

[Announce] Non Invasive Kernel Monitor for threads/processes

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-06/3379.html>

From: Atul Sabharwal (atul_sabharwal_at_linux.jf.intel.com)

Date: 06/15/04

Date: Tue, 15 Jun 2004 12:54:23 -0700

To: linux-kernel@vger.kernel.org

We have been working with a solution for non-intrusively trapping on lifetime of processes/threads. This is useful for management applications running in telecom and enterprise data centers that need to monitor a set of threads/processes.

Project Goal::

~~~~~

To create a kernel patch that shall support methods to non-intrusively monitor processes/threads at the kernel level. It would use a notification mechanism in the kernel that allows registration of events of interest regarding processes/threads. Events of interest could be the following : Process creation (fork), Process exit(exit), Process calls(exec), thread creation & thread exit. The implementation needs to provide high performance with low overhead. Monitoring would be driven directly by the events in the kernel to ensure low latency access to interesting events.

## Solution Approach:

~~~~~

This method requires a user space process (called monitor) to start and stop monitoring in the system. The monitor sends the list of pids and the events to be monitored for that pid. Once the ioctl is received, the kernel establishes a filter for it. Events get flooded in the system as each call (fork, exec and exit) has been modified to generate events. The events which match the filter become interesting and require that a notification is sent to the monitor

Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

process. Events which do not match the filter get dropped. In the case when we have multiple monitor processes, signals are sent to each monitor whose filters match the event which has occurred.

Solution Description :

~~~~~  
The registration and de–registration command are implemented as ioctls. The notification command is implemented as a sigqueue signal. The system calls such as fork, exit, and exec have been changed to add a notification call to the monitor process. The filter for events is populated with the registration command. Events & their respective thread/process get removed from the filter with the de–registration command.

Threads and processes follow the same code path for task creation & termination. Hence, the code path is the same for threads and processes. There is no equivalent of an exec call with threads. The only distinction for a thread is that in the task struct, the thread group id (tgid) and process id (pid) are different. For a process, they are the same.

The signal used for notification is also specified with the registration command. If it is a real–time signal, it would perform better as signals would get queued for the monitor. With non–real time signals, signals would get dropped if one signal exists. Hence, it is recommended to specify a real time signal. Also, it is to be noted that the solution is totally architecture independent. So, would just need a re–compile on the respective platform.

Kmonitor has been implemented as a miscellaneous character driver built into the Linux kernel. It cannot be compiled as a loadable module. It supports the relevant driver calls for open, release, ioctl, init, cleanup calls. The character driver code contains the filtering, registration/de–registration and notification functions of Kmonitor as well as routines to manage the linked list(s) for filtering.

Kmonitor has 256 buckets and each bucket contains a spinlock and a linked list. This works around the sequential search operation in linked list by

## Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

using a hashing function to change the complexity from  $O(N)$  to  $O(N/M)$  where  $M$  is the number of buckets.

Kernel Version & Code location:

~~~~~  
The kernel used was the 2.6.6 kernel and the code is at <http://kmonitor.bkbits.net>

User space programs:

~~~~~  
1. unit\_test : A sample program to monitor future processes.

[http://kmonitor.bkbits.net:8080/tools/anno/unit\\_test.c@1.4?nav=index.html|ChangeSet@-3w|cset@1.35](http://kmonitor.bkbits.net:8080/tools/anno/unit_test.c@1.4?nav=index.html|ChangeSet@-3w|cset@1.35)

2. monitor\_pid: A sample program to monitor a process sub–tree.

[http://kmonitor.bkbits.net:8080/tools/anno/monitor\\_pid.c@1.2?nav=index.html|ChangeSet@-2w|cset@1.37](http://kmonitor.bkbits.net:8080/tools/anno/monitor_pid.c@1.2?nav=index.html|ChangeSet@-2w|cset@1.37)

3. wait\_on\_pid : A sample program which waits till a pid dies.

[http://kmonitor.bkbits.net:8080/tools/anno/wait\\_on\\_pid.c@1.5?nav=index.html|ChangeSet@-2w|cset@1.37](http://kmonitor.bkbits.net:8080/tools/anno/wait_on_pid.c@1.5?nav=index.html|ChangeSet@-2w|cset@1.37)

Source Code:

