placed in the path of the DPEH. We do this in two ways:

Preventatively: by disallowing registration of probepoints within the DPEH module.

Reactively: by using a recursion counter to detect unexpected recursion. This caters for probepoints in subroutines called by the DPEH. If we detect recursion we *silently* remove the recursing probepoint and return to the probed code.

The *Instruction Replacement* form of breakpoint has two undesirable side-affects, which may or may not be troublesome, depending on architecture:

In order to single-step the original instruction in context, we temporarily replace the breakpoint instruction with the original instruction. This opens a window of opportunity for a probepoint to be missed if the same piece of probed code is executed on another processor in close succession. We can avoid this by forcing processor serialisation during the single-step. However, that can badly affect performance, and so, is left as an option for the user to deploy if needed. In practice this has not been a problem because Dprobes has been used to find bugs in code that is already serialised or races against other code that jointly accesses common data.
The other side-affect is very much architecture dependent. When dynamically changing instructions, some architectures will require additional actions to be

carried out in order to guarantee consistent results. For example, not all architectures fetch whole instructions as an atomic entity nor do they do this in address order. Furthermore the problem is compounded when code is stored in read-only memory and an update has to be done using an aliased read/write virtual address. Usually there are ways around these problems; one needs to read very carefully the processor documentation regarding instruction caching, pipe-lining and speculative execution.

The instruction-replacement form of breakpoint and single-step requires that the breakpoint be re-instated on completion of the single-step.

The Dynamic Probe Event Handler

The Dynamic Probe Event Handler (DPEH) is invoked as a result of a probepoint executing. As mentioned in the description of the breakpoint mechanism, the DPEH comprises 4 phases of operation:

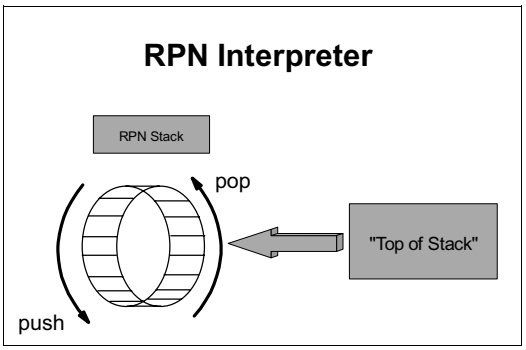
Execute *Probe Program* commands
Single-step original instruction
Commit probe buffer
Return to probed program.

The last three phases were discussed under the previous section - The Breakpoint Mechanism. We now look in detail at the first phase - **The Probe Command Interpreter**.

When a probe is registered with the Dprobes Manager an associated **probe program** is lodged with he Dprobes Manager. This program defines the actions that will be carried out when the probe is executed. The probe program language is of the Reverse Polish Notation (RPN) family of languages. Therefore, an implicit stack is associated with the probe program for temporarily storing operands and results of RPN commands.

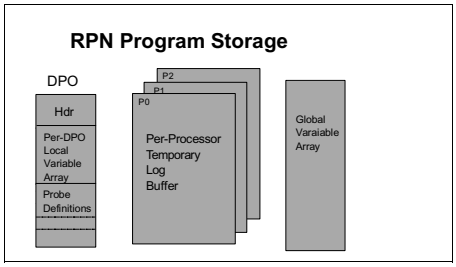
The RPN family of languages are implemented using a stack on which command operands are pushed. When a command is executed the operands are popped from the RPN stack and the result pushed on to the stack. Many implementation use a circular array of fixed width for the RPN stack and maintain a special pointer that locates the Top of Stack (TOS), which is precisely the implementation within Dprobes. Two special (families of) commands are provided to for the program to access the stack:

PUSH
This rotates the stack forward and copies data to the TOS.
POP
This rotates the stack backward, but does not update tack contents.



Dprobes implements a 32-element circular RPN stack of width equal to the bus width of the processor. In the case of Intel[3] 32-bit architecture each RPN stack element is 32-bits.

