

# Reducing downtimes by replugging device drivers

Damien Martin-Guillerez - MIT 2  
Work under Henry Levy and Mike Swift  
Nooks Project

June-August 2004

## Abstract

Computer systems such as servers or real-time systems depend on reliability to offer a continuous service. Downtimes (breaks in the continuity of service) are generally due to hardware or software failures and updates.

Most of system crashes are caused by device drivers (85% of Windows XP crashes are caused by device drivers and there are seven times more bugs in Linux drivers than in the kernel [7]). Nooks [8, 6] is a subsystem to protect most of the faults from the driver to crash the kernel. The shadow drivers [7] are an extension to Nooks in order to safely recover drivers after a fault.

Nooks and the shadow drivers can decrease kernel crashes from driver faults by 99% but system downtime is still required to update drivers. And, if Nooks intercepts a fault, we want to update to a non-faulty driver. In order to update without downtime (i.e. without unloading the module), we have implemented a replugging system to upgrade drivers without service interruption in the shadow subsystem.



*University of Washington*  
Computer Science & Engineering

# Contents

<b>Abstract</b>	<b>1</b>
<b>Contents</b>	<b>2</b>
<b>Introduction</b>	<b>3</b>
<b>1 Nooks internals</b>	<b>3</b>
1.1 Driver Isolation . . . . .	3
1.1.1 Procedure call wrapping . . . . .	3
1.1.2 Virtual memory protection mechanisms . . . . .	4
1.1.3 Fault handling . . . . .	4
1.2 Recovering from faults : the shadow drivers . . . . .	4
1.2.1 Shadow driver description . . . . .	4
1.2.2 Recovery process . . . . .	5
1.3 Some results . . . . .	5
<b>2 Nooks replugging</b>	<b>6</b>
2.1 Replugging definition . . . . .	6
2.2 Implementation . . . . .	8
2.3 Tests . . . . .	9
<b>3 Conclusion and future concerns</b>	<b>9</b>
<b>Thanks</b>	<b>10</b>
<b>References</b>	<b>10</b>
<b>List of figures and tables</b>	<b>10</b>
<b>Annex : results of replugging compatibility tests</b>	<b>11</b>

# Introduction

Operating systems are often designed to serve on different hardware and for different purposes. In order to do that, extensions such as loadable kernel modules (LKM) in Linux have been developed to match the specific hardware. Those device drivers are often developed by third-parties and, as a consequence, they frequently have bugs and require updates.

In order to decrease driver faults that can crash the system, many solutions have been proposed such as new extension infrastructures, new hardware, real-time model-checkers and adding proofs of correctness within the drivers. All those solutions either introduce incompatible changes, lack efficiency or are not bug tolerant techniques. The Nooks subsystem tries to isolate most faults from device drivers (so they will not corrupt the kernel) by isolating the driver from the kernel. If a fault happens, Nooks cleans the driver, unloads it and reloads it, which causes an interruption of service from the driver.

To prevent the interruption of service that needs to interrupt applications using the service, a shadow driver takes the place of the driver during recovery, queues all the requests to the drivers, reloads the driver and replays all the relevant requests to the driver, such as configuration settings, and then makes the reloaded driver handles the queued requests.

The main problem with shadow drivers is that we can trigger the bug again in the driver. So, after recovery, we want to be able to update the driver without interruption. Hence, we needed a replugging system to be able to change the driver during recovery or even before a fault occurs.

The paper is organized as follows. First, Nooks driver isolation is presented in Section 1.1 then recovering using shadow drivers will be discussed in Section 1.2. After that, Section 2 presents the replugging system I have implemented inside Nooks. Finally, Section 3 discuss about future concerns in non-stop systems.