~~~~~  
The patch for Kmonitor is as below :

```
diff -Nru a/Documentation/ioctl-number.txt b/Documentation/ioctl-number.txt
--- a/Documentation/ioctl-number.txt 2004-06-14 11:24:08 -07:00
+++ b/Documentation/ioctl-number.txt 2004-06-14 11:24:08 -07:00
@@ -189,3 +189,4 @@
     <mailto:michael.klein@puffin.lb.shuttle.de>
0xDD 00-3F ZFCP device driver see drivers/s390/scsi/
     <mailto:aherrman@de.ibm.com>
+0xDE all drivers/char/kmonitor <mailto:atul.sabharwal@intel.com>
diff -Nru a/Documentation/kmonitor.txt b/Documentation/kmonitor.txt
--- /dev/null Wed Dec 31 16:00:00 196900
+++ b/Documentation/kmonitor.txt 2004-06-14 11:24:08 -07:00
@@ -0,0 +1,119 @@
+Enabling Non-intrusive Application Monitoring with kmonitor
+-----
+The kmonitor driver implements a kernel level non-intrusive +mechanism
for rapidly notifying a monitoring user space process
+of key (process/thread creation/exit) kernel events.
+
+The monitoring user space process registers and deregisters for events via
+a new misc char device using two new ioctl's. The kernel notifies the
+monitoring process(es) of the events via a signal that the monitor
specifies
+during registration.
+
```

Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+An example of how this would be used in an minimal command line utility
+to wait for an arbitrary process to exit:
+
+<snip>rest of required headers...</snip>
+#include <linux/kmonitor.h>
+
+void target_exit_handler (int signum , struct siginfo * info, void * buf)
+{
+ /*
+ * Since we only registered for one type of event,
+ * KMONITOR_PROCESS_EXIT, on one process, then we already + *
+ know our target process has exited
+ *
+ * If we were doing something more complex, then the data in
+ * info might be usefull:
+ * 1. info->si_pid: process id of target process
+ * 2. info->si_gid: group id of target process
+ * 3. info->si_int: event that triggered this signal
+ * (See include/linux/kmonitor.h for the list of events)
+ *
+ * The target process has exited, so we are done. Kmonitor
+ * will do cleanup when we close the file descriptor for + *
+ /dev/kmonitor (which will happen automatically when we
+ * exit.)
+ */
+ exit(0);
+}
+
+int main(int argc, char *argv[])
+{
+<snip>...</snip>
+
+ /*
+ * kmonitor will notify us that the target process has exited
+ * by sending us a real–time signal with the payload containing
+ * the event that triggered the notification.
+ */
+ action.sa_sigaction = target_exit_handler;
+ action.sa_flags = SA_SIGINFO;
+ if (-1 == sigaction(SIGRTMIN, &action, NULL)) {
+ perror("sigaction");
+ exit(-1);
+ }
+
+ /*
+ * The kmonitor is implemented as misc char device, so the
+ * file that needs to be opened (assumed to be /dev/kmonitor
+ * in this example) is a char device with a major number of 10
+ * and a minor number of 221.
+ * + * mknod /dev/kmonitor c 10 221 + */
+ fd = open("/dev/kmonitor", O_WRONLY);
```

```

+ if (-1 == fd) {
+ perror("open /dev/kmonitor");
+ exit(-1);
+ }
+
+ /*
+ * To sign up for notification of an arbitrary event on an
+ * arbitrary process then we send the KMONITOR_IOW_REGISTER
+ * command to the device via the ioctl system call.
+ *
+ * We indicate the target process and event by sending a
+ * kmonitor_cmd structure.
+ *
+ * NOTE: The 'type' field in the structure is actually +
+ * a bitmask, so it is possible to register for
+ * multiple events on a given target process in one
+ * ioctl call. The possible events are...
+ * #define KMONITOR_THREAD_CREATE 0x01 + *
+ * #define KMONITOR_THREAD_ABORT 0x02 + * #define
+ * KMONITOR_THREAD_EXIT 0x04
+ * #define KMONITOR_PROCESS_FORK 0x08
+ * #define KMONITOR_PROCESS_ABORT 0x10 + *
+ * #define KMONITOR_PROCESS_EXEC 0x20 + * #define
+ * KMONITOR_PROCESS_EXIT 0x40
+ */
+ hdr.signal = SIGRTMIN;

