International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2019): 7.583

Methods of Improving the Robustness of System Software

Karthik Poduval

Email: *karthik.poduval[at]gmail.com*

Abstract: ***System software refers to all the low-level software that runs on an operating system providing service to application. System software may provide hardware abstraction or provide an OS abstraction. System software is often expected to be more robust, yet there may still be different types of problems that can affect the robustness of system software. This review paper firstly defines some of the common problems related to robustness of system software and does a study of related work that try to address the different problem areas.***

Keywords: system software, operating systems, device drivers

1. Introduction

System software can be termed as the lower layers of software that are Operating System specific and help build a layer of service abstraction to user applications. Device drivers, file system code, networking stack are common examples of system software. Of the above types of system software, device drivers are the ones that are subject to more frequent changes as hardware changes are more rapid as compared to file system or networking stack changes. A study [18] has shown that in the Linux 2.4.1 kernel, drivers constitute 1.5 million lines of code while file system comprise 0.1 million lines of code. The above statistic is for the Linux kernel source tree which includes code for multiple architectures and platforms. Hence various device driver sources are present in the kernel source tree. There are even many other driver sources which are often supplied separately. Based on the architecture and platform, a smaller set of system software, like one or more file systems, one or more network stacks and several device drivers are put together into the kernel configuration. The number of device drivers usually exceeds the total number of file system and networking stacks put together. Another important factor is that device drivers are contributed from a much larger community and not all drivers are officially made a part of the Linux kernel source tree.

Drivers downloaded from 3rd party vendor websites, may often not be tested across various versions of the kernel. Some of what is said above its even true for file system or networking code. To summarize, any part of system software that is external to the core kernel and is permitted to execute in kernel mode is a potential threat against system robustness and stability. In the rest of the paper, we study the different types of problems that could affect robustness and discuss the solutions to various problems offered by different operating systems. We continue discussing the problems that affect robustness in monolithic and microkernel operating systems. Device drivers in many monolithic operating systems are run as a part of the kernel image.

This gives them an added (with respect to user space programs) capability in the form of: -

- Access to kernel Address Space - possibility of accidental corruption of kernel data structures.
- Access to kernel functions that go beyond system call's behavior.
- Access to system hardware protected by the kernel.

Different systems constrain the way drivers are a part of the kernel and thus constrain the degree and type of access of the driver to the rest of the kernel. For example, in Linux, a driver could be both implemented as module as well as build as a part of the kernel image. Building the driver as a module restricts the calling capability of the driver to only the exported function in the kernel. Some reasons for allowing the drivers to run in Kernel mode and in the kernel address space:-

a) Kernel mode is often the privileged mode of a processor and hence has unrestricted access to the processor's resources. Some operating systems do not have a very well build up API framework for low level access of processor resources in privileged mode accessible code. Drivers in such operating systems need to be run in kernel mode so that they do not need to implement such low-level functions to gain privileged access to such resources. For example, FreeRTOS, µCos provide only basic thread context switching mechanisms. In such operating systems the kernel, drivers and the user program, all from a part of the same kernel image. There is no real distinction between a driver and an application and everything really runs in most privileged mode of the processor.
b) Being in the kernel address space makes access to the kernel functions a function call away. This is often used to reduce the number of user mode to kernel mode transitions as they can be quite expensive. User mode to kernel mode transitions is facilitated through system calls.
c) Most low-level functions (to access processor architecture specific and platform hardware features) are implemented in the kernel in case of operating systems like Linux and only a few of them are available in the form of system calls to the user mode. Having a driver in the kernel address space enables the reuse of such functions.
d) Drivers need Interrupt Service Routines for the interrupts the Hardware might generate. Most processors expect the ISR's to run in the kernel mode or at least in

one of the privileged modes of the system.

e) The last argument to support running of drivers in kernel mode could be that the Kernel mode could be faster in becauseof the following reasons: -
   - Lesser user-kernel transitions, from an application perspective a single driver call is made and the driver completes all the work in kernel mode and returns when the control is returned back to the user mode. Compare this with a case in which the application has to make several system calls to implement the same functionality the previously mentioned one driver callcould do.
   - Kernel code compiled with different compiler settings, this is true for certain operating systems. In Linux the kernel mode compiler settings do not support floating point operations. The Symbian OS kernel [28] (an OS primarily made for the ARM based cell phone hardware) is compiled in the more efficient 32 bit ARM mode while applications by default usually get compiled with the 16 bit thumb instruction set unless explicitly set to be compiled in ARM mode. The Symbian OS kernel also requires the ARM RVCT compiler as many parts of the kernel are hand optimized keeping the ARM compiler settings in mind.

In microkernel and multiserver operating systems, the drivers are run in user-mode hence they are isolated from the kernel. Utility functions for the drivers are made available in the form of system calls to the kernel or IPC calls to other system servers that implement them. The exact mechanism that exists in different microkernels differs with the common policy of having drivers run in user mode. In certain microkernels, the driver itself serves its clients (another user space process) through IPC calls. Certain applications in microkernel OS might even chose to use shared memory mechanisms to transfer data between user application and kernel. To summarize driver client interaction in microkernel may be through IPC mechanisms or shared memory. In the monolithicvs. microkernel debate, the monolithic kernels got an early lead. Hence a lot of applications and support libraries have been developed for the monolithic kernels (like Linux and Symbian). It could be argued that Linux is not a monolithic kernel. But most device drivers, file system and network code are run in kernel mode in Linux which makes it monolithic from this perspective. The next section discusses the core issues present in both monolithic and microkernel operating systems that impact robustness of system software. Following section discusses various papers that try to solve the core issues.

