

# Generating a White List for Hardware which Works with Kexec/Kdump

Fernando Luis Vázquez Cao  
*NTT Open Source Software Center*  
fernando@oss.ntt.co.jp

## Abstract

The mainstream Linux kernel lacked a crash dumping mechanism from its inception until the recent adoption of kdump. This, despite the fact that there were several solutions available out-of-tree and some of them were even included in major distributions. However concerns about their intrusiveness and reliability prevented them from making it into the mainstream kernel (aka vanilla kernel), the main argument being that relying on the resources of a crashing kernel to capture a dump, as they did, is inherently dangerous.

The appearance of Eric Biederman's kexec patches and its subsequent inclusion in the kernel as a new system call paved the way for the implementation of an idea that had been floating around for some time: the use of a memory-preserving soft-booted kernel to capture the crash dump. This was the approach adopted by kdump, which made it possible to achieve high reliability by isolating the crash dumping process from the crashed kernel.

In theory, kdump's approach constitutes the most reliable way of capturing a dump. Even though testing proved the theory right (i.e. kdump is much more robust and reliable than in-kernel crash dumping solutions), some deficiencies in kdump were revealed too.

Kernel crash dumping is a multi-stage process which involves three basic operations: detecting the crash, a minimal shutdown of the previously running system (i.e. the crashed kernel), and, fi-

nally, the capture of the crash dump. Kdump is very good at the first two but there are still some issues when the dump capture kernel takes control of the system. In particular the new kernel may fail to initialise the underlying devices which, in turn, is likely to lead to a kernel panic or an oops.

The underlying problem is that the state of the devices during a kdump boot is not predictable because no device shutdown is performed in the crashed kernel (it cannot be trusted), and the firmware stage of the standard boot process is skipped (the dump capture kernel is a soft-booted kernel after all). In other words, the inherent assumption that the firmware (known as the BIOS on some systems) is always there to do the dirty work is not valid anymore.

The Linux Kernel in general and device drivers in particular need to be improved so that they are able to boot in potentially unreliably environments, which with the advent of soft-reboot mechanisms such as kexec is likely to become a common scenario. But this is bound to be a painstaking and never-ending task, which requires the creation of a white-list that is updated as bugs are fixed and new hardware appears. This paper discusses possible ways of fixing the aforementioned reliability problems and an automated testing method that can be used to create a white list for hardware that works with kdump.

# 1 Background

Even though there was no kernel crash dumping mechanism in the mainstream Linux kernel, distributors and adventurous users had a plethora of implementations to choose from: LKCD [1], Diskdump [2], Netdump [3] to cite some. This abundance of solutions and the variety of their approaches is a reflection of the technical difficulty of the problem (see section 2 for details).

All the aforementioned implementations were in-kernel crash dumping mechanisms. In other words, to capture a kernel dump they relied on resources and drivers provided by the crashing kernel, which is at the root of their reliability issues and complexity.

The difficulty of setting up a controllable dump route within the kernel sparked the apparition of crash dumping mechanisms capable of capturing a dump from a new context, independently from the crashed kernel. This new context is a soft-booted new kernel that takes control of the system upon a system crash event without clearing the crashed kernel's memory.

Mission Critical Dump [4] using the soft-boot software *bootimg* [5] provided a proof of concept. Later, kdump [6] also followed this design principle [7] and due to its clean design and integration with kexec was adopted as the Linux kernel crash dumping mechanism.

At the same time, device drivers remain a key issue which is crucial to Gnu/Linux being embraced in the enterprise world. They must be open-source, maintained, and fully featured. Additionally, they must be shown to work in all situations. Unfortunately, one area which has not been examined especially closely, is whether a driver comes up properly from a boot initiated by *kexec*.

The Data Center Linux work group at OSDL (Open Source Development Labs)<sup>1</sup> raised the issue that kexec is still not completely reliable. One

---

<sup>1</sup>OSDL would eventually merge with the FSG (Free Standards Group) to create the Linux Foundation

reason was that devices booted through kexec do not go through the normal initialisation process. Some drivers handle this well, but others do not yet and perhaps are not aware that it is a requirement. It was suggested that a *white list* of drivers that do *behave* under these circumstances would be helpful. Initially it was thought that a *black list* of drivers which did not work might be helpful to put some pressure on driver maintainers to maintain this functionality, but this approach was discarded as inappropriate.

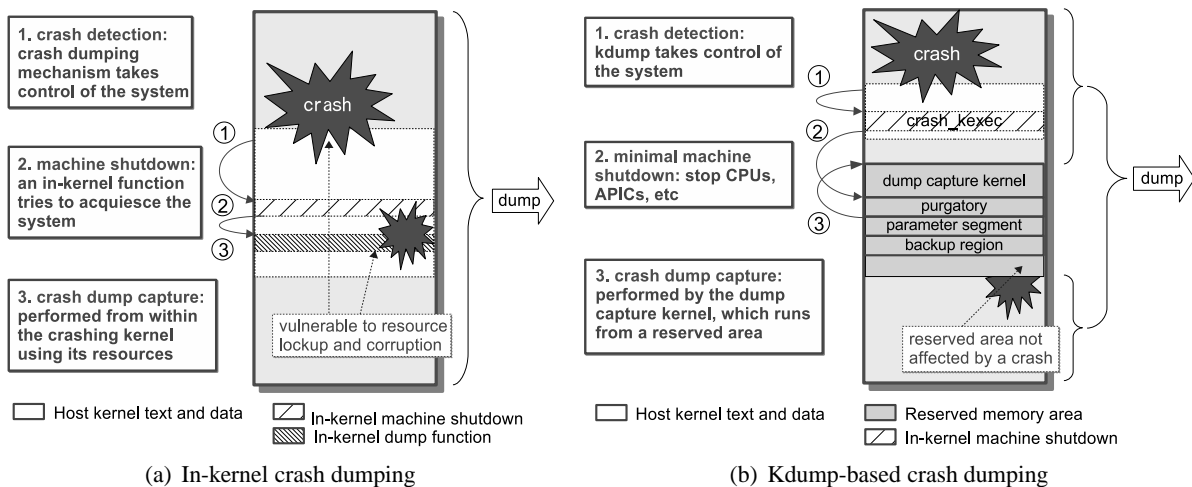
This paper describes the process and results of putting together such a white list. But since some knowledge about kernel crash dumping in general and kdump in particular is needed, a brief introduction comes first.

