DynAMOS manual

RELEASE_0-5-5

The DynAMOS Team

DynAMOS manual: RELEASE_0-5-5

by The DynAMOS Team

Published 2007-07-23

This is the documentation of DynAMOS, a dynamic kernel updating system for commodity operating system kernels.

Table of Contents

1. Introduction **1**
1.1. What Is It? 1
1.2. Example Updates 1
2. Approach **3**
2.1. Related Work 3
2.2. Architecture 3
2.3. Adaptive Function Cloning 3
2.4. Safe Execution Flow Redirection 4
2.5. Version Manager 5
2.6. Safe Removal 6
2.7. Function Image Relocation 6
2.8. Symbol Resolution 10
2.9. Datatype Updates 10
2.10. Constructing Updates 10
3. Installation **12**
3.1. Availability 12
3.2. Installation 12
4. Framework **13**
4.1. Starting 13
4.2. Stopping 13
4.3. Control Tool 13
4.4. Control Commands 13
4.5. Writing Adaptation Handlers 16
4.6. Updating Datatypes 17
4.7. Control Center 20
4.8. Remote Control Service 20
5. Examples **21**
5.1. Updating `get_pid` 22
5.2. Updating The Process Scheduler 22
A. FAQ **23**
Glossary **24**

List of Figures

2-1. A routine that immediately blocks through a call to the scheduler. 4
2-2. Disassembled output of a routine that immediately blocks through a call to the scheduler.....5
2-3. Disassembled output of a highly optimized version of the routine using the gcc arguments `-Os` (optimize for size) a
5
2-4. Definition of `BUG` macro in Linux 2.4 7
2-5. Definition of `switch_to` macro in Linux 2.4 7
2-6. Disassembled output of `switch_to` macro in Linux 2.4 8
2-7. `down` semaphore acquire operation in Linux 2.4 8
2-8. Disassembled output of `down` semaphore acquire operation in Linux 2.4....8
2-9. Multiple entrypoints produced by `icc` 9
4-1. Starting the framework. 13
4-2. Stopping the framework. 13
4-3. Example control commands. 16
4-4. Function signature of `pipe_write`. 17
4-5. Example adaptation handler for `pipe_write`. 17
4-6. Managing a datatype mapping table. 17
4-7. Definition of `struct new_task_struct`. 18
4-8. Creating a shadow variable in `do_fork`. 18
4-9. Freeing a shadow variable in `do_exit`. 19
5-1. Running the example updates. 21
5-2. Expected output of `get_pid()` update. 22
5-3. Expected output of `schedule()` update. 22

Chapter 1. Introduction

1.1. What Is It?

DynAMOS is an on-the-fly kernel updating system that enables commodity operating systems to gain adaptive and mutative capabilities without kernel source recompilation or reboot. It employs a novel and efficient dynamic instrumentation technique called *adaptive function cloning*. Execution flow can be switched adaptively among multiple editions of functions, possibly concurrently running. This approach becomes the foundation for dynamic replacement of non-quiescent kernel subsystems when the timeliness of an update depends on synchronization of multiple kernel paths.

DynAMOS supports:

- **Updates of non-quiescent subsystems**. It accomplishes substantial updates of core kernel subsystems that never quiesce, such as the scheduler and kernel threads.
- **Datatype updates**. It offers a technique for updating compact datatype definitions. Additions of new fields in a datatype is supported using a shadow data structure containing the fields.
- **Safe reversibility**. A methodology of quiescence detection is employed. Updated functions can be removed with the guarantee that they are not used by the stack or program counter of any process.
- **Adaptability**. Execution can be switched adaptively between multiple, possibly concurrently running, function editions. This is the first dynamically applied, adaptive kernel updating system.
- **Synchronized updates**. A multi-phase updating algorithm for replacement of complete kernel subsystems is offered. Notably, for the cases where the timeliness of an update depends on synchronization of multiple kernel paths.

1.2. Example Updates

Dynamic updates DynAMOS is able to successfully carry out include:

- Extending the Linux 2.2 kernel process scheduler to support unobtrusive, fine-grain cycle stealing offered by the Linger-Longer (http://www.cs.umd.edu/~kdryu/papers/LL_SC98/LL_SC98.pdf) system.
- Introducing adaptive memory paging for efficient gang-scheduling (http://www.public.asu.edu/~kryu1/papers/adapt_page_ipdps_camera.pdf) in a Linux 2.4 cluster.
- Adaptively updating the Linux pipefs implementation during large data transfers.
- Introducing kernel-assisted process checkpointing offered by EPCKPT (http://www.research.rutgers.edu/~edpin/epckpt/) in Linux 2.4.
- Applying security fixes provided by the OpenWall project (http://www.openwall.com/).
- Injecting performance monitoring functionality in kernel functions.
- Updating DynAMOS itself.