2. Core Issues

There are some core issues present in both monolithic and microkernel operating systems because of which the robustness of the system gets affected. In this section, we lookat these issues in greater detail.

a) Monolithic Kernels

What is a monolithic kernel? There are many definitions but we define them as kernels that allow system software components like device drivers to execute in kernel mode andshare kernel address space. There are several issues that arise because of running software in kernel mode, let's take a look at some of these issues.

1)Memory Faults

Driver code in a kernel could access invalid memory locations. Linux example: Invalid memory access in driver code causes kernel oops and in ISR causes kernel panic. Such an error can only occur if the driver code accesses a memory location that has not been defined in the Page Tables of the MMU or MPU or for instance marked as read only, or has insufficient access permissions. In all other cases, such memory access could cause corruption of either kernel or even user mode data structures.

2)Added Privileges

Being in the kernel mode means that the driver would have access to all kernel functions that the operating system has access to. Incorrect use of a kernel function could cause catastrophic results and may lead to a dramatic change in behavior of the system. Invalid memory access could also be done through incorrect programming of the DMA. If an IOMMU is not associated with the DMA, such an invalid access of the memory could go totally undetected and might result in aberrant behavior of the system.

3)Resource Hoarding

Suppose a driver code ends up in a code loop which could be possible due to a hardware register or unanticipated state of software, this could lead to hogging of the CPU cycles. In such cases the other processes of the system might get starved of the CPU and in certain cases the system could turn totally unresponsive. To some extent the state of the system depends on whether the kernel is preemptable or not.

4)Resource Leak

Resource leaks caused by allocating and not freeing memory. Since all the kernel mode components share a common heap, such behavior exhibited by kernel mode components could severely affect the OS operation.

b) Microkernels

Now let us discuss core issues present in microkernels. We define microkernels as those operating systems which execute all system software in user space and in a separate address space. Although the fundamental isolation of device driver code from the kernel is achieved in this approach, some issuesstill do exist. We take a look at these issues.

1) Overhead

Since microkernels drivers run as user mode processes, several user- kernel mode transitions are needed to accomplish a given driver function. Additionally, therecould be IPC mechanisms between driver's and their clients that also involve some call overheads.

2) Resource Hoarding

This core issue is also applicable to microkernels only thatin this case the kernel heap does not get affected. The user mode programs could end up allocating large amounts of memory or even get stuck in endless loops. The next two core issues impact monolithic and microkernelsalike.

3) Recovery

In general, once the driver has crashed there should be way torecover the device driver such that the applications using the drivers do not get affected. Recovery is desirable in both monolithic kernels and microkernels. Most current systems donot address recovery resulting in system reboot instead.

***c)* Hardware Errors**

These are the kind of errors that arise due to hardware malfunction. Many times, such issues cause the operating system to panic. This often happens because an undefined state in the hardware causes the corresponding driver to eithermemory fault or get stuck in an infinite loop.

3. Analysis of Solutions to Various Core Issues

In previous section we saw some of the **core issues** that exist in Operating Systems that affect robustness. In this section welook at the solutions so some of these **core issues** presented bydifferent researcher papers.

a) Minix 3

Minix 3 [3], [4] is a multi-server operating system. In Minix 3 device drivers, file systems and other system servers providing various services are implemented as user space processes with a MMU protected address space. Like any other user process, the driver is only allowed to read and write into its own address space. In case of memory mapped I/O devices, the device memory regions are mapped to the corresponding driver's address space using the MEMMAP kernel call. I/O mapped devices are accessed through the DEVIO kernel call. The interrupt processing is also moved to user space through aHWINT IPC mechanism. Now let us see how Minix 3 tries to solve the **core issues**. The **core issues Memory Faults** and **Privileged access** are resolved by bringing the driver to user space. The level of access to Kernel calls and IPC calls is controlled by a per driver policy file.

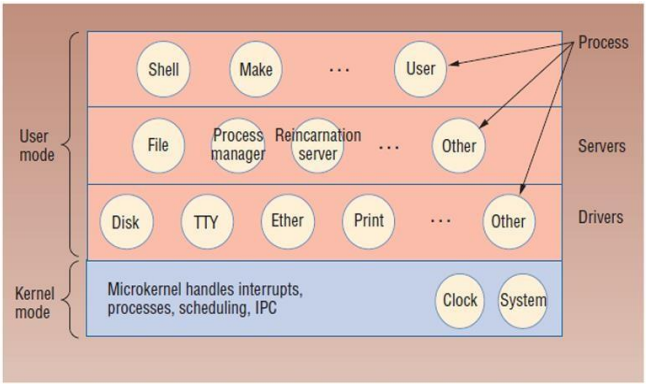

Figure 1: The architecture of Minix 3[4]

An additional feature of Minix 3 protects device drivers from corrupting memory by using DMA. For this Minix 3 requires the presence of IOMMU hardware. DMA, programming that device drivers might want to do, are done with the help of IOMMU server, which internally takes care of programming the IOMMU hardware as well when DMA programming is done thus preventing accidental misprogramming of the DMA. Minix 3's solution to **Resource Hoarding core issue** is by sending out periodic heartbeat requests to all drivers. If a driver fails to respond to one such request, the reincarnation server restarts the driver process. The rest of the **core issues**

remain unaddressed in Minix 3. Minix 3's implementers carried out testing of their operating systems features by introducing software fault isolation techniques. Their system was able to catch most memory faults and also detect driver hangs and restart drivers. However, there was no way of detecting hardware errors and there was no test to determine the overhead of having several IPC and Kernels calls to implement driver functionality. Their argument was that microprocessor systems are fast enough these days and reliability stands above speed.

b) Domain Specific Code Generation for Linux Device Driver

In this work [5], they use a domain specific language to auto generate Device driver code. Details like the class of driver (video, audio, network, disk), category (char, bus, network) and the bus it uses (PCI, platform) can be specified and the tool would generate a template device driver. The domain specification language is an XML like language.