## 2 Traditional crash dumping mechanisms

The primary cause of the bad results obtained with in-kernel crash dumping mechanisms is the flawed assumption that the kernel can be trusted and will in fact be operating in a normal fashion. This creates two major problems (represented in figure 1(a)).

First, there is a problem with resources, notably with resources locking up (*resource lock-up*), because it is not possible to know the locking status at the time of the crash. These crash dumping mechanisms use drivers and services of the crashing kernel to capture the dump. As a consequence, if the operation that has caused the crash was locking resources necessary to capture the dump, the dump operation will end up deadlocking.

For example, the driver for the dump device may try to obtain a lock that was held before the crash occurred and, as it will never be released, the dump operation will hang up. Similarly, on SMP systems as operations being run on other CPUs are forced to stop in the event of a crash, there is the possibility that resources needed during the dump process may be locked, because they were in use by any of



**Figure 1:** The difficulty of setting up a controllable dump route within the kernel sparked the apparition of crash dumping mechanisms capable of capturing a dump from a new clean context.

the other CPUs and were not released before they halted. This may put the dump operation into a lockup too. Even if this does not result in a lockup, insufficient system resources may also cause the dump operation to fail.

The source of the second problem is the reliability of the control tables (and kernel data in general), kernel text, and drivers (*resource corruption*). A kernel crash means that some kind of inconsistency has occurred within the kernel and that there is a strong possibility a control structure has been damaged. As in-kernel crash dump mechanisms employ functions of the crashed system for outputting the dump, there is the very real possibility that the damaged control structures will be referenced. Besides, page tables and CPU registers such as the stack pointer may be corrupted too, which can potentially lead to faults during the crash dumping process. In such circumstances, even if a crash dump is finally obtained, the resulting dump image is likely to be corrupted.

For in-kernel crash dumping mechanisms there is no obvious solution to the resource corruption problems. However, the locking issues may be al-

leviated by using polling mode (as opposed to interrupt mode) to communicate with the dump devices.

### 3 The need for a new approach

Setting up a controllable dump route within the kernel is extremely difficult and this is increasingly true as the size and complexity of the kernel augments. Awareness of this issue set off the development of crash dumping mechanisms capable of capturing a dump from a new context, independently from the crashed kernel. As mentioned before, the basic idea is to soft-boot a dump capture kernel without clearing the crashed kernel's memory (see figure 1(b)).

This idea had been tossed around for some time and, in fact, was embraced by several groups interested in crash dumping and, as is common in the open source world, each group came up with its own implementation. Eventually, kdump's cleaner design earned it the endorsement of the community, and development converged around kdump.

From its design it would seem that kdump is not

vulnerable to the two issues that plague in-kernel crash dump solutions: *resource lock-up* and *resource corruption*.

The *resource lock-up* issues are avoided because kdump uses the resources (memory, drivers, etc) of the *newly booted* kernel to capture the dump. Regarding *resource corruption*, it could be argued that even if the host kernel is careful not to touch the memory region in which the dump capture kernel has been loaded, a kernel bug or a DMA transaction gone wild could end up overwriting it. To alleviate this problem, the host kernel reserves a memory region for crash dump purposes and leaves it out of the kernel's memory map, as figure 1(b) illustrates. This reserved memory area is then used to load the dump capture kernel and to pass information between the two kernels (see section 6 for details). In addition, to be more robust against DMA transactions originated from the context of the crashed kernel, the dump capture kernel is run directly from the reserved memory area.

Before delving into the specifics of kdump some knowledge of kexec and the boot process in Linux is essential.

## 4 The boot process in Linux

As figure 2 shows everything begins when the hardware is powered on (*hardware stage*). After some initialisation the firmware takes control of the system (*firmware stage*). The firmware, also known as BIOS in some architectures, detects and pre-configures the various devices in the system, including memory controllers, storage devices, bus bridges and other devices. Then, based on the settings, control goes to a minimal boot-loader known as the master boot record (MBR). The MBR usually resides on a local disk drive, but, if the firmware supports it, could also be retrieved from removable media or over the network. The actual job of transferring control to the operative system is performed by the second-stage boot-loader (in figure 2 the first and second stage boot

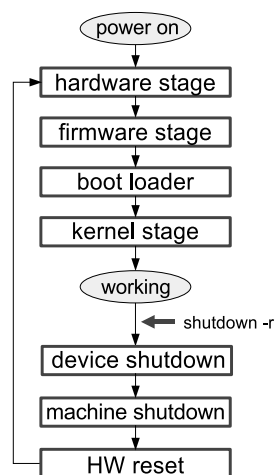


Figure 2: Linux Boot Sequence

loaders were collapsed into a single *boot loader stage* for simplicity). The boot-loader commonly allows the user to choose the kernel to be loaded as well as its command line and, sometimes, the initial ram-disk (*initrd*) can be indicated too. In accordance to the selections, it loads the kernel, *initrd* (when necessary), and related parameters onto memory, sets up the necessary environment and, finally, hands over control to the kernel, thus entering the *kernel stage* of the boot process.

The kernel sets up the necessary data structures, probes the devices present on the system, loads the necessary drivers, and initialises the devices. The last stage of the boot process involves user-level initialisation, but since this is not directly relevant for kdump it is out of the scope of this article.

Regarding system reboots, the kernel should perform a clean shutdown of the currently running system, so that no data is lost and the underlying devices are in usable state after the reboot. Among other things, this involves terminating running processes, writing back cache buffers to disk, unmounting file systems, and, most importantly for this discussion, a hardware reset.