Chapter 2. Approach

DynAMOS is founded on a new dynamic instrumentation technique called *adaptive function cloning*. Dynamic instrumentation is the technique of inserting code instruments directly in the memory image of a running system.

2.1. Related Work

In support of dynamic updates, some operating systems have been designed from scratch to be adaptable or hot-swappable. Examples include *SPIN*, the *Exokernel* and *K42*. These solutions require significant changes in the way the operating system and applications are crafted. They cannot be generally applied to commodity operating systems like Linux without kernel source code modifications.

Dynamic instrumentation systems like *KernInst* and *GILK* made it possible to instrument kernel code in fixed (e.g. SPARC) and variable (e.g. i386) instruction-length architectures respectively. However, they have not addressed the issue of dynamic software updates in general. They focus on performance profiling, including the tools *DTrace* and *Pin*.

Other dynamic software updating systems include *DYMOS* and *Ginseng*, but they are limited to userspace updates. These systems recompile user programs enabling them to be dynamically updateable. They cannot be dynamically applied to a running system.

Binary rewriters like *ATOM* and *EEL* also do not support dynamic software updates.

2.2. Architecture

DynAMOS is developed in C and assembly using the GNU toolchain, currently for the i386 architecture for Linux kernels 2.2-2.6. The mechanisms it employs can also be generally applied to fixed instruction-length architectures. It has been designed with portability in mind and can be enhanced with support for other commodity operating systems besides Linux, such as FreeBSD, Solaris, MacOS and AIX.

The system consists of a kernel module providing the dynamic updating framework and a userspace collection of tools that can build, apply and manage dynamic kernel updates using the framework.

2.3. Adaptive Function Cloning

Unlike existing dynamic instrumentation systems, DynAMOS does not apply instruments at the basic

block level, but instead updates complete functions. It uses an execution flow redirection technique that permits concurrent execution of multiple versions of a function. Updates can be autonomously applied by the kernel based on rules defined by the user. An adaptation handler can be developed that determines prior to invocation of a function the appropriate version of the function that should be called. This capability of adaptively switching between multiple function versions makes DynAMOS the first dynamically applied *adaptive* kernel updating system.

The adaptive function cloning technique provides a more flexible approach of execution flow redirection geared towards procedure updates and adaptive execution. The key differences from similar dynamic instrumentation systems are:

- Instrumentation code is not guarded by processor-state preservation logic, which alters the stack. It is directly invoked, and supplied function arguments are accessed from the stack without modifications to the updated versions.
- The kernel can continuously and autonomously determine the right time an update should occur, by executing an adaptation handler.
- Basic blocks can be bypassed. Control flow graph and register analysis can be inconclusive in code sequences that contain an indirect jump from a memory address. Runtime structural analysis cannot determine whether the data following the jump are valid instructions, dead code, or data, hence cannot guarantee that such subsequent code could be bypassed. In contrast, the starting and ending memory address of a function image available in the linker symbol table guarantee that a function can be safely modified in its entirety.
- Instruments are applied at a higher, function level. The expectation of existing systems that a kernel can be intelligently and considerably modified at the instruction micro-level without access to source code can be overly taxing on developers.

2.4. Safe Execution Flow Redirection

Execution flow redirection is accomplished by installing a trampoline in the beginning of a function. The 6-byte trampoline redirects execution to a redirection handler that offers adaptive execution, as described in Section 2.3. Trampolines may only be installed on function images that are larger than the 6-byte trampoline. Local processor interrupts are disabled during trampoline installation which guarantees the installation is safe from interrupt handlers.

It is also guaranteed that installation is safe from sleeping processes. It is not possible for a process to block midway through execution of code that would be overwritten by the trampoline. Processes in an operating system block when they explicitly call the kernel scheduler. Figure 2-1 shows an example of a routine that immediately calls the Linux scheduler on entry.

Figure 2-1. A routine that immediately blocks through a call to the scheduler.

```
void functionA()
{
  schedule();
```

```
  functionB();
}
```

When disassembled in Figure 2-2, 6 bytes are dedicated to frame management and another 5 bytes consumed by the call to the scheduler. The total of 11 bytes is less than the 6 bytes needed by the trampoline.

Figure 2-2. Disassembled output of a routine that immediately blocks through a call to the scheduler.

```
00000052 <functionA>:
  52:   55                push   %ebp
  53:   89 e5             mov    %esp,%ebp
  55:   83 ec 08          sub    $0x8,%esp
  58:   e8 fc ff ff ff    call   59 <functionA+0x7>
  5d:   e8 fc ff ff ff    call   5e <functionA+0xc>
  6c:   c9                leave
  6d:   c3                ret
```

Lets explore an extreme case in Figure 2-3 where a routine is highly-optimized. The call instruction consumes 5 bytes, and could still be updated by a 5 byte trampoline that used direct addressing. In all three cases, in a fixed instruction-length architecture (e.g. PowerPC) the call to the scheduler is overwritten by the trampoline as a single instruction and ensures safe instrumentation in sleeping processes.