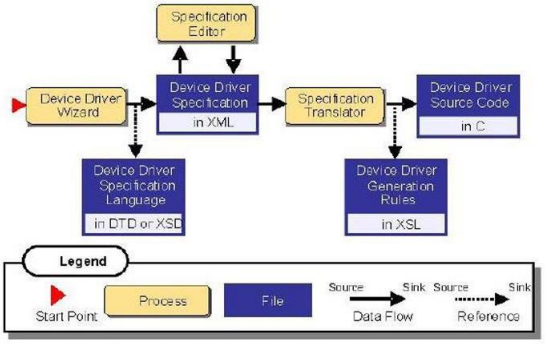

Figure 2: Steps involved in Device Driver Generation [5]

The framework ensures that the device driver code it generatesmakes calls to the kernel that pass the right set of arguments and have the high degree of correctness with respect to use of kernel API's. The tool does not write the entire driver but helps programmer get through fairly easily with all the necessary things required by the kernel from a device driver and lets him focus on the actual device driver core functionality. This tool does not solve any of the problems mentioned in the previous section directly. It is meant to solely reduce the grievances faced by an inexperienced programmer in figuring out many Linux driver specific details even before getting to the actual register programming. In Linux there are many driver frameworks, for example all video devices including radio are supported by V4L2 (Video for Linux 2), audio devices fall under ALSA (Advanced Linux Sound Architecture). Writing a video driver involves studying the V4L2 subsystem and implementing several callbacks.

Based on the bus technology used by the driver, several bus specific subsystems of Linux need to be made us of (PCI, USB, platform etc.). Putting together all of these to get a working driver can be tedious task. Many times, the examples available may not be as appropriate, for example

we may have the example of a video driver of a PCI based device but what if we need to write a USB based video device. This stub code generated by this tool can be thought of as a very good start point for driver programming. So, in an indirect sense, this tool helps to bring some form of reliability in the written driver code by auto generating some of the code hence making the generated code reliable and robust.

c) Nooks

In the Nooks architecture, they try to isolate the Linux kernel drivers into running in a 3rd domain of protection. The first domain of protection is the user mode which is an unprivileged mode. The second mode is the kernel mode which is a privileged mode. In case of Nooks, the drivers are run in a 3rd mode of protection which is also privileged mode, but provides MMU protection from kernel data structures.

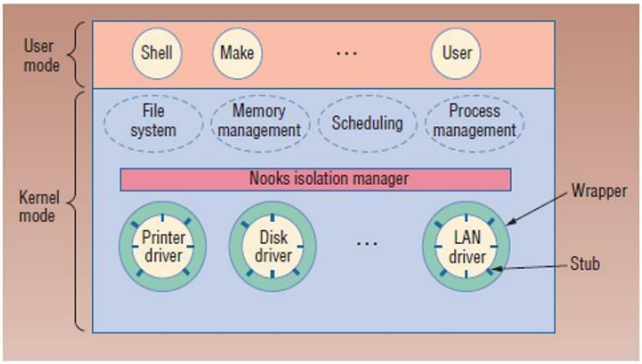

Figure 3: The Architecture of Nooks [4]

The mechanism used to implement the 3rd domain of protection is switching page tables whenever driver code is executed. Hence a copy of the kernel page table is maintained for every driver with the kernel pages marked as read-only. It is difficult to make a clear distinction between kernel mode and driver mode in the kernel as it used to be just kernel mode earlier. To make this possible, the Nooks architecture had to wrap and interposition several kernel-to driver functions and driver to kernel functions. So every time a kernel-driver driver-kernel transition was made, page tables were switched. To make copying safe between driver to kernel, newly introduced functions had to be made use of as direct copying was not possible due to the read only attribute for kernel data in the page table entries of the driver. Because of the x86 architecture limitation, they had to flush the TLB's every timea kernel-driver or driver kernel transition was made. Driver memory faults were caught by the Nooks isolation manager and restarted as required. Some form of recovery was supported. The Nooks wrapper API's would log all call and argument information onto a logging buffer. This information would be used to restart the driver. A full restart would reload the driver completely i.e. the driver would be reloaded like it is loaded for the first time. Rollback is when the driver's datastructures are preserved in recoverable virtual memory. The advantage of the Nooks architecture is that it executed the driver in kernel mode itself. It also provides memory protection (solution to **core issue Memory Faults**).

d) Device driver recovery though use of persistent memory

In this work [8] they have built a recovery framework for drivers on ARCOS a multiserver Operating System that is build upon the L4 kernel. The driver stores all of its significant state bearing variables into a persistent memory. To do this the driver only needs to declare the variable with a special compiler directive that would put those variables into a specialsection in the elf.