## 5 Kexec

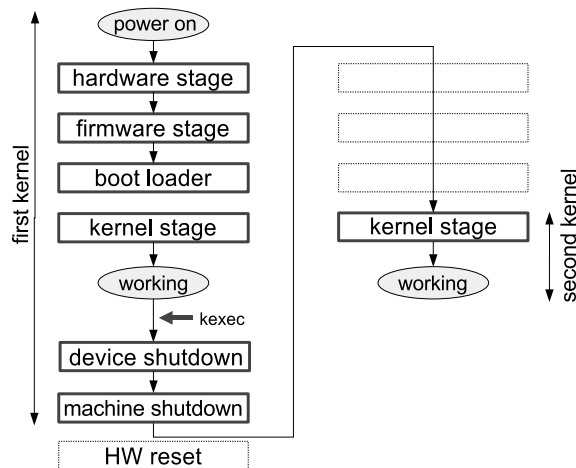


Figure 3: Kexec Boot Sequence

A kexec boot differs greatly from a traditional boot or reboot through the firmware or BIOS. To begin with, during a kexec reboot all the firmware stages present during a cold boot are skipped (this is what soft-booting is all about, after all). This fact is stressed in figure 3. Besides, whereas during a standard reboot through the firmware a hardware reset of the devices is performed, kexec transfers control to the new kernel right after the device shutdown stage, during which neither the underlying devices nor the buses are reset.

Ideally, invoking the device shutdown methods should suffice to quiesce the devices, but this is not always the case. The main reason being that at the time most drivers were developed kexec and its ilk did not even exist, so the device shutdown path has not been properly tested. The implications of this situation will be discussed in later sections.

In this paper the initial boot kernel will be referred to as the first kernel and the kernel that comes up via kexec will be the second kernel.

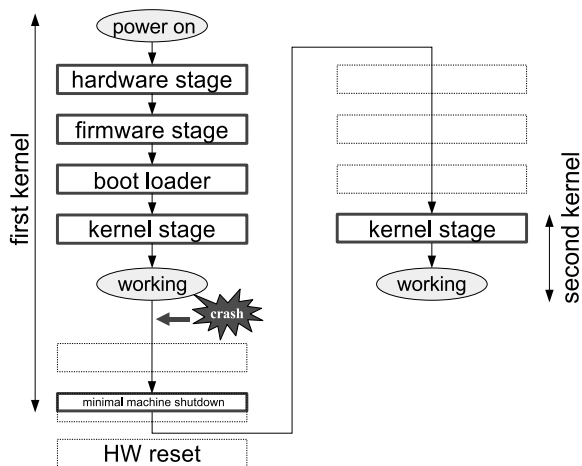


Figure 4: Kdump Boot Sequence

## 6 Kdump

As suggested before, kdump could be considered to be just a special case of kexec. The differences between both arise from the fact that kdump has to deal with a potentially unreliable execution environment: a kernel crash. Kexec, by using the shutdown methods that each driver provides to acquiesce the underlying devices, is implicitly assuming that the current kernel can be relied upon. But this obviously does not hold true in the event of a crash.

Because of this, in kdump's case the machine shutdown stage has to be kept minimal (this is what the slim box in figure 4 attempts to represent). After a crash the state of the kernel is unpredictable: the various kernel stacks could have overflowed, spilling into whatever is adjacent; the CPU registers may hold invalid values; and the kernel text and data may have been corrupted.

One should keep this in mind and avoid using those kernel resources in the reboot path to the second kernel. In other words, all the things that can be deferred to the second kernel should be done there, since, as opposed to the crashed kernel, the dump capture kernel runs on a clean execution environment. Thus, the crashing kernel after stop-

ping CPUs, disabling interrupts, and performing some minor janitorial work transfers control to the dump capture kernel.

Even though crash dumping is all about being paranoid, there is a limit to the things we can do or, better, avoid doing. The small piece of code executed before jumping into the capture kernel is still a *part of the kernel* and, as such, subject to its design principles.

Once we hand over control to the capture kernel it could be said that we have reached a safe haven. As figure 1(b) shows, the capture kernel runs directly from the reserved memory area it was loaded into. The execution context of the new kernel is restricted to this area, which the host kernel protects from accidental overwriting by leaving it out of its memory map. In theory, this ensures that no memory will be allocated from that area for any purpose.

Active DMA transactions during a kexec boot can have catastrophic consequences, because the new kernel could get stomped by one of them. Kexec addresses this issue by stopping all ongoing DMA during the device shutdown stage. But as discussed before, kdump cannot afford that luxury because the crashing kernel, including the device drivers cannot be trusted. Thus a different approach was taken. Kdump solves the problem by booting the kernel directly from the reserved memory area it was loaded into. The basic assumption is that the first kernel will never assign memory from the reserved memory area for DMA<sup>2</sup>.

## 6.1 Initialising devices after a software re-boot (device reinitialisation)

Kdump poses a big challenge for device drivers. In the event of a crash, kdump does not perform

---

<sup>2</sup>To be precise, this does not necessarily hold true in the event of a system crash, which is precisely the case we are trying to address. For example, it is impossible to guarantee that a kernel bug will never cause the kernel to map memory from that area. Similarly, in systems that have one, a bad assignment of IOMMU entries could bring havoc.

any kind of device shutdown (it is not a safe thing to do after a crash) and, to make things worse, the firmware stages of the standard boot process are skipped too (kdump is just a special case of kexec after all).

To put it simple, after a minimal machine shutdown, kdump just jumps into the dump capture kernel. Because of this, the underlying devices might be in an unstable state the new kernel cannot get them out of.

Another possible scenario is that a device remains operational and keeps sending interrupts and messages generated in the context of the previous kernel (i.e. the crashed kernel), and about which the newly booted kernel does not know anything about. Some drivers do not contemplate the possibility of pending or spurious interrupts and show incorrect behaviour that may lead to a crash of the dump capture kernel itself. Other drivers consider these to be anomalous situations and, accordingly, would raise an oops or even a panic.