Figure 2-3. Disassembled output of a highly optimized version of the routine using the gcc arguments `-Os` (optimize for size) and `-fomit-frame-pointer` (omit the frame pointer).

```
  24:   e8 fc ff ff ff    call   25 <functionA+0x1>
  29:   e8 fc ff ff ff    call   2a <functionA+0x6>
  33:   e9 fc ff ff ff    jmp    34 <function_D+0xb>
```

> **Note:** There are plans (http://bugzilla.mkgnu.net/show_bug.cgi?id=989) to replace indirect branch addressing with direct branch addressing. This will reduce the performance overhead due to the processors poor trace-cache engine performance (branch mispredictions) and additionally reduce the trampoline size to 5 bytes.

> **Note:** There are plans (http://bugzilla.mkgnu.net/show_bug.cgi?id=16) to support safe execution flow redirection in multi-processor systems.

2.5. Version Manager

The list of possible updates is maintained in a version database inside the kernel module. This version manager tracks the list of possible different versions of a particular function. All updates must first be registered with the version manager before they can be dynamically applied.

By design, the original version of a function (the one that is active since bootup) must be the first version of a function that is registered with the version manager. This action installs the execution flow redirection mechanism. Alternate versions can only be registered after execution flow can be redirected away from the original.

> **Caution**
>
> Registering an alternate version of a function first, will result in the execution flow redirection mechanism applied to the alternate function instead of the original. This would render the redirection ineffective.

2.6. Safe Removal

Unregistering functions from the version manager is safeguarded by a check for quiescence. Quiescence is detected by monitoring entrance and exit to functions using use counters. However, usage counters are not always sufficient. Some functions, like `do_exit` in Linux, never return. An exit use counter would never be decremented, and quiescence would never be detected. Functions that never return lack a `ret` instruction at the end. For those functions a different methodology of examining the stack is applied. For all processes in the system, a copy of their stack pointer (`%esp`) is decremented until its value reaches the bottom of the stack. If the item pointed to by the stack pointer (the top-most 4-byte value), when interpreted as a pointer, points at an updated function the check fails. If function arguments pushed on the stack coincidentally evaluate to such a pointer address the framework conservatively assume the function is non-quiescent. This *stack walk-through* approach does not require a kernel compiled specially. For example, it will work for kernels compiled without frame pointers.

Some functions may delay their execution complicating removal. Examples include a process sleeping in a device driver waiting for response from hardware, or a process sleeping indefinitely on `sys_wait` while waiting for a child process to exit. DynAMOS first removes the trampoline, enabling access to the original function. While at least one process is still using the cloned function on its stack, the framework waits for the function to exit. If after a period of time (5 secs) the cloned function has still not quiesced, removal fails and the function clone remains active. Removal can be attempted at a later time, or the framework can be instructed to continuously attempt removal until it succeeds. Waiting does not endanger safety and does not require user action. It simply delays removal.

2.7. Function Image Relocation

The adaptive function cloning technique relocates function images to make them adaptively updateable. A series of checks safeguard from and can handle potentially unsafe relocations, such as:

- **Backward branches**. A function could contain backward branches targeting the area occupied by the trampoline. A check for such branches prior to function cloning ensures that a possibly unsafe branch is avoided. The callers of functions that contain backward branches could be updated instead to directly invoke a newer version.
- **Data-in-code**. Dynamic instrumentation systems generally cannot handle self-modifying code, code-in-data or data-in-code without a significant slowdown that is unsuitable for an operating system kernel.

  For example, Linux uses the custom `BUG` macro to produce code raising an exception on a failed assertion. `handle_BUG` handles the exception by extracting the line number and a pointer to the file name containing the failed assertion. As shown in Figure 2-4, this information is stored as data in code right after the `ud2` instruction that raises the exception.

  Figure 2-4. Definition of `BUG` macro in Linux 2.4

  ```
  #define BUG()                 \
  asm volatile( "ud2\n"         \
                "\t.word %c0\n" \
                "\t.long %c1\n" \
                : : "i" (__LINE__), "i" (__FILE__))
  ```

  The relocation logic may be misled to interpret the data as outbound branch instructions and refuse to update the function. Such conservative handling of data-in-code cases do not compromise the correctness of the updating framework.

- **Absolute memory addresses**. Kernel code is sometimes written to contain references to absolute memory addresses in the kernel image.

  One example is the `switch_to` macro in Linux which performs context switiching. As shown in Figure 2-5, it stores the value of the program counter (EIP) that a process will use in the future when it receives the processor again. When disassembled in Figure 2-6, the macro produces an absolute memory address pointing to the original function image and would break the redirection.