There are three other storage areas provided for use by the RPN program:



The Temporary Log Buffer is used to accumulate data during the execution of the RPN program. Depending on how the program terminates, will be passed on to an external debugging facility, for example, **Syslog** or **Trace**. The temporary log buffer contents persists only during the execution of the probe program. Contents are discarded if the program ends abnormally or is deliberately aborted. One temporary log buffer is defined per processor, the usable size of which is defined per module, when probes are registered.

The Local Variable Storage Array is provided per probed module and shared among all probe programs that are operative for a probed module. Each array element, or local variable, is retained across invocations of the RPN program and may be used to

share data between probe programs for a given module. Local Variables can be extracted, displayed and reset by the dprobes command utility. The size of the local variable array is defined when probes for a module are registered.

Global Variable Storage is similar to Local variable Storage, but is common to all probed modules.

The **DPEH RPN Command Interpreter** implements the following classes of commands:

Execution and Sequencing Group.
This group includes conditional jumps, loop, subroutine call, exit control and probe control.

Logging Group.
This group includes commands that update the temporary log buffer.

Local and Global Variable Group.
This includes commands that perform allocation of global variables and command that read and write to global and local variables.

Arithmetic/Logic Group.
This includes commands that operate with the RPN stack. They include addition, multiplication, subtraction, bit masking, bit shifting and rotation.

Address Verification.
This in a small group of two commands that will test the validity of a memory addresses. These are provided because the interpreter runs with interrupts disabled and cannot page-in swapped or discarded memory - see **Data Group** below.

Stack Manipulation.
A single POP command used to rotate the RPN stack without data manipulation. Commands that manipulate data on the RPN stack are included in the groups relating to the data origins and destinations.

Register Group.
This include commands to push processor registers onto the RPN stack and commands to pop registers from the RPN stack. Both the current processor registers and the current user context registers may be accessed.

Data Group.
This includes commands to push and pop data in memory to and from the RPN stack. A subset of this group is the system variable subgroup which will access key system data values, for example current process ID and address of current **task_struct**. Because the DPEH runs with interrupts disabled, exceptions caused by data group commands are soft-failed by the interpreter by halting interpretation and storing a failure code in the temporary log buffer.

IO Group.
This include commands to push and pop data from the IO address space (not implemented on platforms that do not support an IO address space).

Example Probe Programs

Example 1 - fork and kill:

```
name = bzImage
modtype = kernel
major = 1
jmpmax = 32
logmax = 100\\
dfltexit=1
vars = 2

offset = kill_proc
opcode = 0x55
minor = 1
pass_count = 0
max_hits = 1000
inc lv,0
push d,16
push r, esp
log mrf
exit

offset = do_fork
opcode = 0x55
minor = 2
pass_count = 0
max_hits = 1000
inc lv,1
push d,16
push r, esp
log mrf
exit
```

The example above shows an RPN probe that will create a Syslog entry ever time a process forks and is killed. It will accumulate the number of forks and kills in local variables 1 and 0. These variable may be displayed at any time using:

dprobes -g -a

from the command line. Whenever **kill_proc** or **do_fork** is entered, the probe programs above will write to the temporary log 16 bytes of current stack data, which on exit will be written to **Syslog** using **printk**.

Example 2 - malloc:

```
name = "/lib/libc.so.6"
modtype = user
major = 1
jmpmax = 32
logmax = 100
dfltexit=0

offset = __malloc
opcode = 0x55
minor = 1
pass_count = 0
push r,esp                  // push the stack
pointer
push d,4
add                         // TOS -> first parm
to malloc (size)
push d,0x20000000           // size 0x20000000
sub                         // compare
jz take_dump                // if equal, take core
dump
exit                        // else exit without
further ado
take_dump: exit 3
```

In this example we place a probe on entry to the **malloc** routine of **libc**. We look for the instance where malloc is called with a size value of **0x20000000** and when found we force a core dump.

Real-life Examples of Dprobes Use.

Dprobes is a new technology for Linux so the number of example problem determination uses on Linux is limited, however it is worth mentioning that Dprobes was used to debug itself during development.