#define IS_PERSISTENT __attribute__((section(".pdata")))

All variables that the driver author thinks must go into the persistent memory must be suffixed with this macro (IS_PERSISTENT) during declaration. The Operating System puts the .pdata section to a different area in memory and preserves this memory across restarts but not upon termination.

```
class DeviceDriver {
protected:
        virtual status_t Service(
        const L4_ThreadId_t& tid,
        const L4_Msg_t& msg,
        L4_Msg_t& retMsg) = 0;
public:
        virtual status_t Initialize() = 0;
        virtual status_t Exit() = 0;
        virtual status_t Recover();
        status_t MainLoop();
};
```

Figure 4: Device Driver Base Class [8]

The above figure depicts the driver model. All device drivers in ArcOS must derive from this base class and implement the virtual methods. When the driver crashes, it is restarted and its recovery method (Recover()) is invoked. This recovery method must be implemented by the driver writer and it must refer back to the significant state variables to bring back the driver to its previous state. The **core issue Recovery** is being addressed by this paper. The idea of having a persistent memory, to hold state for the drivers, is key to recovery for device drivers.

e) L4 Linux

In the L4 Linux Approach [9],[10],[11],[12] Linux is run as a server on L4. The Linux kernel is run as a L4 user space process called the Linux server. The L4 Linux Server is a single L4 Thread. Upon booting, the Linux server requests memory from its underlying pager. Usually all of the physical memory available to the Linux personality is mapped to the Linux server. The actual hardware page tables are kept within L4 kernel (for security reasons). The Linux L4 server is responsible for handling all the Linux activities like handling system calls, handling page faults. In L4 the interrupts notification is done through IPC and a user level thread does this top half handling. In L4 Linux, every interrupt line is given an individual thread for top half processing. Bottom half processing of an interrupt, if made use of, is done with another thread. Every user process is implemented as a regular L4 task. The Linux server creates these tasks and also takes the role of being its pager. If a

page fault occurs in a user process (created by the Linux Server) is reported to the Linux Server through an IPC. All system calls are changed to IPC calls to Linux Server. This is accomplished by a modified version of libc (libc.so and libc.a). A user level exception handler called trampoline is created which emulates the native system-call trap. Signals in native Linux work by modification of the user mode stack, stack pointer and program counter to make the process believe that it has made a call to the signal handler just before it was scheduled out the last time. However, in L4 Linux, an additional signal handler thread was added to each Linux process, to achieve signal handling. All Linux threads are scheduled by the L4's internal scheduler. The priorities ofall threads are as follows (lowest to highest).

Linux Process < Linux Server < Bottom Half Thread The L4 Linux approach takes Linux and runs it over a multiserver operating system. In running Linux this way, we benefitted from all the advantages of a multiserver operating system. All the issues that were specific to the kernel mode aretaken away in this approach, however the core issues that existin the underlying L4 multiserver OS, like performance overhead of the IPC mechanisms continue to exist. The L4 Linux is ported to the x86 architecture only, thus restricting itsusability.

f) CuriOS

In CuriOS [15], the concept of persistence is extended to arguing that persistence must be maintained on a per client basis for drivers that serve multiple clients and in general forsystem servers. In case the driver fails and then restarts, it would then refer to the client's persistent memory to recover state information and resume service. The persistence of this memory, in this case, is maintained in client's address space. This way even if the server goes down, no special steps need to be taken to preserve the client related state is in the client's address space as the client is still active and running. If the client goes down, then it makes sense to discard the memory as well. CuriOS is comprised of various objects interacting with each other, such an object is known as a Protected Object (PO). All methods on a protected object are executed with reduced privileges and also memory protected through hardware so that code in a PO cannot corrupt other code. In many ways a PO is analogous to a server in microkernel. EachPO has its own private stack and heap space. A Server State Region (SSR) is used to store an OS server's client related information. An SSR is created whenever a client established a connection with a server. Memory for the SSR is acquired from the client's address space. SSR's are memory protected from the client and only the server is allowed to write into it.

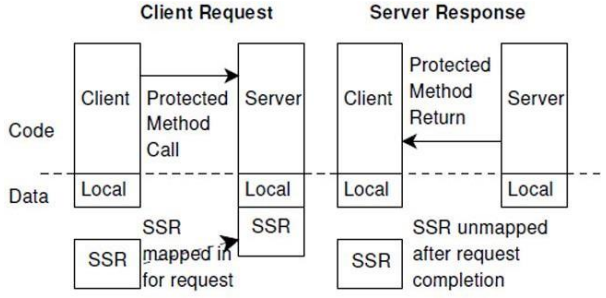

Figure 5: SSR based persistent memory in CuriOS [15]

SSR are managed by a SSR Manager singleton object. Upon restart, a server can query the SSRManager to get informationon what clients it was servicing and get access to their SSR's.

Suppose an SSR is corrupted, then only the corresponding client's servicing would get affected and the server would continue to service other clients normally.