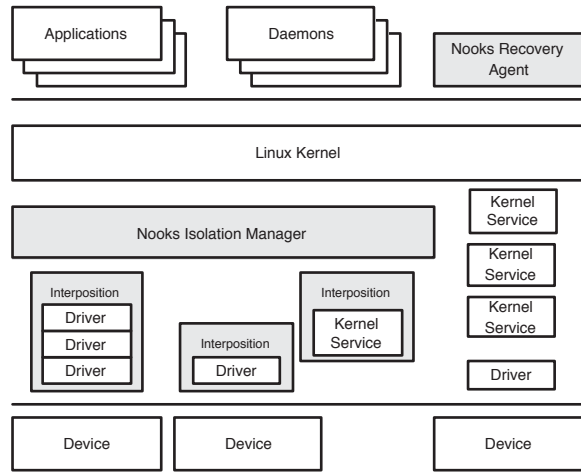


Figure 1: THE NOOKS LAYER (SHADED) INSIDE THE LINUX OS

*showing wrapped Linux extensions executing in isolated protection domains.*

## 1 Nooks internals

### 1.1 Driver Isolation

Nooks tries to protect the kernel from most (*not all*) extension bugs (*not abuse*) by using procedure call wrapping between the kernel and the driver with virtual memory protection.

When we load a driver into the kernel and we want to protect it using the Nooks subsystem, we create a protection domain called a "nook" and we insert our driver inside this domain (of course, we can load several LKMs inside the same nook). Once the driver is inside the nook, every operation done by the driver will be under the protections previously discussed (fig. 1).

#### 1.1.1 Procedure call wrapping

When loading a module inside a nook, each reference to a kernel function is replaced by a call to a wrapper to that function. Inside the kernel all pointers to the functions of the module are replaced by wrappers of the nook.

After that each function pointer given by the driver to the kernel to handle requests is also replaced by a wrapper. Each call from the kernel

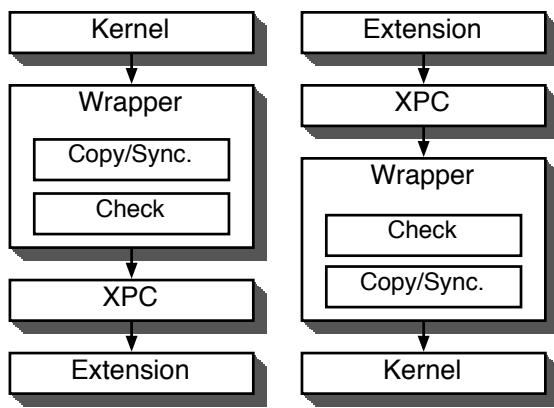


Figure 2: CONTROL FLOW OF EXTENSION AND KERNEL WRAPPERS.

to the driver or from the driver to the kernel is wrapped as shown in fig. 2.

These wrappers try to prevent faults from bad arguments in calls to the kernel or the driver, and they also install new wrappers when needed and switch between the different protection domains.

### 1.1.2 Virtual memory protection mechanisms

Drivers inside a nook are prevented from corrupting kernel or other driver's memory using memory protection. Each time we enter a function protected by a nook we switch to a new protection domain that allows the function to access the data pages of the modules inside the nook in read and write and that refuse write access to the kernel space or other drivers' spaces (fig. 3).

If the driver tries to write directly into the

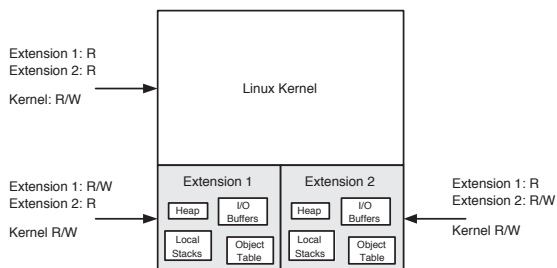


Figure 3: PROTECTION OF THE KERNEL ADDRESS SPACE

kernel or another driver's memory, which it is not supposed to do, we can intercept it by catching a memory access violation fault. Of course, if the driver tries to access a non-mapped address (like trying to read / write a NULL pointer), a page fault will be triggered and Nooks will catch it.

### 1.1.3 Fault handling

Once Nooks catches a fault, it begins by cleaning and releasing the resources the driver used. Then it unloads the failed module and notifies a user-mode agent that a fault occurred from that driver (Nooks recovery agent from fig. 1). If that driver is registered in the userland as a nook driver (i.e. the driver has been loaded with `modprobe`), the nook agent reloads the driver into the kernel.