  Figure 2-5. Definition of `switch_to` macro in Linux 2.4

  ```
  #define switch_to(prev,next,last) do {        \
  asm volatile(...                              \
               "movl $1f,%1\n\t" /* save EIP */ \
               "pushl %4\n\t"                   \
               "jmp __switch_to\n"              \
               "1:\t"                           \
  ```

```
            "popl %%ebp\n\t"                \
            ...);
```

Figure 2-6. Disassembled output of `switch_to` macro in Linux 2.4

```
0xc01127f1   movl   $0xc0112806,0x274(%eax)
0xc01127fb   pushl  0x274(%esi)
0xc0112801   jmp    0xc0107120 <__switch_to>
0xc0112806   pop    %ebp
```

DynAMOS inspects the original functions and detects uses of literals that happen to correspond to absolute memory addresses within the memory image of the original function. It acts conservatively and warns the user of such data uses. If permitted, it adjusts the address for relocation.

- **Outbound branches**. Kernel code is sometimes written to contain outbound branches from one function to a common section of code and a branch back that continues execution.

For example, Linux produces semaphore and locking code in this way. In the `down` semaphore acquire operation shown in Figure 2-7, an atomic counter decrement checks if the semaphore is still in use. In the common case where it's not, execution falls through for improved performance. If the semaphore is in use, `__down_failed` is called. The pair of `LOCK_SECTION` macros insert linker directives that place the uncommon call to `__down_failed` in a separate memory area. The matching assembly produced for this sequence, when used with `pipe_release`, is shown in Figure 2-8. On a failure to acquire a semaphore, execution jumps to global table `Letext`. A wrapper `call` to `__down_failed` is issued, with a subsequent `jmp` back to the main `pipe_release` code.

Figure 2-7. `down` semaphore acquire operation in Linux 2.4

```
static inline void down(struct semaphore * sem)
{
  __asm__ __volatile__(
    "# atomic down operation\n\t"
    LOCK "decl %0\n\t"     /* --sem->count */
    "js 2f\n"
    "1:\n"
    LOCK_SECTION_START("")
    "2:\tcall __down_failed\n\t"
    "jmp 1b\n"
    LOCK_SECTION_END
    :"=m" (sem->count)
    :"c" (sem)
    :"memory");
}
```

Figure 2-8. Disassembled output of `down` semaphore acquire operation in Linux 2.4

```
 __asm__ __volatile__(
  281:          mov    %eax,%ecx
  283:          decl   0x6c(%esi)
  286:          js     158b <Letext+0x3c>
     down(PIPE_SEM(*inode));
  PIPE_READERS(*inode) -= decr;
  28c:          mov    0x108(%esi),%eax
  292:          sub    %edx,0x18(%eax)
  ...
0000154f <Letext>:
  ...
  158b:         call   158c <Letext+0x3d>
  1590:         jmp    28c <pipe_release_v2+0x1c>
  ...
```

If outbound branches are relocated, the `jmp` back to the function image will divert execution flow from a cloned function to its original. DynAMOS detects such wrapper code outbound jumps and relocates their `call/jmp` pairs at the end of the function image, adjusting their relative offsets.

- **Indirect outbound branches**. Compilers sometimes produce code that uses an indirection table when C switch statements or multiple if statements are used.

  For example, when `gcc` compiles a C function containing a `switch` statement with more than 4 `case` options it produces code that uses an indirection table. The table is dereferenced with an indirect jump to determine the next value of the program counter. An example table is found in `do_signal` in Linux. DynAMOS inspects indirect branches to detect indirection tables. The table inspection stops at the first 4-byte table entry whose target address falls outside the range of the original function. The indirection tables identified are relocated at the end of the new function image.

- **Multiple entrypoints**. Compilers sometimes produce functions that contain multiple entrypoints as an optimization.

  For example, `icc` produces multiple entrypoints for some functions as a result of an interprocedural constant propagation optimization. Functions are split between a prologue and a core for a total of two symbols per function. As shown in Figure 2-9, the prologue code <`filp_open`> is the safe entrypoint which moves function arguments from the stack into registers. It does not contain a `ret` and falls through to the core <`filp\_open.`>. Callees invoke either the prologue or the core accordingly.

Figure 2-9. Multiple entrypoints produced by `icc`

```
c01505f8 <filp_open>:
c01505f8:   8b 44 24 04    mov   0x4(%esp,1),%eax
c01505fc:   8b 54 24 08    mov   0x8(%esp,1),%edx
c0150600:   8b 4c 24 0c    mov   0xc(%esp,1),%ecx
```

```
c0150604 <filp_open.>:
c0150604:   55              push  %ebp
c0150605:   53              push  %ebx
c0150606:   83 ec 30        sub   $0x30,%esp
...
c015064f:   5d              pop   %ebp
c0150650:   c3              ret
```

Execution flow of the core (e.g. <`filp_open.`>) in multiple entrypoints can still be redirected by applying the trampoline. However, for newer versions of the function the compiler must produce code that is again split between a prologue and a core. Prologue code (e.g. <`filp_open`>) needs to be at least 6-bytes long for the trampoline to be safely applied. For a smaller prologue, the local bounce allocation technique outlined in GILK, which DynAMOS does not yet implement, can be applied.