Periodic Timer Manager reads clients SSR's to restore services to clients after restart. The scheduler saves the client's controlblock in its SSR and uses this information to resume operation in case of restart. File Systems are also implemented as PO's and they maintain client states in respective SSR's. In case of a crash, information such as open files, read position can be retrieved from the SSR of the client to resume service as last recorded. Similarly, device drivers are also implemented as PO's and they too can store client specific state in SSR's and use them for recovery upon restart. The system was implemented on 96 Mhz OMAP 1610 processor. Faults were simulated through fault injection using a QEMU based fault injection tool. Types of faults induced were, data aborts and register bitflip. Faults were injected into the timer, scheduler, network and file system servers. Recovery was considered successful if CuriOS could schedule a new process and accessdisk. All memory related aborts were recovered, but a few register bitflips were detected but not recoverable, and certain others went undetected. The overhead of protected calls and SSR were recorded to be roughly 200 microseconds more thandirect calling (on a 96MHz OMAP 1610 processor). There were overheads like flushing TLB's while switching between tables. Overheads in recover, getting information about all SSR's from SSR Manager. This system provides solution to **core issue Recovery**. The concept of persistent memory being maintained in the client's address space is excellent and can be generally applied to any system.

g) Exploring Kernel Lockups

This paper [13] discusses the **core issue Resource Hoarding** and how Linux detects soft-lockup. In Linux soft-lockup is detected with the help of a low priority thread that updates a time stamp every second. This time stamp is checked by the timer interrupt thread (to see if it was updated) every 10 seconds. In Linux most errors that are encountered in the kernel mode are handled by terminating the thread. However, an OOPS occurs in interrupt mode it is considered serious and the system turns unusable. Kernel code also calls panic on detecting serious errors. The soft lockup detector cannot detect lockups that occur in ISR's. This work implements a hardware watchdog timer-based system to detect lockups and do a soft reboot of the system. The soft reboot done here is different is from regular soft reboot of systems. The argument presented here is that if one driver causes a fault that must reboot the system, why should all the other innocent processes of the system be restarted, instead only the culprit thread is killed and restarted leaving all the other threads unaffected. The mechanism is described below. They added a new kernel thread that wakes up periodically and pats the watchdog timer. If this thread is not scheduled periodically, the processor is reset. The bootloader was patched so that

restarts caused by the watchdog take an alternate route. The Linux bootup code was also modified to support his operation. The alternate startup includes turning on the MMU, terminating the task that caused the lockup, re-enabling the watchdog and peripheral interrupts. The code is then made to enter the idle loop which then starts to schedule the runnable threads. Lock and semaphore tracking mechanisms need to be implemented in par with this technique to increase the chances of better recovery. On the choices Operating system [14] too, a hardware watchdog was implemented. If a hard lockup occurs, the watchdog bites. Like the Linux implementation, the Choices too had a different recovery startup code to deal with watchdog resets. The recovery routine pretends to be idle thread, switches the MMU ON and restores interrupts. It then pretends to be the locked up thread and calls die() directly to terminate the thread. Killing of this thread restores the next runnable thread in the system. The implementation provides a way of detecting lockups. If a hardware error causes a lockup, that too would be detected by this system and hence we could say that it provides a solution to **core issues Recovery** and **Hardware Errors**.

Choices Operating System

The Choices operating system is a research operating system developed in C++. It has some very innovative features for robustness which are described in the following sections.

1) Exception Handling in Kernel

We discuss some robust features of the Choices OS [14]. Usually when a processor encounters a processor exception,it fail-stops the program except for the cases when the processor exceptions generate a page fault and the virtual memory finds the page in swap space. The Choices OS makes processor exceptions available as C++ exceptions. This gives a chance to the program to react to such issues instead of ending in a fail-stop. In choices OS every interrupt in the system passes through the interrupt manager. Processor exceptions too are delivered to the same Interrupt manager. Hence the interrupt manager checks to see if the interrupt is a processor exception. If yes then it creates an exception object and stores the context and stack trace into the exception object. The handler then changes the PC to the interrupted process's ThrowException Function. After this the regular C++ exception handling takes over. This system was implemented only on the ARM port of the Choices Operating System and for this port 3 types of exception could be thrown.

1) ArmDataAccessException
2) ArmInstructionAccessException
3) ArmUndefinedException

An additional advantage brought in by the exception handling facility of C++ is the stack unwinding and callingof destructors by the C++ exception handling code. This in effect prevents memory leaks making it a solution to **core issue** of **Resource Leak**.

2) Code Reloading

The code-reloading feature of the choices Operating System helps prevent prefetch abort. It does so by periodically checking the CRC of kernel critical code and in case of mismatch, it is reloaded from the disk. This feature in specific tries to reduce the occurrence of **core issue Memory Faults**.

a) VINO

The VINO [16] kernel makes use of a Software Fault Isolation technique called MISFit [17] to provide protection to drivers and extensions. The MISFit technique involves passing code through a post compilation tool that inserts assembly code that converts an invalid address to a valid range. It does this by masking out the upper few bits of an address with a fixed value. Valid address would remain unchanged after this step whereas invalid addresses would get changed into a valid range. For function pointer-based calls and C++ Virtual functions, some OS support is taken from the kernel and a hash table of function pointers is searched through to see if the address is that of a valid function or method. The VINO kernel allows extension in the form of grafts. Grafts may either replace an existing function in the kernel to provide alternate functionality or it may be added to a list of handlers that associate with a given kernel service. Every graft invocation is wrapped into a transaction that is managed by the transaction manager. The transaction mechanism involves pushing of an undo operation into an undo call stack. If the transaction aborts, the undo operation is invoked. At the end of a successful transaction the undo operation is popped back fromthe undo call stack. To prevent resource holding the VINO kernel introduces time constrained locking mechanism, hence every lock has an associated timeout mechanism. However, the most appropriate timeout to be associated with a lock must be experimentally verified. The MISFit software Fault isolation technique is solution core issue **Memory Faults** with a difference that it does not make use of MMU hardware to provide address space protection. The transaction mechanism provides immediate recovery from unsuccessful transactions by invoking the undo functions in the undo call stack thus providing some form of recovery. The timed lock makes sure that no thread can hold on to a resource for more than the stipulated timeout period hence providing a solution to **core issue** of **Resource Hoarding**.