Recovery is not very efficient in that case because it must kill applications using the driver and requires that some modules to be still accessible during the recovery (e.g. those needed to reload the driver as the disk driver). While it is possible to do so for certain drivers such as sound card drivers, it becomes hard to do so with drivers like network drivers or even impossible if the driver is the file system or the hard-drive driver, which are needed for restarting the driver

## 1.2 Recovering from faults : the shadow drivers

Because of the lack of efficiency of classical Nooks recovery, the Nooks team has implemented shadow drivers.

### 1.2.1 Shadow driver description

A shadow driver usually (i.e. outside of the recovery process) transparently hooks the driver calls as discussed in Section 1.1.1. It traces interesting requests sent to the driver such as configuration requests, device opening from the userland, *etc...* in order to replay them later (fig. 1.2.1).

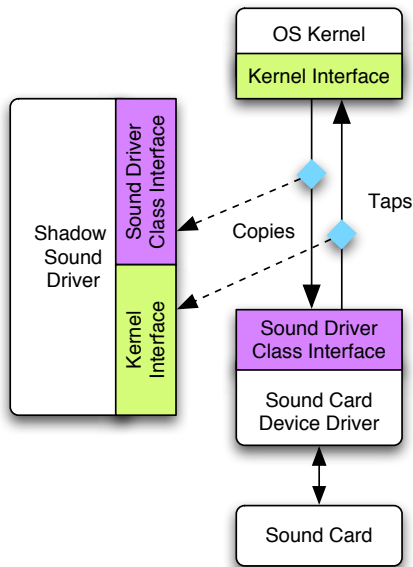


Figure 4: A SHADOW DRIVER OPERATING IN PASSIVE MODE

*Taps are inserted between the kernel and the driver to monitor all communication between the two.*

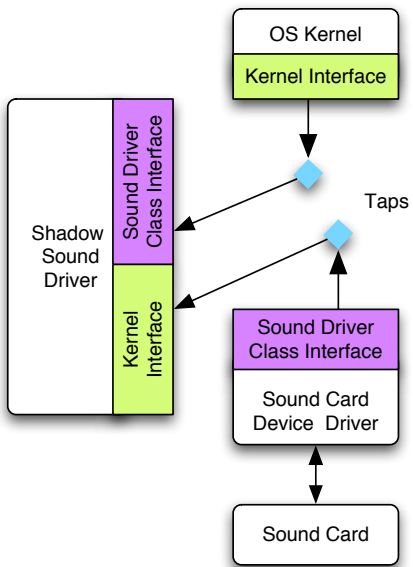


Figure 5: A SHADOW DRIVER OPERATING IN ACTIVE MODE

*The shadow driver takes over communications between the kernel and the failed driver.*

### 1.2.2 Recovery process

Once a recovery is triggered, the shadow driver takes over communications between the kernel and the failed driver (fig. 1.2.2) and handles requests requests instead of the suited driver. During that time, the normal cleaning process of Nooks is triggered but the driver is not unloaded. Once the normal cleaning process is finished, the old driver's data field is reset to its original state and the replaying begins.

The replaying is a step of the recovery process where each relevant request logged during the passive mode (see 1.2.1) is sent again to the driver. After that step, the driver is in the same state as it was before recovery and can now handle again the request from the kernel. Then the shadow driver return control to the driver and passes it the queued requests. After that we are back in passive mode.

### 1.3 Some results

To test the efficiency of the Nooks isolation, a fault injection procedure has been added into the kernel by the Nooks team. We can inject a random fault into the driver and see how well Nooks handles it. After running some tests, we can measure the effectiveness of Nooks

