Hybrid Device Drivers

Chaitanya Soma Shubham Mudgal Gayatri Mestry

Department of Computer Science
University of Colorado, Boulder
CO 80303 USA
{chaitanya.soma,shubham.mudgal,gayatri.mestry}@colorado.edu

***Abstract*—This document describes the proof of concept of hybrid device drivers. The working of a traditional kernel driver is split into user and kernel space. The communication between the two spaces takes place through LPC stubs. Mutex and semaphores are implemented for synchronizing data structures between the two spaces. The hybrid driver is evaluated by benchmarking it with the traditional kernel driver, with the maximum workload in user space, then splitting the workload in user and kernel space and finally with the workload in a user space application. The bulk of this paper is focused on implementation of communication between user and kernel space drivers that help in achieving fault isolation.**

Keywords—fault isolation; splitting drivers

I. INTRODUCTION

Device Drivers are extensions that communicate with I/O devices and the kernel. Their performance is critical for processing IO quickly and thus, the drivers reside inside the kernel address space. While the use of commodity operating systems is becoming ubiquitous in most modern embedded systems, it becomes crucial to address the problem of kernel crashes due to third party extensions like device drivers. Since these commodity operating systems have monolithic architecture a fault in the kernel extensions may corrupt vital kernel data structures, causing the system to crash. Thus, it becomes important to implement fault isolation mechanisms so that even if there are bugs in the driver functionality the user mode process fails to protect the kernel from crashing.

One of the goals of this project is to identify the code in kernel device driver which can be ported to user mode. We try to leave all the performance critical code write writing or reading data physically to the device in the kernel itself. There is a lot of code in a device driver which is not called very often like initialization, registration and diagnostic information which can be moved to the user space.

The report is structured by first getting into the details of splitting process of the driver. Then the implementations in user space driver and kernel space driver to build the hybrid driver is described. Further synchronization method using mutex and semaphores is explained. Lastly the performance evaluation of the hybrid driver is given in detail.

II. SPLITTING DEVICE DRIVER

A. Refactoring Device Driver functions

One of the basic concepts involved in this project is splitting the driver code into kernel space and user space. Therefore it is important to assess the code and figure out the parts of the driver to be sent and executed in the user space.

The portions of the code which needs to reside in kernel and user space by the use of call-graphs have been identified. The device driver is analyzed and the determination of how functions implemented in the driver are split between kernel-mode and user-mode is done by building a static call-graph of the driver and identifying the critical root functions.

The driver code is profiled using Callgrind, which is a tool of the Valgrind toolchain. When the Callgrind is used to profile the application, it is transformed in an intermediate language and then run in a virtual processor emulated by valgrind. This has run-time overhead, but the precision is good and the profiling data is complete. Kcachegrind is also used which is a GUI application which uses data generated by the Callgrind profiler and generates a graph showing the list of functions sorted according exclusive or inclusive cost metrics, and optionally grouped by source file. It also shows the number of times a function is being called and the time taken by each function w.r.t the whole program/the caller function.

B. Interpreting results from Kcachegrind

The GUI tool displays all the functions called within the program that is being profiled. The two cost attributes involved in it are ‘incl’ and ‘self’. ‘Self’ attribute gives the cost in terms of time taken by the function itself whereas ‘Incl’ shows the cost of the selected function including the cost of all the other functions called by the this function. This is an important distinction as it gives us the details on how time is being split within a function. ‘Called’ attribute gives the information on the number of times the particular function is called from its parent function. Although in the callgraph generated, we can get overall detail on which function is called how many times and from where in that cycle.

Critical functions are inferred by analysing the graph flow with edge and node weights. While node weights are the number of lines in the code, edge weights denote the call frequencies of the particular function. Also, it is important to

notice that the critical section is not split between kernel and user space.

To explain how it works in simple terms, look at the example of the producer-consumer thread performing simple tasks in the user space of the driver:

In figure 1, is shown a part of the whole callgraph. The user defined functions, like producer_thread, consumer_thread, task1, etc, are enclosed in the red square. The graph is focused on the function 'start_thread' and shows the flow and cost of functions that are called along with the number of times they are called and their parent functions. The graph skips all the functions which have negligible impact in the execution. As you can also see that it shows the list of all the inbuilt functions that are called within the execution process which take up most of the real execution time.

Figure 2 shows the flat profile of the executed program. Starting from the top, it gives the total inclusion and self cost of each function involved, the number of times it is called and the caller functions.