2.8. Symbol Resolution

The original kernel image object file is consulted (e.g. with `-R vmlinux` in Linux) when it is desired to update symbols (variables or functions) that were not exported in the original kernel source.

2.9. Datatype Updates

For some types of updates it is necessary to update the datatype of a variable. When a new field is added, there isn't any reserved room in the data structure to host the field. DynAMOS uses *shadow* variables in support of such datatype updates. On variable creation a new shadow variable is created. The memory address of the variable is used to map into it's shadow using a hash table. When the new datatype is freed, it's shadow is also freed. A benefit of this technique is that only the functions that must use the new field need to be updated, instead of all functions that use the old datatype.

2.10. Constructing Updates

Constructing kernel updates is currently a manual process. There are plans to introduce a semi-automatic tool that, given as input a diff file, will automatically produce alternate versions of the functions that need to be updated.

Updates to functions are constructed by duplicating in source code the originals, applying modifications in the source, naming the functions differently (e.g. append the postfix "`_v2`"), and recompiling them with the compiler and kernel flags used to originally compile the kernel.

Updates to variables are constructed as a function containing logic that must be executed to update the variables. This function is executed once when the update is applied.

Chapter 3. Installation

3.1. Availability

DynAMOS is available for Linux 2.2-2.6 on the i386 architecture. There are plans to port the system to FreeBSD (http://bugzilla.mkgnu.net/show_bug.cgi?id=121) and OpenSolaris (http://bugzilla.mkgnu.net/show_bug.cgi?id=901). It's webpage (http://freshmeat.net/projects/dynamos) contains the most up to date information on the project, including the latest release and manual.

A users mailing list is available for subscription (http://lists.mkgnu.net/mailman/listinfo/dynamos-users), or simply for sending email (mailto:dynamos-users@lists.mkgnu.net).

3.2. Installation

DynAMOS is available in the form of Debian and RPM packages, but is not yet available in source code form. The provided packages are:

- `dynamos-framework`: The DynAMOS system.
- `dynamos-doc`: Documentation including this manual.

Chapter 4. Framework

This section provides instructions on using the DynAMOS framework to dynamically apply updates.

4.1. Starting

The framework kernel module must first be loaded. This is accomplished as shown in Figure 4-1.

Figure 4-1. Starting the framework.

```
bash$ su -
Password:
bash# /etc/init.d/dynamos start
```

4.2. Stopping

Stopping the framework when updates are already applied could crash the system. Before stopping the framework it is recommended that any applied updates are properly deactivated.

Stopping the framework is accomplished as shown in Figure 4-2. Function updates that are already in effect will be automatically reversed, but not in any particular order that guarantees safe deactivation of the update.

Figure 4-2. Stopping the framework.

```
bash$ su -
Password:
bash# /etc/init.d/dynamos stop
```

4.3. Control Tool

The framework is managed with the `dynamos_control` tool. Additional programs and scripts are built on top of calls to this tool.

4.4. Control Commands

One way of controlling the framework is by writing shell scripts containing special macros. These scripts are supplied as an argument to the tool `dynamos_run_commands`. This tool will translate the macros and produce a new script with the postfix "`.translated.sh`" containing the actual calls to the *Control Tool*.

> **Note:** The current method of issuing control commands is weak. It does not support retrieving the return value of the control commands to react appropriately (e.g. abort the update on error). It also lacks a macro of defining a group of function updates that should be applied atomically. There are plans to replace this method with a Perl-based library of calls. Control programs will then be written in a more reliable way and in a more powerful language.

The list of the existing control command macros follows:

- DYNREPLACE_REGISTER_INTUITIVELY(update_name, id, function_name)

  This macro is used to register updates with the version manager.

  By convention, the `id` is a number indicating the number of parameters the function accepts. Multiple updates may be registered in the version manager with the same `update_name` but different `id`s. Such updates would correspond to different routines. And for each <`update_name, id`> pair, multiple `function_name`s could be registered as the alternate versions of this update. `update_name`s must be enclosed in double(") quotes.

- DYNREPLACE_DEREGISTER_FUNCTION(update_name, id, version)

  This macro is used to unregister specific versions of updates.

  Attempting to unregister version 0 will unregister all versions for the particular <`update_name, id`> pair.

- DYNREPLACE_ACTIVATE_FUNCTION(update_name, id, version)

  This macro is used to activate specific versions of updates.

  By convention, the original version of an update is version number 1. To deactivate an update, activate version number 1.

- DYNREPLACE_SET_PREACTIVATION_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed before a particular version of a specific <`update_name, id`> pair is activated.