b) Shadow Driver

This work [19] introduces a concept called shadow driver framework to support device driver recovery. It is built on topof the Nooks architecture. Device drivers often fall under different classes. Drivers falling under the same class have similar kernel-programming interfaces. Hence the recovery once the failed driver is recovered, the taps are re enabled and the shadow driver goes back into passive mode. The driver however is not run from the start as some things like kernel registration are preserved by the shadow drivers. This is done through the wrapper calls of Nooks which can record the callsthe driver makes to the kernel along with the arguments.

However, tasks like enabling interrupts, remapping I/O memory are re-performed. The exact steps of recovery would depend on the class of the driver. This work is a clear exampleof device driver **Recovery core issue**. However, the recovery technique presented here is closer to being stateless. Some parts of the state are being captured in the form of arguments to the calls the driver would make to the kernel. But the actualdriver's internal state variables are not preserved. This model is an improvement over the Nooks

driver recovery as it takes into account the possibility of driver's clients making calls at the time the driver is down and in the process of restarting.

c) Exceptional Kernel. Using C++ exceptions in the Linux kernel

In this work [21], a new C++ runtime support was added to the Linux kernel. The runtime was initially derived from the user mode, but enhanced later for performance. The implementation was done by purely implementing the ABI calls of the compiler (for example the throw operator translates to ABI call __cxa_allocate_exception followed by strategy presented here is implemented on a per class basis, cxa_throw), hence eliminating the need for a special i.e. there is one shadow driver per class of drivers. But there may be several instances of it running for each driver. Thus having a few shadow drivers, one per class, is sufficient and driver developers need not worry about developing them. Shadow drivers can only recover from transient and fail-stop driver failures. This is because it makes use of the Nooks system which could only detect these two types of failures andrestart the driver. The shadow driver executes in two modes, passive and active. The implementers created a tap system where calls from kernel to driver and vice versa are also redirected to the shadow driver. All of this call redirection is managed by the shadow manager. The shadow manager also receives notification from the fault-isolation subsystem when a driver fails. At that time, it switches the shadow driver from passive to active mode. In passive mode driver records several kinds of information which differs based on the class of the driver. Generally, in passive mode driver records the registration with specific subsystems of the kernel, interrupt lines acquired by the driver. The shadow driver however does not maintain any persistent state of the drivers, it expects the clients to do so. The active mode is triggered when shadow manager informs the shadow driver about the failed driver andasks it to switch to active mode. In active mode the shadow driver disabled the interrupt of the hardware so that the hardware does not continue to fire interrupts during recovery (may not work with shared interrupt lines). The I/O mappings of the hardware are also removed, to prevent any DMA into kernel memory. For most application calls during recovery, the driver may give a busy error code as most applications may be designed to handle it. For certain other applications blocking the call until the driver recovers is a better solution.

The GNU G++ compiler generates a crtbegin.o andcrtend.o ELF's in addition to compiled code object which it uses to invoke the C++ runtime and call global constructors and destructors. In the EFL section, these appear as .init and .fini sections. In this implementation, the Linix kernel module loader was modified to call these sections along with the calls to module initialization and module exit codes. The ABI's implemented for the C++ runtime constituted about 7000 LOC in length. With respect to their kernel configuration, this caused an image size increase of 2%. Their analysis of user side ABI revealed that 93% of the total ABI execution time was devoted to unwinding of the stack. This was because it was a two stage process in the user side ABI. The first optimization was done in making the 2 stage unwinding of the G++ ABI as one step. This brought down the execution time from an earlier 12.7µs to 6µs. The evaluations further revealed that the cost of throwing exceptions increased as the number of stack frames increased. The overall effect of having C++ code in kernel (especially for drivers) mean that code could be written with exception handlers. This provides robustness in a more generic way meaning the code gets a chance to correct itself. So in a general sense this could be categorized as a solution to the **core issue Recovery**. The recovery is on a smaller level with one catch block serving a recovery routine for several code blocks and several such catch blocks forming the total recovery code.

d) D-Bus Based User Device Driver Framework Design for Linux Mobile Software Platform

This work [22] explores the feasibility of having user mode device drivers in Linux. They implemented several device drivers on Linux running over a mobile phone hardware. They made use of the UIO framework to implement the drivers. In addition, they made use of udev subsystem to give the UIO based driver device nodes a more logical name. The UIO kernel driver creates device node entries at the time of registration in the form of /dev/uiox where x represents an incrementing minor number. This number depends on the order in which the device nodes are created. The HAL daemon, through the udev notification then creates symbolic links for each UIO user mode driver with a more logical name, like /dev/uio/leds for an LED driver. This paper serves a proof of concept that user mode device drivers are feasible in Linux. The UIO framework provides solution to **core issue Memory Faults** and **Added Privileges**.