III. HYBRID DEVICE DRIVER

Bulk of this paper is focused on the communication and synchronization between the user and kernel parts of the device driver. The kernel driver is very similar to the traditional device

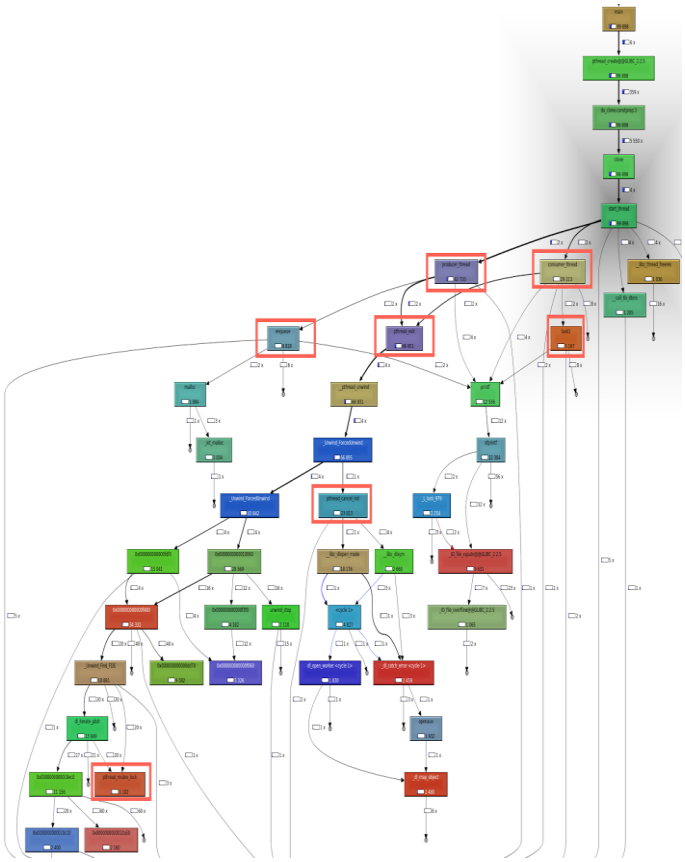

Figure 1. Call graph from Kcachegrind

Incl.	Self	Called	Function	Location
67.89	0.01	(0)	0x000000000000012d0	ld-2.19.so
40.94	0.26	1	_dl_start	ld-2.19.so: rtld.c, get-dynamic-i...
40.67	0.12	1	_dl_sysdep_start	ld-2.19.so: dl-sysdep.c, dl-sysd...
40.53	0.40	1	dl_main	ld-2.19.so: rtld.c, get-dynamic-i...
35.64	10.03	154	_dl_lookup_symbol_x	ld-2.19.so: dl-lookup.c
32.58	9.67	6	_dl_relocate_object	ld-2.19.so: dl-reloc.c, dl-machin...
32.10	0.11	4	start_thread	libpthread-2.19.so: pthread_cre...
25.62	14.81	161	do_lookup_x	ld-2.19.so: dl-lookup.c
24.73	0.01	1	0x0000000000400b50	user
24.39	0.02	1	(below main)	libc-2.19.so: libc-start.c
24.32	0.10	1	main	user
21.55	0.32	4	pthread_create@@GLIBC_2...	libpthread-2.19.so: pthread_cre...
21.51	0.01	4	pthread_exit	libpthread-2.19.so: pthread_exi...
21.50	0.01	4	__pthread_unwind	libpthread-2.19.so: unwind.c
21.49	0.01	4	_Unwind_ForcedUnwind	libpthread-2.19.so: unwind-forc...
20.15	0.04	2	producer_thread	user
19.57	0.03	4	_dl_allocate_tls	ld-2.19.so: dl-tls.c
19.40	0.14	6	calloc	libc-2.19.so: malloc.c
18.91	0.00	1	malloc_hook_ini	libc-2.19.so: hooks.c, arena.c
18.52	0.07	1	ptmalloc_init.part.7	libc-2.19.so: arena.c
18.44	18.01	1	_dl_addr	libc-2.19.so: dl-addr.c
14.09	0.07	4	_Unwind_ForcedUnwind	libgcc_s.so.1
13.62	0.30	44	_dl_runtime_resolve	ld-2.19.so: dl-trampoline.S
13.32	1.23	44	_dl_fixup	ld-2.19.so: dl-runtime.c, dl-mac...
13.23	1.08	7	<cycle 1>	ld-2.19.so
11.03	1.41	20	0x000000000000f480	libgcc_s.so.1
9.39	0.13	2	consumer_thread	user
9.18	0.13	4	0x0000000000010060	libgcc_s.so.1
8.42	2.42	142	check_match.9458	ld-2.19.so: dl-lookup.c
8.19	1.53	680	strcmp	ld-2.19.so: strcmp.S
7.89	0.47	9	_dl_map_object	ld-2.19.so: dl-load.c
7.39	0.02	1	pthread_cancel_init	libpthread-2.19.so: unwind-forc...
6.74	0.14	11	_dl_catch_error <cycle 1>	ld-2.19.so: dl-error.c
6.67	6.67	2 898	strcmp'2	ld-2.19.so: strcmp.S
6.07	0.36	20	_Unwind_Find_FDE	libgcc_s.so.1
5.84	0.01	1	__libc_dlopen_mode	libc-2.19.so: dl-libc.c
5.30	0.04	7	openaux	ld-2.19.so: dl-deps.c
5.02	0.08	1	dl_open_worker <cycle 1>	ld-2.19.so: dl-open.c

callgrind.out.8063 [1] - Total Cycle Estimation Cost: 311 245

Figure 2. Flat profile of user program

driver except this one has additional functions built in which will allow it to communicate to the user driver. The user driver is a daemon which runs in the user space and can communicate with the kernel driver. The kernel driver exposes some functions to the user driver and vice versa. Our communication mechanism allows the kernel and user drivers to execute functions exposed by each other.