- DYNREPLACE_SET_POSTACTIVATION_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed after a particular version of a specific <`update_name, id`> pair is activated.

- DYNREPLACE_SET_PREREMOVAL_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed before a <`update_name, id`> pair is removed from the version manager.

- DYNREPLACE_SET_POSTREMOVAL_HOOK_BY_NAME(update_name, id, version, function_name)

  This macro is used to set a hook that will be executed after a <`update_name, id`> pair is removed from the version manager.

- DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_NAME(update_name, id, version, function_name)

  This macro is used to define an adaptation handler. There can be only one adaptation handler per <`update_name, id`> pair.

- DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_ADDRESS(update_name, id, version, memory_address)

  This macro is also used to define an adaptation handler. Supplying a `memory_address` of `0` will remove an existing adaptation handler.

- DYNREPLACE_CALL(function_name)

  This macro is used to invoke an initialization function.

Figure 4-3 shows parts of actual control commands used to enable the Linger-Longer system, adaptive updating of the `pipefs` implementation, and enable EPCKPT process checkpointing.

Figure 4-3. Example control commands.

```
# Linger-Longer -- Update the kswapd thread
DYNREPLACE_REGISTER_INTUITIVELY("interruptible_sleep_on", 1, interruptible_sleep_on)
DYNREPLACE_REGISTER_INTUITIVELY("interruptible_sleep_on", 1, interruptible_sleep_on_v2)
DYNREPLACE_ACTIVATE_FUNCTION("interruptible_sleep_on", 1, 2)

DYNREPLACE_REGISTER_INTUITIVELY("kswapd", 1, kswapd)
DYNREPLACE_REGISTER_INTUITIVELY("kswapd", 1, kswapd_ll)
DYNREPLACE_SET_PREACTIVATION_HOOK_BY_NAME("kswapd", 1, 1, kswapd_pre_activation_hook)
DYNREPLACE_SET_POSTACTIVATION_HOOK_BY_NAME("kswapd", 1, 1, kswapd_post_activation_hook)
DYNREPLACE_SET_PREACTIVATION_HOOK_BY_NAME("kswapd", 1, 2, kswapd_ll_pre_activation_hook)
DYNREPLACE_SET_POSTACTIVATION_HOOK_BY_NAME("kswapd", 1, 2, kswapd_ll_post_activation_hook
DYNREPLACE_ACTIVATE_FUNCTION("kswapd", 1, 2)

# pipefs -- create an adaptation handler
DYNREPLACE_REGISTER_INTUITIVELY("pipe_read", 4, pipe_read)
DYNREPLACE_REGISTER_INTUITIVELY("pipe_read", 4, pipe_read_v2)
DYNREPLACE_REGISTER_INTUITIVELY("pipe_read", 4, pipe_read_v3)
DYNREPLACE_ACTIVATE_FUNCTION("pipe_read", 4, 2)
DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_NAME("pipe_read", 4, pipe_adaptation_handler_r

# Disable the adaptation handler
DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_ADDRESS("pipe_write", 4, 0)
DYNREPLACE_SET_RULE_EVALUATION_FUNCTION_BY_ADDRESS("pipe_read", 4, 0)

# Unregister some functions
DYNREPLACE_DEREGISTER_FUNCTION("pipe_write",  4, 0)
DYNREPLACE_DEREGISTER_FUNCTION("pipe_read",  4, 0)
DYNREPLACE_DEREGISTER_FUNCTION("pipe_release",  3, 0)

# EPCKPT -- Set a preremoval hook
DYNREPLACE_SET_PREREMOVAL_HOOK_BY_NAME("restart_binary", 2, epckpt_cleanup)

# Invoke an initialization function
DYNREPLACE_CALL(epckpt_init)
```

4.5. Writing Adaptation Handlers

An API is available for writing adaptation handlers. Calls to activate a different version of a function, or query the framework for presence of a specific version of a function are available. Part of the function signature of the adaptation handler is defined by the framework, and the remaining arguments match the original arguments of the function on which the adaptation handler is applied.

Figure 4-4 shows the signature of `pipe_write`, the producer function of `pipefs` in Linux 2.4. Figure 4-5 shows an example adaptation handler written for `pipe_write`. The arguments supplied to the original `pipe_write` are still accessible on the stack and can be used to determine which version of `pipe_write` should run next. In this (simplified) example, when more than 64K of data are written through a pipe the second version of `pipe_write` is used.