At the root of this problem is the flawed assumption that the firmware is always there to do the dirty job of resetting and preparing the devices. But this does not hold true after a soft-reboot. Device drivers and related subsystems should take into account the kdump case, and try to quiesce the underlying devices and restore them to a good known state when necessary. Possible ways of ways of doing this are discussed later in this section.

It should be noted that this problem is not exclusive to kdump. It also occurs when soft-booting into Linux and device shutdown has not been performed properly by the previous kernel. Sometimes even after a regular hardware reboot drivers find devices in a state they cannot handle and panic. The culprit in such cases is buggy firmware not doing its job properly.

Finally, sometimes the reason a device driver fails to initialise the underlying device is not related to the device itself but the system firmware or BIOS, which is supposed to provide some data, such as configuration information or a binary blob (firmware for a SCSI controller for example), that

is no longer there. This is relatively common in x86 architectures where sometimes the information or data residing in the BIOS is not accessible after the first read (which commonly occurs in the first kernel).

Several approaches have been proposed to tackle the device reinitialisation issue:

- Creation of a black list/white list of devices.
- Device soft-reset.
- Device configuration save/restore.
- Driver hardening.

These methods far from being mutually exclusive could be combined to achieve better results. Each of them will be discussed individually.

### 6.1.1 Device black list

A possible way to address the device reinitialisation problems is to avoid using potentially problematic drivers. For that we would need to create a list of drivers that are known to have problems and use that list to mark the drivers, either at compile or run-time, so that they are not used in kexec or kdump booted kernels.

Even though this measure could encourage device driver maintainers to fix their drivers, some people may consider this approach politically incorrect. As an alternative, a white list or grey list could be used instead.

### 6.1.2 Device reset/quiescing

Perform a soft-reset of each device, so that any pending interrupts are discarded and the device returns to a good know state. Once the device is reset the kernel could proceed safely with the rest of the initialisation. The soft reset can be carried in two major ways:

- Per-device basis.

- Bus reset.

This approach has some drawbacks, though. Firstly, soft-reset is a functionality that is not supported by all devices and buses, so it does not constitute a generic solution on its own. Secondly, even in devices that support it the soft-reset might be a time-consuming operation (especially in the case of SCSI controllers), thus it is not an option when there are time restrictions.

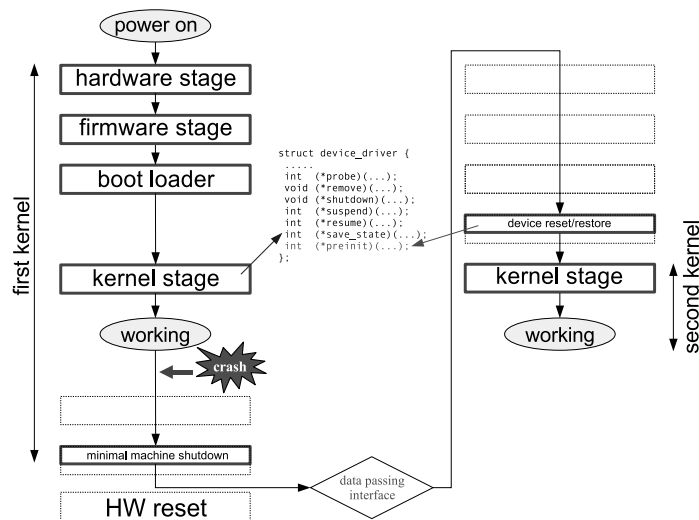
The problem of pending interrupts coming in could be alleviated by taking advantage of special hardware features, such as the interrupt enable/disable functionality added in PCI 2.3. But again the kernel should not rely too much in such functionality because it is not present in all systems.

### 6.1.3 Device configuration save/restore

Try to put the devices into a good known state before proceeding with the standard initialisation. When it is not possible to know what the right configuration is (due to lack of specs for example) the following technique can be used (figure 5 depicts a possible implementation):

- *Save the configuration as performed by the firmware:* After a normal boot through the firmware save the configuration of all devices before the kernel modifies the default configuration.
- In the event of a crash *pass this information to the second kernel.* Kexec already provides the basic infrastructure to do this [7].
- *Use this information to pre-configure devices.* This simulates the work done by the firmware during a hardware reboot.
- Proceed with the rest of the initialisation.

It is quite obvious that this shows many similarities to and was inspired by suspend/resume so it should be possible to reuse part of the code or, at least, get some ideas from it.



**Figure 5:** Device configuration restore

Regarding this approach the development community seems to be divided. Some people oppose this technique because they consider it to be overkill and that drivers should be fixed instead. But, on the other hand, this approach provides a clean and generic solution that has potential synergies with suspend/resume. Not to mention that it would provide an easy way to deal with buggy BIOSes.

#### 6.1.4 Driver hardening

Instead of creating new funky infrastructures focus on actually fixing the drivers as bugs are found, so that they are able to initialise the underlying devices even when they are in an unstable state.

This is considered by many to be the way to go, but due to the big number and variety of drivers in Linux the involvement of device driver maintainers is indispensable.

## 6.2 How to proceed

In the previous section several solutions were proposed. It was also suggested that they could be combined, which is very important considering

that not all the mechanisms are applicable in all cases.

But the problem of how to start tackling the device reinitialisation issue is also important, because adding new infrastructure to such a brittle part of the kernel is not acceptable unless a specific case is made for it to get in. Some consensus has been reached to proceed as indicated below:

- Start by creating either a list of devices known to have problems, that is a black list, or its opposite, a white list (see section 6.1.1).
- Fix bugs in drivers as they are found (see section 6.1.4).
  - Speed up the process through testing (see section 7).
- In some cases just fixing the driver may not be enough to solve the problem.
  - Reconsider the device reset 6.1.2 and the device configuration save/restore 6.1.3 approaches if such a case is found.