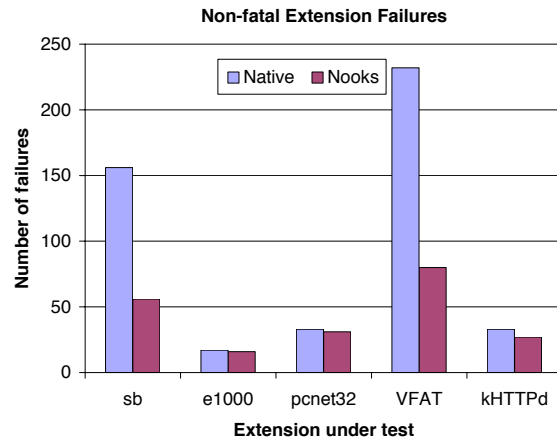


Figure 6: RELIABILITY EXPERIMENTS - NON-FATAL FAULTS

*sb is a sound card driver, e1000 and pcnet32 are network drivers, VFAT is the extended FAT filesystem driver and kHTTPd is a kernel web server.*

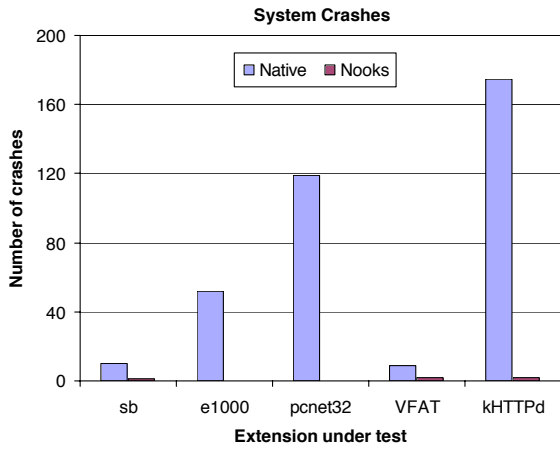


Figure 7: RELIABILITY EXPERIMENTS - FATAL FAULTS

(fig. 6 & 7). We can see that nooks usually decrease the number of non-fatal extension failures to less than one third and that non-fatal extension represents more than half of all failures. Figure 7 shows us that more of 99% of fatal faults are handled by Nooks. Those two figures show us that Nooks handles more than 75% of usual faults from drivers.

Nooks and even shadow drivers usually does not decrease too much the performance of the driver (fig. 8). However when the frequency of communications between the kernel and the driver is huge, like for the kernel web server, the performance can be reduced to less than 50% of the original performance (fig. 9).

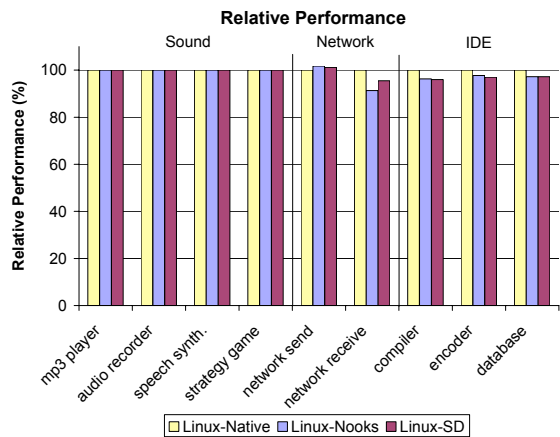


Figure 8: MEASURES OF NOOKS PERFORMANCES

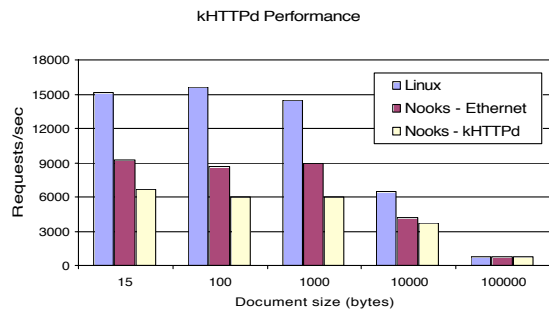


Figure 9: MEASURE OF NOOKS PERFORMANCES ON THE KERNEL WEB SERVER

## 2 Nooks replugging

We already seen that Nooks can prevent driver faults from corrupting the kernel and making it crash. In addition of Nooks, shadow drivers can prevent drivers downtime by recovering them after a fault. This section will present the replugging system I have implemented that can reduce system downtimes from driver updates.