The following examples have been taken from the OS/2[2] platform from which Dprobes was developed. I'll summarise two contrasting examples: the first an operating system bug, the second an application space bug.

Example: A deep-rooted operating system bug.
We had a situation where we found the file-system was page-faulting unexpectedly. What was very odd about this situation was that the page-fault was occurring on a page that should have been locked in memory. We could never reproduce this problem in the lab, it only happened on a customer's server, once a day at peak load.

The first hypothesis was that the file system had a bug and had accidentally unlocked a locked page. So we created a PPDF that contained probe definitions of the file system's locking and unlocking subroutines. We sent this to the customer and asked him to insert the probes. When the system next crashed, the customer sent the crash dump and log created by dprobes.

The dump and dprobes logs showed only a lock request for the faulting page, however the page status could be seen from the dump to be unlocked.

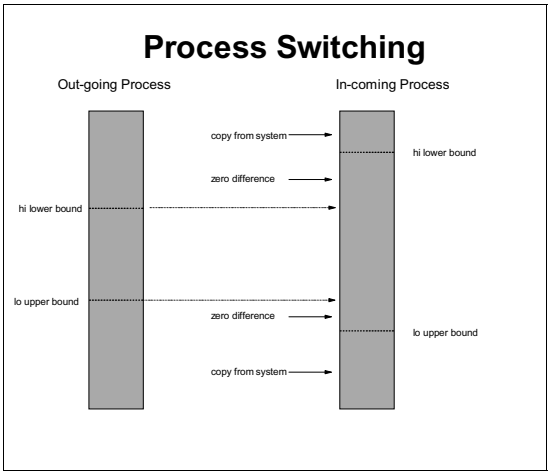
The second hypothesis was to assume that another kernel module was unlocking the page. So we sent the customer a PPDF that logged all calls to the kernel's lock and unlock routines. Once again the log showed that no-one had unlocked the faulting page. However we did notice that every time the locked page faulted a great deal of process switching had taken place. We began to suspect that there might be a page-manager problem in the kernel. To see whether something odd was happening we placed probes inside the process context switching routine.

Code inside the context switcher is very difficult to debug, We are in no defined process state. Page Tables and system state variables in an inconsistent state with respect to each other. But because the DPEH was designed to operate in the most hostile conditions we could use Dprobes to find out what was happening. Inside the Context Switcher we placed two probes: one before page data was saved. This probe logged the out-going processes page tables. The other probe was placed near the exit of the context switcher and logged the in-coming process' page tables.

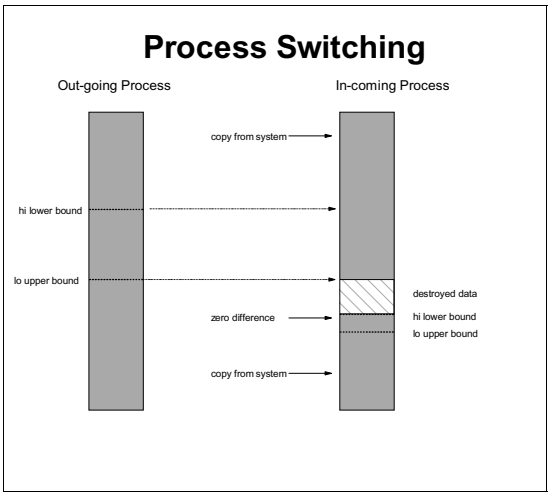
On re-creating the problem we could see that the out-going process' page tables showed the faulting page present and when the process was next run the faulting page table entry was zero. But the copy of the page data, maintained by the system while the process was not being run , still had the page table entry intact (we could see this from the dump). There had to be a bug in the context switcher - despite having been written some 15 years ago and not had a defect in all that time.

It was time to examine the code: someone had introduced a performance enhancement to avoid double-updating of page tables. This enhancement had a

bug which caused valid page tables entries to be zeroed when the outgoing process' upper bound for low memory over-lapped the in-coming process' lower bound for high. See the picture below for clarification.