Figure 4-4. Function signature of `pipe_write`.

```
static ssize_t
pipe_write(struct file *filp, const char *buf,
           size_t count, loff_t *ppos)
```

Figure 4-5. Example adaptation handler for `pipe_write`.

```
long pipe_write_total_count = 0;

void
pipe_write_adaptation_handler(dynreplace_version_table_entry_t *entry,
                              redirection_state_t redir_state,
                              rule_evaluation_call_state_t rec_state,
                              struct file *filp, char *buf,
                              size_t count, loff_t *ppos)
{
  pipe_write_total_count += count;
  dynreplace_version_table_lock();

  if (pipe_write_total_count > 64 * 1024)
    dynreplace_activate_function(&entry->unique, 2);
  else
    dynreplace_activate_function(&entry->unique, 1);

  dynreplace_version_table_unlock();
}
```

4.6. Updating Datatypes

Datatype mappings are maintained in a separate hash table per datatype. The table must be initialized before applying an update, and freed when reversing the update, as shown in Figure 4-6. Figure 4-7 shows a set of new fields that should be added in the existing datatype definition of `struct task_struct`, the process control block in Linux 2.4. All these fields are grouped in a new datatype definition called `struct new_task_struct`, instead of extending the existing definition.

Figure 4-6. Managing a datatype mapping table.

```
/* Mapping table for the updated datatype of struct task_struct. */
```

```
dynreplace_access_t access_new_task_struct;

void epckpt_init()
{
  dynreplace_access_init(&access_new_task_struct);
}

void epckpt_cleanup()
{
  dynreplace_access_cleanup(&access_new_task_struct);
}
```

Figure 4-7. Definition of `struct new_task_struct`.

```
struct new_task_struct {
int collect_ckpt_data:1;
struct mmap_list *mmap_list;
struct shmem_list *shmem_list;
struct sem_list *sem_list;
};
```

A datatype definition, like the one in Figure 4-7, may define the initial values fields will need to be initialized to. This initialization must be carried out manually. Shadow variables must be instantiated when the original datatype is instantiated, and a mapping between the original and its shadow must be established. Figure 4-8 shows an example creating a shadow variable instance on `do_fork` which will hold the new fields of `struct new_task_struct` as defined in Figure 4-7. A shadow instance is created, and a reference to the shadow is obtained. The reference is used to access the new fields.

Figure 4-8. Creating a shadow variable in `do_fork`.

```
int do_fork_v2(unsigned long clone_flags, unsigned long stack_start,
       struct pt_regs *regs, unsigned long stack_size)
{
  ...
  struct task_struct *p;
  void *new_p;

  ...

  dynreplace_access_lock(&access_new_task_struct);

  /* Create a shadow instance for this task_struct */
  dynreplace_access_create(&access_new_task_struct,
                           (void *)p, sizeof(struct new_task_struct));

  /* Obtain a reference to the shadow instance */
  new_p = dynreplace_access_find(&access_new_task_struct, (void *)p);

  /* Initialize the instance */
```

```
  if (new_p != NULL) {
    /* Access the various fields */
    dynreplace_access_field(struct new_task_struct, new_p, collect_ckpt_data) = 1;
    dynreplace_access_field(struct new_task_struct, new_p, mmap_list) = NULL;
    dynreplace_access_field(struct new_task_struct, new_p, shmem_list) = NULL;
    dynreplace_access_field(struct new_task_struct, new_p, sem_list) = NULL;

    /* Use the new fields to change control flow as required by the update. */
    if (dynreplace_access_field(struct new_task_struct, new_p, collect_ckpt_data)) {
      /* Perform process checkpointing bookkeeping */
      ...

    }
  }

  dynreplace_access_unlock(&access_new_task_struct);

 ...

}
```

Shadow variable instances must be freed when the original datatype is freed. Figure 4-9 shows an example freeing the shadow variable instance on `do_exit`.

Figure 4-9. Freeing a shadow variable in `do_exit`.

```
ATTRIB_NORET NORET_TYPE void do_exit_v2(long code)
{
  struct task_struct *tsk = current;

 ...


  {
     void *new_current;

     dynreplace_access_lock(&access_new_task_struct);

     new_current = dynreplace_access_find(&access_new_task_struct, (void *)current);
     if (new_current != NULL)
       dynreplace_access_remove(&access_new_task_struct, (void *)current);

     dynreplace_access_unlock(&access_new_task_struct);
  }

 ...

}
```

On shadow variable instance creation and deletion the code that accesses the new fields and the code that frees the shadow variable is written in a manner that permits existing kernel code paths to execute without dereferencing a NULL pointer. For example, a process executing `do_exit` that has not had a shadow variable instance created (because when the process was created the update had not been applied) will still execute normally. Similarly, code in other functions that uses the new fields should not be executed if there is no shadow instance available. In essence, it is possible to have multiple versions of a function concurrently executing safely.

4.7. Control Center

There are plans (http://bugzilla.mkgnu.net/show_bug.cgi?id=32) to develop a GUI tool that controls the framework.

4.8. Remote Control Service

There are plans (http://bugzilla.mkgnu.net/show_bug.cgi?id=990) to offer a service that can handle remote update requests and control the framework over the network.