## 7 Testing

Crash dump mechanisms are special in the sense that if the kernel were bug free they would never be used. But if for whatever reason something goes wrong and the kernel crashes the crash dumping mechanism is the kernel developer's last (and ultimate) resort to try to figure out what went wrong. Besides, experience shows that certain kernel bugs are very difficult to reproduce, so failure to obtain a dump can be very costly in terms of time. Thus, reliability should be a primary concern and, accordingly, crash dumping mechanisms should be designed from the premise that you only get one shot at it.

While kexec is the generally accepted solution to debugging panicking machines, the reliability of it across a wide variety of hardware has not been investigated. When kexec boots the second kernel it bypasses the normal start up processes which normally initialise the devices through the BIOS (on x86\_32/64). The question becomes, will the hardware and the drivers behave well under these circumstances?

Kernel crash dumping is a multi-stage process which involves three basic operations: detecting the crash, a minimal shutdown of the previously running system (i.e. the crashed kernel), and, finally, the capture of the crash dump. Kdump is already pretty good at the first two but there are still some issues when the dump capture kernel takes control of the system. In particular the new kernel may fail to initialise the underlying devices which, in turn, is likely to lead to a kernel panic or an oops. This paper will focus on the latter, since the first two stages have already been analysed in depth in other articles [8].

Considering that the goal is finding problems that arise *after* being kdump-booted, one could think that the only way to do meaningful testing is to cause a crash in the first kernel so that the dump capture kernel takes control of the system. This is a legitimate approach and usually gives good results (see section 7.1 for details on a test suite that

follows this approach). However, this method offers poor control over the testing process and, as a consequence, certain crash scenarios may be difficult to reproduce. As discussed later, there are techniques that alleviate this problem and make it easier to obtain reproducible results.

### 7.1 LKDTT-based testing

The Linux Kernel Dump Test Tool (LKDTT)[8] is a kit that makes it very easy to cause various types of kernel failures at a few pre-coded locations. Besides, failures at custom locations can be easily added. The location where a failure is injected is referred to as Crash Point (CP), and it determines the execution context at the time of the crash.

CPs have two attributes that are configurable. The first is the *crash type* which determines the type of crash scenario to be recreated. There is panic, oops, exception, stack overflow, and lockup to choose from. The other attribute is the *counter* which indicates the number of times the crash point has to be crossed before the kernel is forced to crash. These attributes can be changed on a per-crash point basis using a user-land tool (ttutils).

The criteria for labelling a driver as *good* is important to set at the outset. When running tests it is important to understand whether the test is successful in its purpose. The crash points must be tripped appropriately. Some are a bit difficult to trip (LKDTT provides several helper applications to help with this) and it was necessary to repeat the test and be certain that it tripped appropriately. If a an oops occurs, but no panic and kdump does not load, this is a failure of the fist kernel rather than of the driver. These are legitimate bugs but they are not what this series of tests was looking at. If the panic happens and the kexec begins to load the second kernel and then it panics before it is up, this can be indicator of a kernel problem or a driver problem. These messages should be examined and reported to the correct mailing lists for kexec or for the driver.

The LKDTT testing environment consists of the following elements:

- *LKDTT kernel*: The machine will be installed with the desired first kernel which has the correct configuration parameters and the LKDTT patches applied and built as a module. Provided are the configuration parameters to be added.

It is important to get the first kernel settings correct. For the compile required options are `CONFIG_KEXEC=y`, `CONFIG_DETECT_SOFTLOCKUP=y`, `CONFIG_DEBUG_STACKOVERFLOW=y`, `CONFIG_MAGIC_SYSRQ=y`, `CONFIG_DEBUG_KERNEL=y`. Boot options include `nmi_watchdog=1`. Set `/proc/sys/kernel/panic_on_oops` to one in order to help trigger a panic.

- *Test script*: Included are the init scripts (written in bash) which are installed at the outset of the test.

- First kernel (LKDTT kernel) side: *kdump-test* starts the test on the first kernel. The test script was written to be certain that the system was manipulated so that the crash points were tripped (LKDTT's auxiliary tools were used for this). As an example, for the `MEM_SWAPOUT` crash point (located in the code that implements paging) the tool *brk* from the `dtc_tools` was used to use enough memory to start swapping. There is an option to start some disk IO to and from the disk being tested, so that a crash point inserted in the SCSI code is crossed.

- Dump capture kernel side: *kdump-fetch* starts it for the second kernel. This script should be included in the customised `initrd` discussed below.

- *Customised initrd*: Added options were

`dump-dev`, `dump-fs`, and `firmware`. `dump-dev` and `dump-fs` specified the device of interest and its file system respectively. `firmware` causes the contents of `/lib/firmware` to be copied into the `initrd` image.

If the CP method fails the test script uses the `sysrq` mechanism to reset the kernel. If this too fails manual `sysrq` is attempted.

These tests are difficult to automate entirely due to several issues. The first is that when there is a software lockup it will just loop forever. It would be nice to have a kernel option to panic on software lockup. Secondly, `sysrq-c` seems to be very unreliable through the serial console. It is supposed to work, but 95 percent of the time did not, while it would work fine at the actual console. Thirdly, the failures being sought are often in the form of a panic, which leaves the machine hanging. This could be addressed by adding a timer on a remote machine, but this feature has not been implemented yet.

There are a few cautions regarding this sort of testing. Disks do not always come up in the expected order, as it is possible that the driver detects too many or too few disks. Therefore be certain that you do not value any data on the array you are working on when writing to the drives and watch for the correct number of disks during initialisation.

### 7.1.1 Test Results

Table 1 shows the IO adaptors that were tested. The Linux distributions used for this testing were Fedora Core 5 and Fedora Core 6. However, Fedora Core 5's `mkinitrd` scripts were customised to automatize the testing process, and since they also worked on Fedora Core 6 they were reused there.