This situation worked until the following occurred:



Example: who sent the parcel-bomb?
This is an application-space example. The messaging facility within OS/2[2] has an asynchronous function: **WinPostMsg**. When a message is posted the application's message thread is posted - made ready to run. When it runs it will find one or messages on its message queue along with optional parameters. A posted message is asynchronous. By the time the receiver wakes, the poster could have even terminated. And there's no way of knowing who it was.

We had a situation where a message was being posted to an application but every now and then a message was posted with an erroneous parameter that caused the receiver to trap. The question is who sent it?

We found this out simply by putting a probe on the entry to the WinPostMsg. The probe contained conditional logic to examine the message ID and the parameter. When the ID and Parameter matched the values that caused the receiver to trap we invoked a crash dump. The dump was taken in the context of the poster and allowed us to see who it was and where in their code it was happening.

Conclusions:
Both these examples could have been solved with other debugging tools, but not easily so. Both would have needed an ability to place global breakpoints at certain code locations and at the same time exercise conditional logic. Potentially a kernel debugger can do this. However, the breakpoints deployed require performance to be maintained, furthermore kernel debuggers tend to assume a breakpoint really means break - so are designed to perform serialisation function to allow direct user communication. The DPEH, on the other hand, is designed with minimal serialisation dependencies and no user interaction. It can therefore maintain system performance with complex conditional breakpoints applied.

Other uses of Dprobes has been to provide a high-speed conditional breakpoint facility which gives control to a kernel debugger when the correct situation presents itself.

Generalised Kernel Hooks Interface

Like other modifications to the base operating system, there are problems with having to manage large modifications, for example:

> If more than one independent enhancement needs to co-exist with another, then patches for each may conflict and have to be resolved - possibly by the user.
>
> If the enhancement needs to be updated then a new patch must be supplied and integrated with existing patches. The kernel will require recompilation and possibly also some kernel modules.
>
> If the kernel needs to be modified for maintenance reasons then the patches need to be re-worked and re-applied.
>
> If the user wants to use patches, albeit infrequently, they must either run with that additional function even when not needed , or switch kernels when the function is needed. Either way there's an inconvenience and an overhead implied.

Dprobes, like other diagnostic and instrumentation facilities tends to insert additional function rather than replace it. For this category of kernel enhancement we have the opportunity to separate the kernel enhancement from the kernel by confining it to a loadable kernel module - provided - interfaces are added to the kernel to allow kernel modules to be called at the appropriate time.

A particular problem that needs to be overcome is how to define such an interface. We cannot simply code within the kernel, calls to external modules because we would not be able to resolve those calls at kernel load time. To overcome this , we define **hooks** within the kernel.

A **hook** is a small piece of code inserted into the kernel source, actually a one-line change because we define the hook using a C macro, an example of which is shown below:

```
#define GKHOOK_2VAR_RO(h, p0, p1) asm volatile
(".global GKHook"h";
                    .global GKHook"h"_ret;
                    .global GKHook"h"_bp;
          /* replace with nop;nop;nop; to activate */
                    GKHook"h": jmp GKHook"h"_bp;
                              leal %1,%%eax
                              push %%eax;
                              leal %0,%%eax;
                              push %%eax;
                              push $2;
 /* replace with jmp GKHook"h"_dsp when active*/
                              Nop;nop;nop;nop;nop
                    GKHook"h"_ret: add $12,%%esp;
                    GKHook"h"_bp:;"
                    ::"m"(p0),"m"(p1):"%eax")
```

The hook begins with a jump to the end of the hook, in its dormant state. The body of the hook contains space for instructions to be added when the hook becomes active. When a hook activates, the body of the hook is populated with instructions to call the **Generalised Kernel Hook Interface (GKHI)** module at the hook's dispatcher entry point and the initial jump instruction is nullified by replacing it with NOP instructions. The GKHI is responsible for activating hooks and makes these modifications when another kernel module calls the GKHI to request that a hook be enabled for its use.