### 2.1 Replugging definition

When a driver generates a fault, we often want to switch to a newer (or an older) version of this driver without the bug. Of course, while we are looking for reducing downtimes, we do not want to have to stop that driver : here comes the idea of replugging. Projects like K42 [5] have developed the idea of component hot-swapping using objects tracking and requests queueing to load the new version of a driver but K42 is a new system and requires drivers to be completely rewritten. Replugging should also ensure to be able to swap drivers without a brutal cleaning of the old version.

In Nooks with shadow drivers, object tracking and requests queueing is already made by the shadow drivers subsystem. As a consequence, implementing a replugging systems was an easy step.

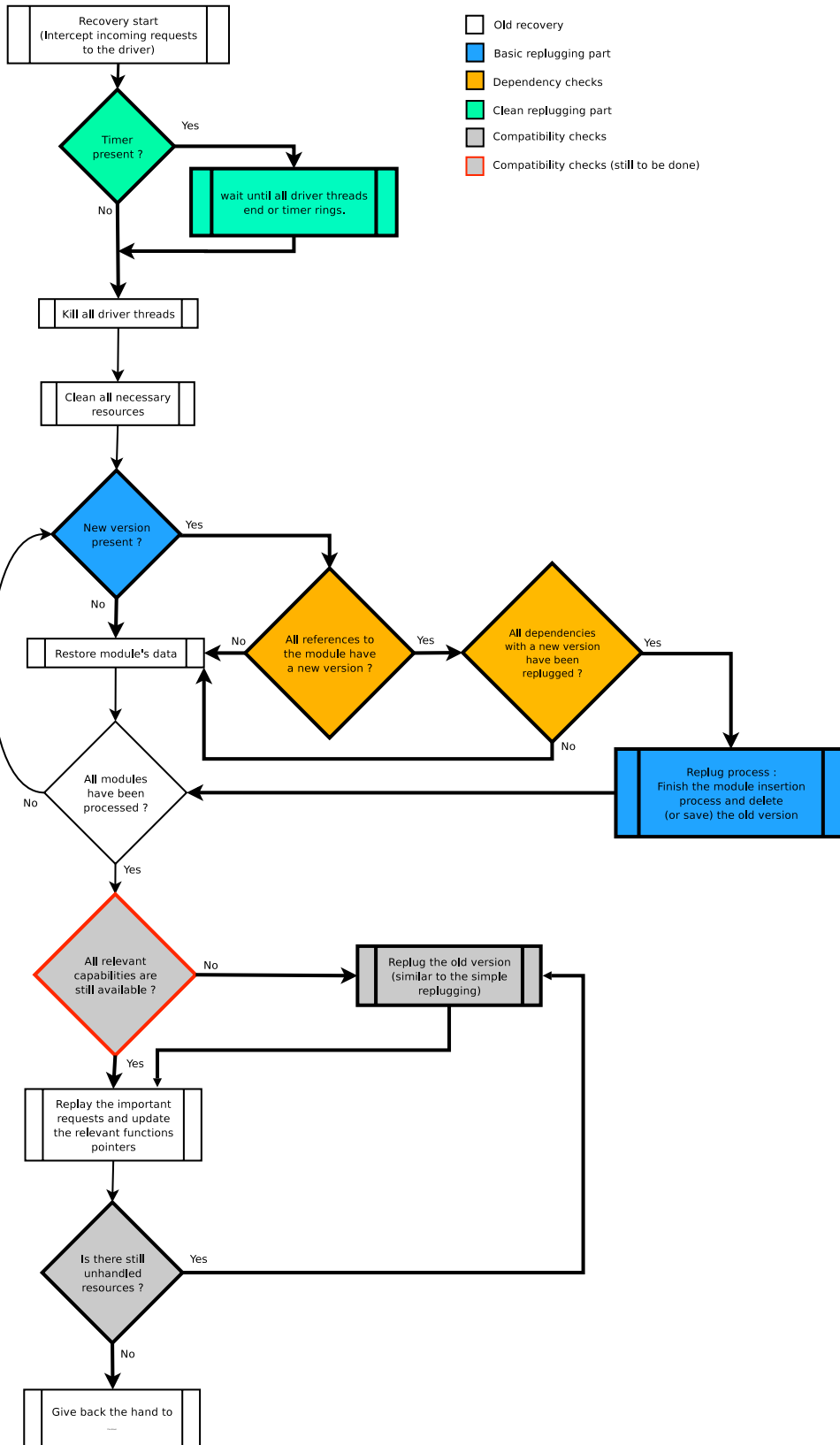


Figure 10: REPLUGGING FLOW CHART

## 2.2 Implementation

First of all, there was a need for the inclusion of a loading system for the new driver. So by adding a pointer to the new version in the module structure in the kernel and by creating modified module loading system calls that just load without initializing the new version we were able to do it. Once those system calls were created, we just needed to load the new driver by freeing the old driver memory and initializing the new version (see blue part of fig.10).

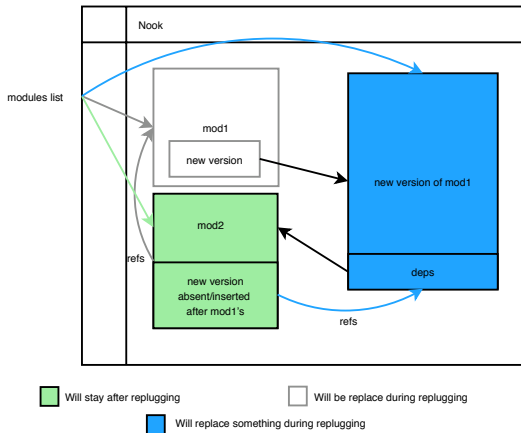


Figure 11: REPLUGGING OF ONE MODULES WITH ONE REF

That simple part finished, we need to worry about what we really need to handle it in a efficient way. To begin, there is the dependency problem : a module can depend on another module for some functions or variables so we much ensure that the dependencies are still correct after the replugin. To do so, the symbol resolver asks for symbols from loaded new