The panics initiated are the full series available through LKDTT. There are eight crash points (CPs) and for each CP there are five methods of crashing to choose from: bug, exception, loop, overflow and panic. When a particular crash point fails to initiate

Driver	Device	Kernel version	Result
qla2xxx	QLogic Corp. QLA2422 Fibre Channel	2.6.21-rc1	Success
lpfc	Emulex Corporation Helios LightPulse Fibre Channel Host Adapter (rev 01)	2.6.21-rc1	Success
lpfc	Emulex Corporation Zephyr LightPulse Fibre Channel Host Adapter (rev 02)	2.6.21-rc1	Success <sup>a</sup>
megaraid	Dell PowerEdge Expandable RAID controller (SCSI)	2.6.21-rc1	Reported bug
mptsas	LSI Logic / Symbios Logic SAS1068E PCI-Express Fusion	2.6.21-rc1	Reported Bug
mptfc	LSI Logic / Symbios Logic FC929X Fibre Channel Adapter (rev 81)	2.6.21-rc1	Success
mptspi	LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320 SCSI (rev 08)	2.6.21-rc1	Success
aic94xxi	Adaptec AIC-9410W SAS (Razor HBA RAID) (rev 09)	2.6.20-rc7-scsi-misc	Success <sup>b</sup>
aacraid	AACRAID/SAS	2.6.21-rc3-scsi-misc	Panics Intermittently

<sup>a</sup>But needed pause between load driver and mount.

<sup>b</sup>But requires firmware in initrd.

**Table 1:** Summary of Drivers.

a panic, then `sysrq-c` is used to initiate one (this is done automatically by the test script).

One card, Adaptec AIC-9410W SAS, required firmware at load time. Normally this happens through `udev`, which is not up in the `initrd`. An option (`-w`) was added which uses the program `firmware_helper` in the `initrd`, statically compiled and accessed from `/proc/sys/kernel/hotplug`.

Hardware used were all `x86_64`, both AMD and Intel with 4G of memory. The equipment was donated by Intel and AMD to the former OSDL. The IO cards included the internal Megaraid and the internal IDE/ATA. Additionally, tested are Fusion SAS, Fusion FC, Fusion PSCSI, Aacraid SAS, Aacraid PSCSI, Emulex, QLA2432 FC, QLA2422 FC, QLA2312 FC. These were also donated by LSI, Adaptec, Emulex and Qlogic respectively.

Drivers for the IO adaptors above were tested for whether they come up reliably during a boot initi-

ated by `kexec`, where `kdump`'s dump capture kernel is triggered by various controlled crashes. It was believed that drivers may not behave correctly if they do not go through the normal BIOS initialisation process. It was found that two drivers had bugs which were reported to the mailing lists and the remainder came up and were writable. A *white list* of the Adapters/drivers was compiled (see table 1). One other boot issue was found with IOAPIC and the timer which caused a panic at boot time on AMD Opteron(tm) Processor 244.

The tests began on an AMD 244 processor, but, as mentioned before, found that for certain crash types the system panicked when booting. The underlying issue was that the APICs and the timer are not brought up in proper sequence (the right fix would be to configure the APICs before trying to bring up the timer).

Then tests were started on Dell servers, with the

internal Megaraid SCSI. On the very first test, there was found an initialisation problem when booting in the kdump kernel. Looking into this it was found that the interrupts were being enabled by the Megaraid driver earlier than they could be handled in the start-up process. The fact that it was not the driver we intended to test also confirmed that limiting the boot process in the initrd would simplify the testing greatly. Thus, an initrd was created that would mount the test disk only, then reboot, rather than switching to root filesystem.

Emulex Fiber Channel (lpfc) tested successfully for the full series.

LSI MPT SAS (mptsas) had an issue with insmod in the kdump kernel where the disk that normally loaded last would load first and last. It was found that if the module was removed (*rmmmod*) and then loaded *modprobe* again, the same thing would happen. This was reported to the correct mailing list but unfortunately precludes this driver from making the white list currently. The disks were mountable and writable otherwise.

## 7.2 Fake interrupt-based testing

As discussed before, in kdump case the underlying devices can very well have pending interrupts (generated in the context of the first kernel) while the dump capture kernel is booting. Ideally quiescing the underlying devices should suffice but not all drivers do this, either because it is not possible (the device does not provide that capability) or because they did not contemplate this case.

The bottom line is that drivers ought to be able to handle interrupts coming in as soon as the interrupt handler is registered. If we wanted to test drivers to see if they behave as expected when such an interrupt comes in we could consider using LKDTT. That would certainly work in many cases (see section 7.1), but, in many others, in order to leave an interrupt in a pending state it would be necessary to introduce driver-specific crash points and possibly further modifications that not fit well within the LKDTT infrastructure. However, a different

approach is possible: the kernel could generate a *fake* interrupt that looks as if it had come in before its interrupt line was allocated. Such a fake interrupt can be easily implemented:

```
int request_irq(...)
{
    struct irqaction *action;
    int retval;
    ....
    if (irqflags & IRQF_DISABLED) {
        unsigned long flags;

        local_irq_save(flags);
        retval = handler(irq, dev_id);
        local_irq_restore(flags);
    } else
        retval = handler(irq, dev_id);
    if (retval == IRQ_HANDLED) {
        printk(KERN_WARNING
               "%s (IRQ %d) handled a spurious
               interrupt\n",
               devname, irq);
    }

    retval = setup_irq(irq, action);
    if (retval)
        kfree(action);

    return retval;
}
```

The underlying idea is that a driver must be ready to handle interrupts as soon as it calls `request_irq()`, because even if the driver has not explicitly activated the underlying device it might already be operative if this is a kdump dump capture kernel. The pending interrupt is simulated by invoking the interrupt handler in `request_irq()`. This is done before actually registering the interrupt with `setup_irq()`, to make sure that a *real* IRQ does not run in parallel with the fake one.