+ hdr.pid = args.pid;
+ hdr.type = KMONITOR_PROCESS_EXIT;
+ if (-1 == ioctl(fd, KMONITOR_IOW_REGISTER, &hdr)) {
+ perror("ioctl");
+ exit(-1);
+ }
+
+ /*
+ * In this very simple example all we need to do is wait + *
+ * for a signal to be sent. The signal handler will exit
+ * when it is notified of the target process exiting.
+ */
+ sigemptyset(&empty_mask);
+ sigsuspend(&empty_mask);
+
+ /*
+ * kmonitor will perform a cleanup of all request a given + *
+ * monitoring process has made, so it is required to deregister
+ * each of the specific request.
+ *
+ * If for some reason a monitoring process wanted to deregister
+ * a specific request, then the procedure is just like registering,
+ * other then the ioctl command is KMONITOR_IOW_DEREGISTER.
+ */

```

Linux-Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+ close(fd);
+ exit (0);
+}
+
+
diff -Nru a/drivers/char/Kconfig b/drivers/char/Kconfig
--- a/drivers/char/Kconfig 2004-06-14 11:24:08 -07:00
+++ b/drivers/char/Kconfig 2004-06-14 11:24:08 -07:00
@@ -974,5 +974,15 @@
     out to lunch past a certain margin. It can reboot the system
     or merely print a warning.
```

```
+config KMONITOR
+ bool "Kmonitor driver"
+ help
+ The kmonitor driver implements a kernel level non-intrusive
+ mechanism for rapidly notifying a monitoring user space process
+ of key (process/thread creation/exit) kernel events. For more
+ info see Documentation/kmonitor.txt +
+ If in doubt, say 'N'.
+
+endmenu
```

```
diff -Nru a/drivers/char/Makefile b/drivers/char/Makefile
--- a/drivers/char/Makefile 2004-06-14 11:24:08 -07:00
+++ b/drivers/char/Makefile 2004-06-14 11:24:08 -07:00
@@ -79,6 +79,7 @@
obj-$(CONFIG_DRM) += drm/
obj-$(CONFIG_PCMCIA) += pcmcia/
obj-$(CONFIG_IPMI_HANDLER) += ipmi/
+obj-$(CONFIG_KMONITOR) += kmonitor.o

obj-$(CONFIG_HANGCHECK_TIMER) += hangcheck-timer.o
```

```
diff -Nru a/drivers/char/kmonitor.c b/drivers/char/kmonitor.c
--- /dev/null Wed Dec 31 16:00:00 196900
+++ b/drivers/char/kmonitor.c 2004-06-14 11:24:08 -07:00
@@ -0,0 +1,336 @@
+/*
+ * linux/kmonitor.c
+ *
+ * Copyright (C) 2004 Intel Corporation. All rights reserved.
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public
+ * License v2.0 as published by the Free Software Foundation; + *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
+ * General Public License for more details.
+ *
+ */
```

Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+ * You should have received a copy of the GNU General Public
+ * License along with this program; if not, write to the
+ * Free Software Foundation, Inc., 59 Temple Place – Suite 330,
+ * Boston, MA 02110–1307, USA.
+ *
+ * Authors: Atul Sabharwal <atul.sabharwal@intel.com>
+ *
+ * The kmonitor driver implements a kernel level non–intrusive + *
mechanism for rapidly notifying a monitoring user space process
+ * of key (process/thread creation/exit) kernel events.
+ * + * See Documentation/kmonitor.txt for more details
+ */
+
+#include <linux/config.h>
+#include <linux/module.h>
+#include <linux/moduleparam.h>
+#include <linux/fs.h>
+#include <linux/hash.h>
+#include <linux/init.h>
+#include <linux/kmonitor.h>
+#include <linux/miscdevice.h>
+#include <linux/proc_fs.h>
+#include <linux/sched.h>
+#include <linux/types.h>
+#include <linux/timer.h>
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+
+/*
+ * kmonitor maintains a database of resources containing:
+ * – the pid of the process being monitored (target)
+ * – the pid of process that should be notified (monitor)
+ * – an event mask + * – a signal number to notify the monitor with
+ * – a list_head
+ * + * To improve scalability, a hash of linked list is maintained
+ * with a per bucket read–write lock. KMONITOR_BITS defines the
+ * order of the list size, so setting KMONITOR_BITS to 8 would translate
+ * to 256 buckets. Using a hash of linked list improves the order of the
+ * search from an O(n) to O(n/m) where n is the number of resources and
+ * m is the number of buckets.
+ *
+ * Note that since we are using a per bucket read–write lock, the number
+ * of buckets also has an effect on SMP performance since the smaller the
+ * list of resources in each bucket, the less likely a processor will +
+ * spin waiting for another processor to finish writing to the same list.
+ */
+
+struct kmonitor_bucket +{
+ struct list_head list;
+ rwlock_t lock;
+};
```

Linux-Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+
+#define KMONITOR_BITS 0x8
+
+struct kmonitor_bucket kmonitor_resources[1<<KMONITOR_BITS];
+
+static inline unsigned long khash(pid_t target)
+{
+ return hash_long((unsigned long)target, KMONITOR_BITS);
+}
+
+struct kmonitor_res
+{
+ struct list_head rlist;
+ pid_t target;
+ pid_t monitor;
+ int signal;
+ __u32 event_mask;
+};
+
+#define to_kmonitor_res(r) \
+ container_of(r, struct kmonitor_res, rlist);
+
+kmem_cache_t *kmonitor_cache;
+
+/*
+ * kmonitor_active is used to improve the performance of searching
+ * an empty resource database
+ */
+atomic_t kmonitor_active = { .counter = 0 };
+
+/* #define DEBUG */
+#ifdef DEBUG
+#define DBG(format, arg...) printk("%s: " format "\n", __FUNCTION__ ,
## arg)
+#define TRACE(format, arg...) printk("%s(" format ")\n", __FUNCTION__,
## arg)
+#else
+#define DBG(format, arg...) do { } while(0);
+#define TRACE(arg...) do { } while(0);
+#endif
+
+static int kmonitor_add(pid_t target, int type, int signal)
+{
+ struct kmonitor_res *r;
+ struct list_head *tmp;
+ unsigned long bucket = khash(target);
+
+ TRACE("%i, %i, %i", target, type, signal);
+ read_lock(&kmonitor_resources[bucket].lock);
+ list_for_each(tmp, &kmonitor_resources[bucket].list) {
+ r = to_kmonitor_res(tmp);
```

Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+ if (r->monitor == current->pid && r->target == target) {
+ /*
+ * There is an existing resource for this
+ * target/monitor combination so just flip + *
+ * a bit in the event mask indicating that + * this monitor
+ * also wants to be notified + * for this type of event.
+ */
+ r->event_mask |= type;
+ r->signal = signal;
+ DBG("Added %i to existing res", type);
+ read_unlock(&kmonitor_resources[bucket].lock);
+ return 0;
+ } + }
+ read_unlock(&kmonitor_resources[bucket].lock);
+
+ /*
+ * This is the first event registered for this target + * on
+ * this monitor. Allocate a new resource from our + * cache and add
+ * the resource to the global list.
+ *
+ * Note: There is no race condition between the top half & the
+ * bottom part of this function as if multiple monitors are
+ * active,
+ * each will have a unique resource for itself. So, the function
+ * is SMP safe.
+ */
+ r = kmem_cache_alloc(kmonitor_cache, SLAB_KERNEL);
+ if (!r)
+ return -ENOMEM;
+
+ r->target = target;
+ r->monitor = current->pid;
+ r->event_mask = type;
+ r->signal = signal;
+ write_lock(&kmonitor_resources[bucket].lock);
+ list_add(&r->rlist, &kmonitor_resources[bucket].list);
+ write_unlock(&kmonitor_resources[bucket].lock);
+ return 0;
+ }
+
+static int kmonitor_remove(pid_t target, int type)
+{
+ struct list_head *tmp, *next;
+ int ret = -EINVAL;
+ int bucket = khash(target);
+
+ TRACE("%i, %i", target, type);
+
+ write_lock(&kmonitor_resources[bucket].lock);
+ list_for_each_safe(tmp, next, &kmonitor_resources[bucket].list) {
+ struct kmonitor_res *r = to_kmonitor_res(tmp);
```

Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+ if (r->monitor == current->pid && r->target == target) {
+ r->event_mask &= ~type;
+ if (!r->event_mask) {
+ /* No more events so go ahead and cleanup */
+ list_del(tmp);
+ kmem_cache_free(kmonitor_cache, r);
+ }
+ ret = 0;
+ break;
+ } + }
+
+ write_unlock(&kmonitor_resources[bucket].lock);
+ return ret;
+}
+
+void __kmonitor_notify_event(struct task_struct *tsk, int type) +{
+ struct siginfo info;
+ struct list_head *tmp, *next;
+ int bucket = khash(current->pid);
+
+ read_lock(&kmonitor_resources[bucket].lock);
+ list_for_each_safe(tmp,next,&kmonitor_resources[bucket].list) {
+ struct kmonitor_res *r = to_kmonitor_res(tmp);
+ if (r->target == current->pid && type & r->event_mask) {
+ info.si_signo = r->signal;
+ info.si_code = SI_QUEUE;
+ info.si_int = type;
+ info.si_pid = tsk->pid;
+ info.si_uid = tsk->uid;
+ kill_proc_info(r->signal, &info, r->monitor);
+ }
+ }
+ read_unlock(&kmonitor_resources[bucket].lock);
+}
+
+static int kmonitor_open(struct inode *inode, struct file *file)
+{
+ atomic_inc(&kmonitor_active);
+ return 0;
+}
+
+static int kmonitor_release(struct inode *inode, struct file *file)
+{
+ struct list_head *tmp, *next;
+ int i;
+
+ TRACE("%p, %p", inode, file);
+ atomic_dec(&kmonitor_active);
+
+ /*
+ * Remove any resources associated with this monitor
```

Linux–Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+ */
+ for (i=0; i<(1<<<KMONITOR_BITS); i++) {
+ write_lock(&kmonitor_resources[i].lock);
+ list_for_each_safe(tmp, next, &kmonitor_resources[i].list) {
+ struct kmonitor_res *r = to_kmonitor_res(tmp);
+ if (r->monitor == current->pid) {
+ list_del(tmp);
+ kmem_cache_free(kmonitor_cache, r);
+ } + }
+ write_unlock(&kmonitor_resources[i].lock);
+ }
+
+ return 0;
+}
+
+static int kmonitor_ioctl(struct inode *inode, struct file *file,
+ unsigned int cmd, unsigned long arg)
+{
+ struct kmonitor_cmd hdr;
+
+ TRACE("%p, %p, %i, %i", inode, file, (int)cmd, (int)arg);
+ if (copy_from_user((struct kmonitor_cmd *)&hdr, +
+ (struct kmonitor_cmd *)arg, + sizeof(struct kmonitor_cmd)))
+ return -EFAULT;
+
+ switch (cmd) {
+ case KMONITOR_IOW_REGISTER:
+ return kmonitor_add(hdr.pid, hdr.type, hdr.signal);
+ case KMONITOR_IOW_DEREGISTER:
+ return kmonitor_remove(hdr.pid, hdr.type);
+ }
+
+ DBG("Unexpected ioctl, %i", cmd);
+ return -EINVAL;
+}
+
+static struct file_operations kmonitor_fops = {
+ .owner = THIS_MODULE,
+ .llseek = no_llseek,
+ .open = kmonitor_open,
+ .release = kmonitor_release,
+ .ioctl = kmonitor_ioctl,
+};
+
+static struct miscdevice kmonitor_miscdev = {
+ .minor = KMONITOR_MINOR,
+ .name = "kmonitor",
+ .fops = &kmonitor_fops,
+};
+
+static int __init kmonitor_init(void)
```