Chapter 5. Examples

DynAMOS is distributed with examples of kernel updates in source code form for Linux 2.4. The source code can be compiled and prepared to be applied as an update. The updated functionality can be loaded in the kernel and activated using the DynAMOS framework. The following examples demonstrate how the collection of tools offered by DynAMOS are used together from start to finish in dynamically applying a kernel update.

The general format of running these examples is shown in Figure 5-1.

Figure 5-1. Running the example updates.

```
# DynAMOS must run with administrator privileges.
bash$ su -
Password:
bash#

# Start the framework
bash# /etc/init.d/dynamos start

# Check if everything started correctly
bash# dmesg -c

# Build an example's source
bash# cd /usr/share/doc/dynamos/examples/get_pid
bash# make

# Load the example's module
bash# insmod final_dynreplace_file.o

# Activate the update
bash# ./activate.sh

# Verify things work
bash# dmesg -c
bash# cat /dev/dynamos

# Deactivate the update
bash# ./deactivate.sh

# Verify the update is not effective
bash# dmesg -c
bash# cat /dev/dynamos

# Unload the example's module
bash# rmmod final_dynreplace_file

# Stop the framework
bash# /etc/init.d/dynamos stop
```

```
# Check if everything stopped correctly
bash# dmesg -c
```

5.1. Updating `get_pid`

`get_pid()` is the Linux process allocation routine. It returns the next available process id that can be used by a newly created process. When applied, this update will report in the kernel logs on process creation the id of the process that calls `get_pid()` and the pid it returns, as shown in Figure 5-2.

Figure 5-2. Expected output of `get_pid()` update.

```
This is get_pid_v2 from 3012 and will return pid 3018.
```

5.2. Updating The Process Scheduler

`schedule()` is the Linux process scheduler. When applied, this update will report an informative message in the kernel logs after every 5000 invocations of the scheduler, as shown in Figure 5-3.

Figure 5-3. Expected output of `schedule()` update.

```
schedule_v2 called for 5000 times (25000 times total), this time from 3174.
```

Appendix A. FAQ

This FAQ includes questions not covered elsewhere in this manual.

Glossary

A

ATOM

ATOM is a library for modifying object files.

D

DTrace

DTrace (http://www.sun.com/bigadmin/content/dtrace/) is a comprehensive dynamic tracing framework for the Solaris Operating Environment. DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs.

DYMOS

DYMOS (http://www.cis.upenn.edu/~lee/mydissertation.doc) is a dynamic modification system.

E

EEL

EEL (http://www.cs.wisc.edu/~larus/eel.html) is a C++ library that hides much of the complexity and system-specific detail of editing executables. EEL provides abstractions that allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code. EEL greatly simplifies the construction of program measurement, protection, translation, and debugging tools. EEL differs from other systems in two major ways: it can edit fully-linked executables, not just object files, and it emphasizes portability across a wide range of systems.

Exokernel

Exokernel (http://pdos.csail.mit.edu/exo.html) is the name of an operating system kernel developed by the Parallel and Distributed Operating Systems group at MIT, and of a class of similar operating systems. An exokernel eliminates the notion that an operating system should provide abstractions on which applications are built. Instead, it concentrates solely on securely multiplexing the raw hardware: from basic hardware primitives, application-level libraries and servers can directly implement traditional operating system abstractions, specialized for appropriateness and speed.

G

GILK

GILK (http://www.doc.ic.ac.uk/~djp1/gilk.html) is a dynamic instrumentation tool for the Linux 2.2 kernel.

Ginseng

Ginseng (http://www.cs.umd.edu/projects/dsu/) is a dynamic updating system for userspace programs.

K42

K42

K42 (http://www.research.ibm.com/K42/) is a high performance, open source, general-purpose research operating system kernel for cache-coherent multiprocessors.

KernInst

KernInst (http://www.paradyn.org/html/kerninst.html) is a framework for dynamically splicing code into a running kernel, almost anywhere, anytime.

P

Pin

Pin (http://rogue.colorado.edu/pin/) is a tool for the dynamic instrumentation of programs. Pin was designed to provide functionality similar to the popular ATOM toolkit for Compaq's Tru64 Unix on Alpha, i.e. arbitrary code (written in C or C++) can be injected at arbitrary places in the executable. Unlike ATOM, Pin does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

S

SPIN

SPIN (http://www-spin.cs.washington.edu/) is an operating system that blurs the distinction between kernels and applications. Applications traditionally live in user-level address spaces, separated from kernel resources and services by an expensive protection boundary. With SPIN, applications can specialize the kernel by dynamically linking new code into the running system. Kernel extensions can add new kernel services, replace default policies, or simply migrate application functionality into the kernel address space. Sensitive kernel interfaces are secured via a restricted linker and the type-safe properties of the Modula-3 programming language. The result is a flexible operating system that helps applications run fast but doesn't crash.