versions of all modules if he tries to load a new version of an existing module. During replugin, dependencies are checked to see if all references of a module have a new version with the good symbol resolution, if it is the case then all those modules will be replugged else only the modules with a new version that has been loaded before will be replugged (fig. 11 & 12).

We would like to be able to cleanly replugin a driver when updating it without a fault occur. For that, we want the driver to be cleanly shut down before replugin instead of killing all its

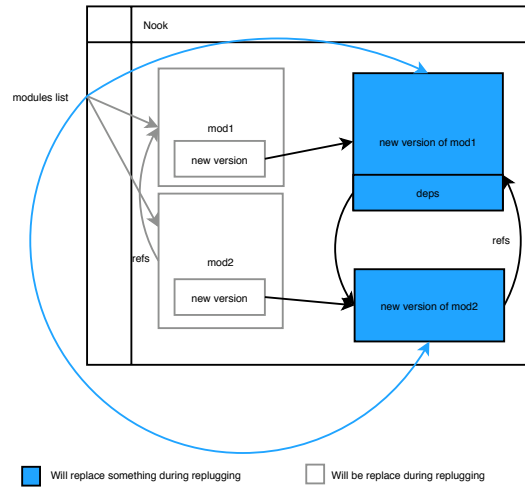


Figure 12: REPLUGGING OF TWO DEPENDENT MODULES

threads. To do so, we just provide a time limit (which can be infinite) to the replugin system for waiting all threads to stop. Of course during the waiting, the shadow driver queues incoming requests to avoid new threads to be created (green part of fig. 10).

```
# ls /proc/nooks/
mil-nook nooks scully
# cat /proc/nooks/nooks
nook          shadowed      state
scully        sound         running
mil-nook     netdev       running
global-nook  no           running
# cat /proc/nooks/scully
name:         scully
type:         sound
last fault:   14535 s
reason:       default fault
eip:         c886913d
module:       scull
last replugin: 14535 s

module      replugin    last replug state
scull      no          succeed
#
```

Table 1: /PROC ENTRIES OF THE REPLUGGING SYSTEM

We also want to have a mean to communicate some informations to the userland so a userland program can keep trace of driver updates (for bug report and choosing new updates to do). This is done using procfs entries that specify which nook has a shadow driver and which one, and a proc entry for each shadowed nooks let us know some informations about last replugin and last fault (tab. 1).