```

+{
+ int ret, i;
+
+ TRACE("void");
+
+ for (i=0; i<(1<<KMONITOR_BITS); i++) {
+ INIT_LIST_HEAD(&kmonitor_resources[i].list);
+ kmonitor_resources[i].lock = RW_LOCK_UNLOCKED;
+ }
+
+ ret = misc_register(&kmonitor_miscdev);
+ if (ret) {
+ printk(KERN_ERR "unable to register kmonitor misc device\n");
+ goto error_misc_drvr_register;
+ }
+
+ kmonitor_cache = kmem_cache_create("kmonitor_res",
+ sizeof(struct kmonitor_res), 0,
+ SLAB_HWCACHE_ALIGN,
+ NULL, NULL);
+ if (!kmonitor_cache) {
+ printk(KERN_ERR "unable to allocate kmonitor_cache\n");
+ ret = -EINVAL;
+ goto error_in_kmonitor_cache;
+ }
+
+ return 0;
+
+ error_in_kmonitor_cache:
+ misc_deregister(&kmonitor_miscdev);
+ error_misc_drvr_register:
+ return ret;
+}
+
+static void __exit kmonitor_exit(void)
+{
+ struct list_head *tmp, *next;
+ int i;
+
+ TRACE("void");
+ misc_deregister(&kmonitor_miscdev);
+ for (i=0; i<(1<<KMONITOR_BITS); i++) {
+ write_lock(&kmonitor_resources[i].lock);
+ list_for_each_safe(tmp, next, &kmonitor_resources[i].list) {
+ struct kmonitor_res *r = to_kmonitor_res(tmp);
+ list_del(tmp);
+ kmem_cache_free(kmonitor_cache, r);
+ }
+ write_unlock(&kmonitor_resources[i].lock);
+ }
+ kmem_cache_destroy(kmonitor_cache);

```

Linux-Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+}
+
+module_init(kmonitor_init);
+module_exit(kmonitor_exit);
+
+MODULE_AUTHOR("Atul Sabharwal");
+MODULE_DESCRIPTION("Kmonitor Char Driver");
+MODULE_LICENSE("GPL");
+MODULE_ALIAS_MISCDEV(KMONITOR_MINOR);
diff -Nru a/fs/exec.c b/fs/exec.c
--- a/fs/exec.c 2004-06-14 11:24:08 -07:00
+++ b/fs/exec.c 2004-06-14 11:24:08 -07:00
@@ -46,6 +46,7 @@
#include <linux/security.h>
#include <linux/syscalls.h>
#include <linux/rmap.h>
+#include <linux/kmonitor.h>

#include <asm/uaccess.h>
#include <asm/pgalloc.h>
@@ -1144,6 +1145,7 @@
    free_arg_pages(&bprm);

    /* execve success */
+ kmonitor_notify_event(current, KMONITOR_PROCESS_EXEC);
    security_bprm_free(&bprm);
    return retval;
}
diff -Nru a/include/linux/kmonitor.h b/include/linux/kmonitor.h
--- /dev/null Wed Dec 31 16:00:00 196900
+++ b/include/linux/kmonitor.h 2004-06-14 11:24:08 -07:00
@@ -0,0 +1,62 @@
+/* + *
+ * linux/kmonitor.h
+ *
+ * Copyright (C) 2004 Intel Corporation. All rights reserved.
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public
+ * License v2.0 as published by the Free Software Foundation; + *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
+ * General Public License for more details.
+ *
+ * You should have received a copy of the GNU General Public
+ * License along with this program; if not, write to the
+ * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
+ * Boston, MA 02110-1307, USA.
+ *
+ * Authors: Atul Sabharwal <atul.sabharwal@intel.com>
```