This implementation requires that a minimal change be made to the kernel: a hook be identified at each location in the kernel source where external modules may wish to gain control. But, it allows multiple modules to access the same hook without further modification to the kernel as we shall see when the GKHI is described in more detail.

The GKHI provides four interfaces for kernel modules to call in order to access or relinquish access to one or more kernel hooks.

GHK_register:
This is used to identify a location (a **hook exit**) in a kernel module that wishes to gain control at a given hook. The caller passes a hook registration record to the GKHI which contains the hook exit address and a flag indicating the desired dispatching priority which may be:

> First in list of exists for this hook
> Last in list of exits for this hook
> Only exit for this hook
> Unspecified.

A hook exit will not be dispatched until it is armed.

GHK_arm:
This allows one or more hook exits to be made dispatchable. The kernel module calling this interface will pass a list of chained hook registration records that are to be armed. If for any of the hooks referenced this is the first instance of the hook being used the GKHI will activate the hook. So, it is not until a module needs to use a hook will the additional code path be imposed on the kernel at the hook location. Arming is carried out in an atomic fashion, under SMP this requires other processors to be temporarily suspended from performing useful work. The impact, however, is minimal, since arming amounts to activating the hook if not already active and updating status maintained with in the hook registration record for each hook exit being armed. This is a CPU-bound operation on resident memory.

GKH_disarm:
This has the opposite affect of GKH_arm: a list of hook registration records is passed to the GKHI, which will mark each one as no longer dispatchable. If any one of these is the last to disarm for a given hook then the GKHI will make the hook inactive by restoring the original bypass jump instruction. Disarming is carried out atomically.

GHK_deregister:
This has the opposite affect of GKH_register where each hook exit in the list of exits requiring de-registration will be removed from knowledge of the GKHI.

When a hook is active and exits are armed, the dispatcher routine will be called when the hook executes. This routine is responsible for calling each of the armed hook exits for that hook, in priority order. If any exit returns a non-zero result no further exits are dispatched on that invocation of the dispatcher. As part of the registration of a hook exit, entry conditions may be specified:

All CPUs are temporarily stopped
Interrupts are disabled

In addition, the exit may adjust the status flag in its registration record so to disarm itself. This together with the ability to specify entry conditions gives the exit the an opportunity to be dispatched and disarmed as an atomic entity.

Part of the definition of a hook is whether it will pass any parameters to the hook exit and in addition whether those parameters will be modifiable. In the example given above the hook macro defines two read-only parameters. But in addition to passing these parameters we also pass a count of the number of parameters. In this way a hook may be modified to have additional parameters added. And the exit will know whether it is compatible with the current hook definition. (By convention we add new parameters to the end of this list).

GKHI may be loaded at any time using **insmod**, however no hooks can be activated until the GKHI is loaded. It is possible for kernel modules to define hooks within themselves, but using the regime described so far would require that the kernel module containing the hook be loaded before the GKHI. To achieve even greater flexibility, the GKHI defines two further interfaces:

GKH_identify:
This interface is called to notify the GKHI of a new hook that has become available since it initialised. Obviously exits cannot register for the new hook until it has identified itself.

GKH_delete:
This interface is called to notify the GKHI the a hook is not longer eligible for registration.

Where to obtain Dprobes:

Dprobes, including the GKHI, is available from IBM's Linux Technology Centre's web page at:

http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes

The development team comprises:
Richard J Moore (Dprobes Project Lead) - richardj_moore@uk.ibm.com
Bharata B Rao - rbharata@in.ibm.com
Subodh Soni - ssubodh@in.ibm.com
Maneesh Soni - smaneesh@in.ibm.com
Vamsi Krishna Sangavarapu - r1vamsi@in.ibm.com
Suparna Bhattacharya - bsuparna@in.ibm.com

Notes and references:

[1]Dprobes website: http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes

[2]OS/2 is a trademark of International Business Machines Corporation.

[3]Intel is a trademark of the Intel Corporation

[4]The dynamic probe object is a complex set of structures, the principle being dp_module_struct. Each of these is defined in **dprobes.h**

[5]S/390 is a trademark of International Business Machines Corporation.