Finally, an important case to handle is the

compatibility check. We can say that a driver is compatible to an old one if it supports at least the same capabilities as the old one and use the same protocols. In fact, the new driver does not have to be that compatible with the old one, it just has to support at least what was asked to the old driver. For that, the replugging system considers the driver has to handle all the resources that the old one was using (procfs entries, network interfaces, character devices, etc...). The old driver has also to accept the same configuration requests and to provide capabilities asked by the userland. While we did not implemented it, once replugging done, we should first check for relevant capabilities support then, during the replaying step of the recovery, we should check for answers compatibility. If a compatibility check fails or if the driver fails later, we just replug the old version of the module (grey part of fig. 10).

### 2.3 Tests

First of all, I needed to validate replugging by several ways. I began by replugging a simple driver by itself then I inserted a bugged version of that driver and inserted the correct version as a the replugging version. After that, I have tested a composite driver (two modules) in the same way. Once all that was done, I had to test to some real drivers : `pcnet32` (network driver) and `sb` (sound driver).

Once these validation steps passed, the only thing left is to test compatibility checks by using all versions of the `pcnet32` driver : over 20 different versions of that driver has been successfully replugged (see tab. 2).

## 3 Conclusion and future concerns

As we have seen, Nooks can prevent most driver faults from crashing the system and shadow drivers transparently recover driver from those faults. The implemented replugging system removes downtimes needed for driver

updates but it still need for more compatibility checks as discussed in Section 2.2.

Reducing downtimes is an important concern for systems that need to run 24 hours a day. The current systems usually duplicate expensive hardware to prevent downtime problems. Non-stop systems can help to reduce hardware duplication due to possible failure or updates.

Actually, to keep on reducing downtime we can switch to a micro-kernel written from scratch where every part of the kernel can be updated without system restart like GNU Hurd [2]. Userland program is also a concern, we would like to recover and update applications such as web server without restarting them. For that we can adapt shadow drivers and replugging with userland resources such as network connections. Using the queueing system of shadow drivers and the current system architecture we can also recover them from fault transparently.

Finally, downtimes from hardware seems to be hard to cover but by using duplicate hardware and hot-swappable device like PCMCIA cards we can suppress downtimes due to hardware failures and updates. Such techniques already exist for hard drives (RAID) and just need to be extended to more devices instead of duplicating the whole system.

As we have seen, the path towards non-stop system is still long but worth it. Non-stop software does not seem to far now and can clearly reduce the price of systems that need to run 24 hours a day.

## Thanks

I would like to thanks many people :

- Michael Swift and Henry Levy for mentoring me.
- Brian Milnes for introducing me to the dark paths of Nooks.
- My roommates (Patrick, Shayne and Sean) and theirs friends for being so nice.

## References

- [1] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.
- [2] The GNU Hurd Kernel. <http://www.gnu.org/software/hurd/hurd.html>.
- [3] The Linux Kernel Archives. <http://www.kernel.org>.
- [4] A. Rubini. *Linux Device Drivers*. O'Reilly, 1998.
- [5] C. A. N. Soules, J. Appavoo, K. Hui, D. D. Silva, G. R. Granger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenberg, and J. Xenidis. System support for on-line reconfiguration. In *USENIX Annual Technical Conference*, 2003. K42 Project (<http://www.research.ibm.com/K42>).
- [6] M. Swift, H. M. Levy, and B. N. Bershad. Improving the reliability of commodity operating systems. In *The 19th ACM Symposium on Operating Systems Principles*, 2003. <http://www.cs.washington.edu/homes/mikesw/nooks/sosp-present.pdf>.
- [7] M. Swift, H. M. Levy, B. N. Bershad, and M. Anamalai. Recovering device drivers. In *The 6th Symposium on Operating Systems Design and Implementation*, 2004. Draft Paper.
- [8] M. Swift, H. M. Levy, S. Martin, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In *The Tenth ACM SIGOPS European Workshop*, 2002. <http://www.cs.washington.edu/homes/mikesw/nooks/nooks-sigops.pdf>.
- [9] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