e) Safe Device Driver Model Based on Kernel-Mode JVM

This paper [23] explores and evaluates the idea of running a Kernel Mode JVM and running device drivers under it. The kernel mode JVM was a modified version of the Tiny-VM, an open source JVM which can run on an OS less Microcontroller. The kernel-mode JVM incorporated Device Driver Interface (DDI) to the Tiny-VM. The DDI manages type conversions between C and java and vice versa. Inside the JVM an object consists of two parts, Object Head and Object Body. JVM needs to read information off the Object Head to access the Object. In typical JVM's these are in contiguous memory locations. But in the Kernel mode JVM, since it needs to data access to kernel, it manages this by maintaining a Object head in its local memory and a pointer tothe actual object (which could be a kernel data structure). This also simplifies the memory management as the JVM now needs to maintain a list of Object heads with pointer references. This makes the structure of identical sizes which is maintained in the form of a Link list in the JVM. The actual object body could be preexisting (as a kernel data structure) or else it is allocated using the regular kmalloc and kfree kernel calls. Unlike regular JVM's this JVM does not include a Garbage Collector since the semantics for garbage collection in user-mode do not apply to kernel mode. As a result, a free_object API is defined to free object and drivers would explicitly call them when they need to free objects. Driver ISR's are also written in Java. The JDD ISR also runs in borrowed context like in the regular case. Like ISR's implemented in C in the Linux kernel, even JDD ISR is not permitted to

sleep. This is checked by the JVM at the time of invocation. If called within Interrupt context, and a sleep is encountered, an exception is thrown. For accessing register memory (something very often required by a driver), a JNI call is made. Writing device drivers in Java sure provides isolation and provides the driver with the language provided safety, but the performance penalty is quite high. The ISR's in particular are expected to have very low processing time and a Java ISR cannot guarantee that. The system provides solutions to the **core issues Memory Faults** and **Added Privileges**.

f) Creating User-Mode Device Drivers with Proxy

In this paper, user-mode driver for Windows NT is developed using a Proxy driver. The proxy driver makes no changes to the NT Operating system. In Windows NT, I/O is packet driver. Once a driver I/O call(from application) enters the NT executive, individual I/O requests are encoded into an I/O Request Packet. IRP's could pass through multiple drivers. (Example: file System driver to disk driver and back).

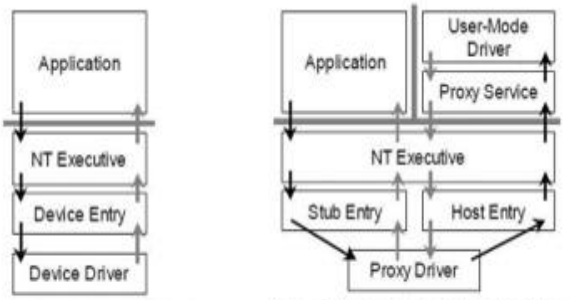

Figure 1. Composition of a Kernel-Mode Driver.

Figure 2. Composition of a User-Mode Driver. The kernel-mode proxy driver passes IRPs to the User-Mode driver through a host device entry and a COM service.

Figure 6: Proxy Driver [28]

IRP's are passed to the corresponding driver through the Device Entry. In this system the user mode drivers are implemented using a kernel-mode proxy driver. The user-mode driver connects to the proxy through a special device entry called the host entry. The user-mode driver registers with the proxy driver informing it about the I/O requests it would process. In response to this registration, the proxy driver creates a stub entry. All access to the user-mode driver is through this stub-entry. Whenever an application makes an IRP request, it is routed through the stub-entry to the proxy driver. The IRP then gets forwarded to the Host Entry and then further to the user mode device driver, through the proxy service. The evaluations suggested that proxy based user drivers had a 50% performance overhead for the calls that made data transfers between the user and the kernel. Calls bearing no data transfer may not suffer from this overhead as much. The technique presented here is very interesting as the user application remains unmodified as it still thinks that it is calling the kernel mode driver. Having the real driver in user mode brings in the automatic advantages of user space protection (**core issues Memory Faults** and **Added Privileges**).

4. Conclusion

In all the techniques seen so far, we have seen that each operating system has its own approach to the core issues that impact robustness. Overall, from the protection and robustnesspoint of view the microkernels have shown better robustness because of the system software components being executed in user mode. Minix 3 in its current from implements all drivers and other system software components in user mode. It has provided solution to most core issues. The POSIX compatibility makes application porting easier. The only unaddressed problem is the performance overhead of the IPC mechanisms. As CPU speeds keep increasing this problem is becoming less important as compared to the reliability aspects. L4 Linux patch makes Linux very reliable by putting the entire Linux OS into user mode. The L4 Linux patch is being actively being maintained by the Operating Systems Group at TU Dresden. Nooks and shadow drivers are excellent pieces of demonstrating that driver in kernel mode too can be robust can recover from fault. These patches are not being actively maintained and were last seen working on 2.4 versions of Linux. The group at University of Illinois at Urbana-Champaign [13] work on recovering from Linux lockups with a modified startup is also very interesting and is available for download from their site. This idea of user mode drivers generally accepted by the Linux community as well. For example, the X Server for example in Linux has its drivers written in user mode. This also allows the graphic card manufacturers to supply their drivers as binaries. There is a framework for writing drivers in user mode in Linux that is known as UIO (Universal Fieldbus and Industrial I/O Framework). UIO was developed by OSADL (Open Source Automation Development Lab) and is now available as config option under the Linux kernel source tree. The Window Server and File Server of Symbian OS [28] are written in user mode. Even in Windows operating system a user mode driver programming framework has been introduced with Windows Vista [29]. In general, we see that the focus is generally shifting from functionality alone to functionality and robustness. So not only is a piece of system software expected to work right but work reliably as well. Because of the pace at which newer hardware arrives, it is almost impossible to ensure perfect testing of newly written drivers and hence the robustness support must be enforced by the operating system itself. In the era when only monolithic operating system existed, the microkernels introduced the concept of having a small kernel providing just the core kernel services and having the rest of system services to be implemented as user mode servers. This definition is sometimes referred to as a multiserver operating system. The microkernels have made their point as we can see that the initially believed monolithic kernels like Linux are having more and more microkernel like features.