Linux-Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+ * + */
+#ifndef _LINUX_KMONITOR_H_
+#define _LINUX_KMONITOR_H_
+
+#include <asm/atomic.h>
+
+#define KMONITOR_THREAD_CREATE 0x01 +#define KMONITOR_THREAD_ABORT
0x02 +#define KMONITOR_THREAD_EXIT 0x04
+#define KMONITOR_PROCESS_FORK 0x08
+#define KMONITOR_PROCESS_ABORT 0x10 +#define KMONITOR_PROCESS_EXEC
0x20 +#define KMONITOR_PROCESS_EXIT 0x40
+
+struct kmonitor_cmd +{
+ pid_t pid;
+ int type;
+ int signal;
+};
+
+#define KMONITOR_IOW_REGISTER_IOW(0xDE, 1, struct kmonitor_cmd)
+#define KMONITOR_IOW_DEREGISTER_IOW(0xDE, 2, struct kmonitor_cmd)
+
+#ifdef __KERNEL__
+#include <linux/sysctl.h>
+
+extern atomic_t kmonitor_active;
+extern void __kmonitor_notify_event(struct task_struct *tsk, int type);
+
+static inline void kmonitor_notify_event( struct task_struct *tsk, int
type)
+{
+#ifdef CONFIG_KMONITOR
+ if (atomic_read(&kmonitor_active))
+ __kmonitor_notify_event(tsk, type);
+#endif /* CONFIG_KMONITOR */
+}
+#endif /* __KERNEL__ */
+#endif /* _LINUX_KMONITOR_H_ */
+
diff -Nru a/include/linux/miscdevice.h b/include/linux/miscdevice.h
--- a/include/linux/miscdevice.h 2004-06-14 11:24:08 -07:00
+++ b/include/linux/miscdevice.h 2004-06-14 11:24:08 -07:00
@@ -35,6 +35,7 @@
#define SGI_USEMACLONE 151

#define TUN_MINOR 200
+#define KMONITOR_MINOR 221

struct device;

diff -Nru a/kernel/exit.c b/kernel/exit.c
--- a/kernel/exit.c 2004-06-14 11:24:08 -07:00
```

Linux-Kernel: [Announce] Non Invasive Kernel Monitor for threads/processes

```
+++ b/kernel/exit.c 2004-06-14 11:24:08 -07:00
@@ -22,6 +22,7 @@
#include <linux/profile.h>
#include <linux/mount.h>
#include <linux/proc_fs.h>
+#include <linux/kmonitor.h>

#include <asm/uaccess.h>
#include <asm/pgtable.h>
@@ -748,6 +749,11 @@
    */
    _raw_write_unlock(&tasklist_lock);
    local_irq_enable();
+
+ if (tsk->pid == tsk->tgid)
+ kmonitor_notify_event(tsk, KMONITOR_PROCESS_EXIT);
+ else
+ kmonitor_notify_event(tsk, KMONITOR_THREAD_EXIT);

    /* If the process is dead, release it - nobody will wait for it */
    if (state == TASK_DEAD)
diff -Nru a/kernel/fork.c b/kernel/fork.c
--- a/kernel/fork.c 2004-06-14 11:24:08 -07:00
+++ b/kernel/fork.c 2004-06-14 11:24:08 -07:00
@@ -41,6 +41,7 @@
#include <asm/mmu_context.h>
#include <asm/cacheflush.h>
#include <asm/tlbflush.h>
+#include <linux/kmonitor.h>

/* The idle threads do not count..
 * Protected by write_lock_irq(&tasklist_lock)
@@ -1167,6 +1168,11 @@

    if (!IS_ERR(p)) {
        struct completion vfork;
+
+ if (p->pid == p->tgid)
+ kmonitor_notify_event(p, KMONITOR_PROCESS_FORK);
+ else
+ kmonitor_notify_event(p, KMONITOR_THREAD_CREATE);

        if (clone_flags & CLONE_VFORK) {
            p->vfork_done = &vfork;
-
- To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
- the body of a message to majordomo@vger.kernel.org
- More majordomo info at http://vger.kernel.org/majordomo-info.html
- Please read the FAQ at http://www.tux.org/lkml/
```