## List of Figures

1	THE NOOKS LAYER (SHADED) INSIDE THE LINUX OS . . . . .	3
2	CONTROL FLOW OF EXTENSION AND KERNEL WRAPPERS. . . . .	4
3	PROTECTION OF THE KERNEL ADDRESS SPACE . . . . .	4
4	A SHADOW DRIVER OPERATING IN PASSIVE MODE . . . . .	5
5	A SHADOW DRIVER OPERATING IN ACTIVE MODE . . . . .	5
6	RELIABILITY EXPERIMENTS - NON-FATAL FAULTS . . . . .	5
7	RELIABILITY EXPERIMENTS - FATAL FAULTS . . . . .	6
8	MEASURES OF NOOKS PERFORMANCES	6
9	MEASURE OF NOOKS PERFORMANCES ON THE KERNEL WEB SERVER . . . . .	6
10	REPLUGGING FLOW CHART . . . . .	7
11	REPLUGGING OF ONE MODULES WITH ONE REF . . . . .	8
12	REPLUGGING OF TWO DEPENDENT MODULES . . . . .	8

## List of Tables

1	/PROC ENTRIES OF THE REPLUGGING SYSTEM . . . . .	8
2	REPLUGGING COMPATIBILITY TESTS ON DRIVER PCNET32 . . . . .	11

## Annex : results of replugging compatibility tests

Table 2 shows tests of running several versions of the same driver (`pcnet32` network device driver). This driver is composed of two modules (`pcnet32` & `mii`). The tests have been run using the Linux 2.4.18 driver and the Linux 2.4 CVS tree from kernel.org [3]. Versions from 1.8 to 1.15 for the `mii` module and 1.16 to 1.32 for `pcnet32` use a new structure which is incompatible with the 2.4.18 kernel (compilations of those versions failed).

module's name	version	changes	replugged from	Success
pcnet32	Linux 2.4.18	-	CVS 1.16	yes
	CVS 1.2	Initial revision	Linux 2.4.18	yes
	CVS 1.3	Idem as CVS 1.2	CVS 1.2	yes
	CVS 1.4	Bug inside an if (& instead of &&)	CVS 1.3	yes
	CVS 1.5	Include change (minor change)	CVS 1.4	yes
	CVS 1.6	32-bit mode related bugs correction	CVS 1.5	yes
	CVS 1.7	Minor changes	CVS 1.6	yes
	CVS 1.8	Hardware related corrections	CVS 1.7	yes
	CVS 1.9	Coding convention changes (minor changes)	CVS 1.8	yes
	CVS 1.10	Fix oops during insmod, plug i/o resource leak	CVS 1.9	yes
	CVS 1.11	Minor changes	CVS 1.10	yes
	CVS 1.12	Removed an ID of the PCI table	CVS 1.11	yes
	CVS 1.13	Code cleaning (minor change)	CVS 1.12	yes
	CVS 1.14	PowerPC Code (minor change)	CVS 1.13	yes
	CVS 1.15	Total change of the IOCTL function	CVS 1.14	yes
	CVS 1.16	Name changes (minor change)	CVS 1.15	yes
CVS >1.16	Incompatible structure changes	-	compilation errors	
mii	Linux 2.4.18	-	CVS 1.7	yes
	CVS 1.2	Initial revision	Linux 2.4.18	yes
	CVS 1.3	Minor changes	CVS 1.2	yes
	CVS 1.4	Bug in chip initialization	CVS 1.3	yes
	CVS 1.5	2 functions added	CVS 1.4	yes
	CVS 1.6	Name changes (minor changes)	CVS 1.5	yes
	CVS 1.7	Tests added	CVS 1.6	yes
	CVS >1.7	Incompatible structure changes	-	compilation errors

Table 2: REPLUGGING COMPATIBILITY TESTS ON DRIVER `PCNET32`