It is worth mentioning that most of this code was taken from an existing debugging mechanism aimed at testing shared handlers. If the underlying device is sharing an interrupt line the driver should be prepared for an interrupt to happen immediately. As we have seen, with the advent of kdump this requirement was extended to non-shared handlers too.

These patches can affect device drivers in several ways and 5 possible scenarios can be identified:

1. Shared handlers that behave as expected and ignore spurious interrupts.
2. Shared handlers that unconditionally handle any incoming interrupt.
3. Shared handlers that cannot distinguish between interrupts that it needs to handle and interrupts generated by other devices in certain situations.
4. Non-shared handlers that behave as expected and ignore spurious interrupts.
5. Drivers for legacy devices in which it is not possible to determine whether a given interrupt has actually come from the underlying device.

Obviously, 1) and 4) are the ideal cases and the majority of drivers (probably) fall into this group.

2) means that the driver cannot cohabit harmoniously with other drivers and it would end up handling any interrupt coming in from that interrupt line regardless of its origin.

Most devices have a interrupt pending bit in some port or an interrupt cause register (ICR) that can be checked by the driver to determine what interrupts it needs to handle. The problem is that some devices are operational before `request_irq` has been called and react to external events by updating the interrupt pending bit or the ICR. This means that after actually registering the interrupt handler, any interrupt coming in would be handled. 3) refers to such cases.

When it comes to 5) there is not much we can do. Sometimes you simply cannot tell.

The bottom line is that we have to decide how to tackle 2), 3) and 5). In most cases, 2) and 3) constitute a legitimate bug, because if the driver is not fixable then the flag `IRQF_SHARED` (which is used to identify an interrupt handler as a shared handler) should not have been set in the first place. Regarding 5), there is no choice, so it seems we would need to mark such drivers

somehow (another possible solution would be calling the interrupt handler from `request_irq` only when `IRQF_SHARED` is set, because shared handlers should not be affected by this problem).

If we find drivers that are not fixable we should probably consider this new behaviour on a per-driver basis. This would probably require passing a new flag into `request_irq`. Besides, when such a driver is detected we should emit a warning that it may not work properly in a `kdump` kernel.

### 7.2.1 Mainline kernel inclusion

But is this new functionality appropriate for the mainline kernel?

On the one hand, some preliminary testing caused quite a lot of breakage for a relatively small sample of machines (see 7.2.2 for details). The problem is that we are weeding things out at runtime. Some drivers do not get used by many people and users of some architectures (embedded in particular) tend to lag mainstream for a long time. As a consequence it could be years before all the fallout from this change is finally wrapped out.

On the other hand, if this functionality were to be merged into mainline, we would have *everybody* testing device drivers without even knowing it. Pretty much everything which goes wrong is due to incorrect or dubious driver behaviour, and there is value in weeding these things out. And most importantly, users may suffer the issues we are trying to weed out anyway, even if this new behaviour is not added. The difference is that with this new functionality it is possible to catch potential problems relatively easily, because any incorrect behaviour this may cause will be easily reproducible and, in most cases, will reveal itself early at boot time.

As things stand now, we will probably keep seeing occasional crashes and strange behaviour in `kexec`-booted kernels, which in some cases will be due to incorrect handling of spurious interrupts. Besides, such problems are really difficult to reproduce because, commonly, we would need to hit

an obscure corner case.

## 7.2.2 Test results

To have a glimpse of the impact of this new functionality some testing was performed on several architectures:

- *x86\_64 [EM64T]*  
CPU: Intel(R) Xeon(TM) CPU 3.20GHz (2 cpus).  
SCSI: Adaptec AIC-7902B U320.  
IDE: Intel Corporation 82801EB/ER (ICH5/ICH5R) IDE Controller.
- *x86\_64 [Opteron]*  
CPU: AMD Opteron(tm) Processor 240 (2 cpus).  
IDE: Advanced Micro Devices [AMD] AMD-8111 IDE (rev 03).
- *ia64*  
CPU: Itanium 2 Madison (2 cpus).  
SCSI: LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320 SCSI.  
IDE: Intel Corporation 82801DB (ICH4) IDE Controller.
- *i386*  
CPU: Intel(R) Xeon(TM) CPU 2.40GHz (2 cpus).  
IDE: Intel Corporation 82801EB/ER (ICH5/ICH5R) IDE Controller (rev 02).

Before proceeding to the test results it is necessary to define what the right behaviour is. Essentially, when the driver *receives* the fake interrupt the interrupt handler should just ignore it and, consequently, return `IRQ_NONE`. If the interrupt is handled, i.e. `IRQ_HANDLED` is returned, that is a sign that something is wrong in the driver and could lead to incorrect behaviour of the driver. In the latter case, the kernel would emit error messages like these:

```
# dmesg | grep -i spurious
i8042 (IRQ 12) handled a spurious interrupt
rtc (IRQ 8) handled a spurious interrupt
MPU401 UART (IRQ 10) handled a spurious
interrupt
parport0 (IRQ 7) handled a spurious interrupt
eth0 (IRQ 54) handled a spurious interrupt
```

Finally, in some cases the interrupt could try to access a structure that has not been initialised yet and cause a kernel crash.

Below two drivers that were found to show incorrect behaviour are analysed in detail:

- *i8042*  
The culprit here was the `i8042_aux_test_irq` interrupt handler unconditionally returning `IRQ_HANDLED` despite being marked as `IRQF_SHARED`. This was fixed by considering that it should not handle an interrupt when there is no data to read and, accordingly, it returns `IRQ_NONE` in such cases.
  - *e1000*  
The problem here is that the network card is active before it has been assigned an interrupt line. This means that if the card detects an event such as a link status change it will update the status register (ICR) by setting the corresponding flag, even if `request_irq()` has not been invoked yet. In such a case, once the interrupt handler is set if a spurious interrupt comes in it will be handled. The reason is that the handler checks the ICR to determine whether the underlying the device has generated an interrupt or not, and the flag that indicates a link status change will happen to be set. The right solution for this problem would be to prevent the device from reacting to such events before an interrupt has been allocated.
- Other drivers found to show potentially problematic behaviour were: *rtc*, *parport\_pc*, *mpu401\_uart*, *tpm\_tis*.