References

[1] Jacob B, Mudge T, "Virtual Memory:Issues of Implementation", IEEE Computer Society journal,Volume: 31 , Issue: 6, Publication Year: 1998, Page(s): 33 - 43.
[2] Khalidi Y.A., Talluri M, Nelson M.N., Williams, D, "Virtual memory support for multiple page sizes", Proceedings of the Fourth Workshop on Workstation Operating Systems, 1993.

[3] Herder J.N, Bos H, Gras B, Homburg P, Tanenbaum A.S, "Fault isolation for device drivers", Dependable Systems & Networks, 2009.
[4] Tanenbaum A.S, Herder J.N, Bos H, "Can we make operating systems reliable and secure?", IEEE Computer Society Journal Volume: 39 , Issue: 5, Publication Year: 2006 , Page(s): 44 - 51.
[5] Park J.C, Choi Y.H, Kim T.H, "Domain Specific Code Generation For Linux Device Driver", Advanced Communication Technology, 2008. ICACT 2008.
[6] Swift M, Martin S, Levy H.M, Edders S.J, "Nooks: an architecture forreliable device drivers", Proceedings of the 10th workshop on ACM SIGOPS European workshop, Year of Publication: 2002.
[7] Hunt G.C., "Creating User-Mode Device Drivers with a Proxy", Proceedings of the USENIX Windows NT Workshop, Seattle, Washington, August 1997.
[8] Ishikawa H, Courbot A,Nakajima T, "A Framework for Self-Healing Device Drivers", Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self Organizing Systems, Year of Publication: 2008.
[9] Häig H, Hohmuth M, Wolter J, "Taming Linux",Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems, Year of Publication: 1998.
[10] Häi H, Hohmuth M, Wolter J,L iedtke J, Schörg S S, "The performance of µ-kernel-based systems", Proceedings of the sixteenth ACM symposium on Operating systems principles, Year of Publication: 1997.
[11] http://os.inf.tu-dresden.de/papers_ps/adam-diplom.pdf
[12] http://os.inf.tu-dresden.de/papers_ps/adam-beleg.pdf
[13] David F.M, Carlyle J.C, Campbell R.H, "Exploring Recovery from Operating System Lockups", USENIX Annual Technical Conference (USENIX.07), June, 2007.
[14] David F.M, Carlyle J.C, Campbell R.H, "Building a Self-Healing Operating System", 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing, September, 2007.
[15] David F.M, Chan, E.M, Carlyle J.C, Campbell R.H, "CuriOS: Improving Reliability through Operating System Structure", USENIX Symposium on Operating Systems Design and Implementation, December, 2008".
[16] Seltzer M.I, Endo Y, Small C, SmithK.A,"Dealing with disaster: surviving misbehaved kernel extensions" - Proceedings of the second USENIX symposium on Operating systems design and implementation1996.
[17] Small C, Seltzer M, "MiSFIT", Third Conference on Object-Oriented Technologies and Systems (COOTS .97).
[18] Chou A, Yang J, Chelf B, Hallem S, Engler D, "An Empirical Study of Operating Systems Errors", Proceedings of the eighteenth ACM symposium on Operating systems principles.
[19] Swift M.M, Annamalai M, Bershad B.N, Levy H.M, "Recovering Device Drivers". In Symposium on Operating Systems Design andImplementation (2004).
[20] David F.M, Carlyle J.C, Chan E.M, Raila D.K, Campbell R.H, "Exception Handling in the Choices Operating System", Advanced Topics in Exception Handling Techniques , Vol. 4119 Springer (2006).
[21] Gylfason H, "Exceptional Kernel. Using C++ exceptions in the Linux kernel", Department of Computer Science. ReykjavÃ University.
[22] Cho Y.J, Cho Y.C, Jeon J.W,"D-Bus based user device driver framework design for Linux mobile software platform", ISIE 2009 IEEE International Symposium on Digital Industrial Electronics, 2009, Publication Year 2009.
[23] Chen S, Zhou L, Ying R, Ge Y, "Safe device driver model based on kernel-mode JVM" Virtualization Technology in Distributed Computing archive Proceedings of the 2nd international workshop on Virtualization technology in distributed computing (Year of Publication: 2007).
[24] Bovet D.P, Cesati M, "Understanding the Linux Kernel 3rd Edition".
[25] Sloss A.N, Symes D, Wright C, "ARM System Developer's Guide".
[26] Beck M, Bohme H, Dziadzka M, Kunitz U, Magnus R, Verworner, "Linux Kernel Internals 2nd Edition".
[27] Stallings W, "Operating Systems internals and Design Principles 6thEdition".
[28] Sales J, "Symbian OS Internals, Real-time Kernel Programming".
[29] http://channel9.msdn.com/blogs/charles/peter-wieland-user-mode- driver-framework.