### 7.2.3 Future work

The approach used to test the behaviour of drivers when there are pending interrupts could be applied to test other problematic areas. Many modern devices keep a lot of state information that will be preserved across a kdump reboot. This can potentially cause a lot of breakage because all that state is likely to be invalid in the context of the new kernel, and any action the kernel takes based on that state can have catastrophic consequences. For example, responding to an interrupt from a SCSI device its driver could read a message from the reply FIFO of the device which corresponds to an IO request emitted by the first kernel; then the driver would try to access the associated message frame, but the message frame would not be valid in the context of the new kernel, so the kernel would panic.

The failure injection framework of the Linux kernel could be improved in such a way that it is possible to recreate this type of crash scenarios. To achieve that it would be necessary to *inject* errors at the bus and device driver levels. The advantages are obvious: improved reproducibility and ease of testing.

## 8 Conclusion

After a period of hectic development kdump has reached a level of maturity where it can be considered ready for the enterprise. As a proof of this, the major Linux distros will ship kdump in the enterprise editions of their products.

However, there are still some issues that remain to be resolved, especially in device drivers, which were not designed with kexec and kdump in mind (in a majority of cases kdump did not even exist when the driver was written). The basic problem is that when the kdump kernel is booting the underlying devices can potentially be in any random state (as a consequence of the crash in the first kernel), which may expose two types of bugs:

- The kernel or a driver sees unexpected be-

haviour in a device that it deems anomalous and, consequently, raises a panic or an oops.

- The kernel or a driver did not contemplate all eventualities and makes assumptions that do not hold true in certain crash scenarios. This could lead to the execution of an invalid operation (such as a null pointer dereference) and, in turn, to a crash of the dump capture kernel.

These bugs have passed unnoticed so far because traditionally one could count on the firmware or BIOS doing the dirty work for us and putting all the devices in the system into a good known state, which is not the case after a soft-reboot. That said these problems have always been there to a certain extent due to the existence of buggy BIOSes.

The fact that some drivers do not get used by many people means that it could be years before all the problems are detected and weeded out. Besides the seemingly random nature of these bugs makes it very difficult to reproduce many of them. Thus, to speed up the process and find bugs before they hit end-users testing is fundamental.

The obvious approach to testing is to install and configure kdump and deliberately cause a kernel crash, so that control is transferred to kdump and we can see how the dump capture kernel and its drivers behave after the crash. There is a test suite, LKDTT (Linux Kernel Dump Test Tool), which does exactly this.

LKDTT has revealed many issues in the kernel's crash detection mechanisms, the reboot path to the dump capture kernel, and the kernel itself when it is being kdump-booted. However, for some crash scenarios there would be consistent failures and for others there were inconsistent failures. The underlying issue is that once that control is handed over to second kernel LKDTT loses control of the testing process and after that there are too many variables in play. Thus, even though LKDTT can recreate crash scenarios rather faithfully sometimes the outcome will differ from one run to the next.

The solution to this lack of reproducibility is to do all the testing using just one kernel. The basic idea is quite simple: simulate the occurrence of exceptional events and situations that are common when a kdump-kernel is booting, such as pending interrupts and devices not responding to certain commands. This can be done using a failure injection mechanism or generating *fake* events at boot time. The latter is particularly useful to test how drivers behave when there are pending interrupts.

If fact the unconditional generation of a *fake* interrupt at driver initialisation time is being considered for inclusion in mainline, so that we would have *everyone* testing drivers without even knowing it. The problem is that weeding things out at runtime can be problematic at first, because many latent bugs may be exposed, and it will take some time before they get fix and things stabilise.

## 9 Acknowledgements

First of all, I would like to express my deepest gratitude to Judith Lebzelter for her vital contribution to OSDL's kdump testing project and to this paper, which presents the results of her work in section 7.1. Thanks also to Tom Hanrahan, former director of engineering at OSDL and the Linux Foundation, for his unconditional support. Without these two individuals doing such a great job this project would not have been possible.

This material is based upon work supported by the NTT Open Source Software Center, NTT Data Intellilink, the former OSDL and the Linux Foundation.

## References

- [1] Linux kernel crash dump (LKCD) home page. <http://lkcd.sourceforge.net/>.
- [2] Diskdump patches. <http://sourceforge.net/projects/lkdump/>.

- [3] Michael K. Johnson. Red Hat, Inc.'s network console and crash dump facility, 2002. <http://www.redhat.com/support/wpapers/redhat/netdump/>.
- [4] Mission critical linux in memory core dump home page. <http://mclx.com/projects/mcore/>.
- [5] bootimg home page. <http://bootimg.sourceforge.net/>.
- [6] Kdump home page. <http://lse.sourceforge.net/kdump/>.
- [7] Vivek Goyal, Eric W. Biederman, and Hariprasad Nellitheertha. A kexec based dumping mechanism. In *Ottawa Linux Symposium*, July 2005. <http://lse.sourceforge.net/kdump/documentation/ols2005-kdump-paper.pdf>.
- [8] Fernando Luis Vázquez Cao. Linux kernel dump test tool (LKDTT) home page, 2006. <http://lkdttd.sourceforge.net/>.
- [9] Vivek Goyal, Neil Horman, Ken'ichi Ohmichi, Maneesh Soni, and Ankita Garg. Kdump: Smarter, easier, trustier. In *Ottawa Linux Symposium*, June 2007. <https://ols2006.108.redhat.com/2007/Reprints/goyal-Reprint.pdf>.
- [10] Hariprasad Nellitheertha. Reboot linux faster using kexec, 2004. <http://www-128.ibm.com/developerworks/linux/library/l-kexec.html>.
- [11] Kexec-tools code. <http://www.xmission.com/~ebiederm/files/kexec/>.