We implement local procedure calls to call kernel functions from the user space and user space functions from the kernel. In our current implementation, one procedure call can call a function in user space which can intron call a function in kernel space and so on before the procedure returns. The fundamental idea is to develop a communication mechanism similar to Remote procedure calls. The two main elements in this system are user drivers and kernel drivers. Special IOCTL calls have been built into the kernel driver to support upcall's and downcall's as necessary.

A. *User Driver*

The user space driver contains the communication mechanisms to implement Local Procedure calls in kernel and also have procedures which pertain to the specific device driver. During initialization the user driver first creates a Request Queue which is a FIFO and stores all the procedure call requests received from the kernel driver. In the initialization process the user driver also creates a thread pool of worker threads and a leader thread. The leader thread blocks in the kernel whereas the worker threads wait sleep on a semaphore for the Request Queue and execute the requested procedures when data is available in the queue. The user driver also creates an array of pointers for all procedures. This array of pointers is initialized with all the procedures which are exposed to the kernel driver. A procedure is translated to a function pointer index using a hash table.

The leader thread blocks in the kernel on POLL_LPC during initialization, when the kernel wants to request a procedure call in the user driver, the kernel returns the POLL_LPC with the function name and its arguments. The leader thread now enqueues this procedure request from the user in the Request Queue, increments the semaphore and goes back and blocks in the POLL_LPC ioctl call. The worker threads sleeping on the semaphore wake up and dequeue the element in the request queue. The worker thread uses a hash function to get the function pointer index of the requested procedure and executes the procedure. When the procedure is completed, the worker thread returns the return values of the procedure and indicates the completion to kernel using the RESPONSE_LPC ioctl call.

The user driver can also call a procedure in the kernel driver, the user driver does so by calling the kexec_lpc function which in turn calls the EXEC_LPC ioctl function. The user provides the function name and arguments as parameters encapsulated in a structure as parameters to this function. The user thread then enters the kernel, executes this procedure and passes the return values back in the same structure which was initially passed. The data structures can be serialized using the TPL library when passed to kernel ioctl calls.

The user driver can also place mutexes and semaphores on the kernel data structures, this will be further discussed in the synchronization section.

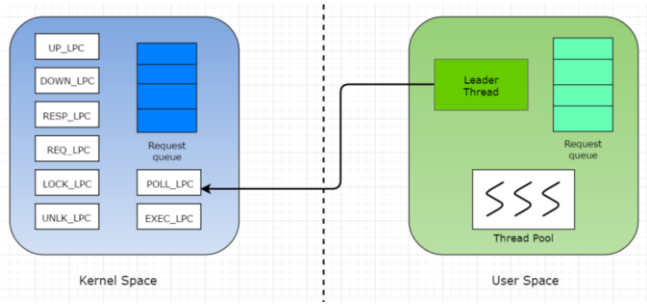

Figure 3. System Overview

B. *Kernel Driver*

The kernel driver contains all the driver function which are exposed to user applications. The kernel drivers also implements special ioctl function to enable communication with the user driver. The kernel driver has a Request Queue which is a FIFO and all request to access the user space functions are enqueued here. During initialization the user driver first creates a Request Queue, every time someone needs to run a procedure in the user driver, they enqueue a job in the request queue. The kernel driver also has an array of pointers for all functions, mutexes and semaphores exposed to the user driver, these can be indexed using a hash table. The kernel driver implements 8 ioctl functions to facilitate the communication mechanisms between user and kernel driver. For IOCTL calls implement the Local Procedure Calls between user and the kernel space and the rest implement the synchronization between mutex and semaphores.

POLL_LPC: The leader thread in the user driver block in this function, this is how we facilitate an upcall between the kernel driver and the user driver. The leader thread in the user driver calls POLL_LPC and blocks on this ioctl call, POLL_LPC sleeps on a semaphore on the Request Queue, when data is available in the queue, it dequeues the job, decrements the semaphore and returns to the user driver. We use copy to user function to send the job structure, which contains the function name and its arguments to the user driver.

EXEC_LPC: This ioctl call allows a user driver to execute a function in the kernel driver. The user driver calls this function and passes a struct which has the function name and its arguments. The EXEC_LPC uses a hash table to index the correct function pointer, executes the function and sends the return values in the same struct using copy to user function. This function is a blocking function.

RESPONSE_LPC: This function tells a kernel driver that a requested function in the user driver has finished execution and wakes up the waiting kernel thread. The kernel driver can call a procedure in the user driver, by enqueuing a job in the Request Queue. After enqueuing a job, the kernel thread which requested this procedure goes to sleep using a wait queue. The user driver calls RESPONSE_LPC when it finishes executing the procedure. This ioctl call wakes up the sleeping kernel thread and gives it the return values of the procedure. At this point the user thread continues execution.

REQUEST_LPC: This ioctl call will directly call a requested procedure without spending much time in the kernel driver, this has been primarily implemented for testing and performance evaluation purposes.

IV. Implementation

A. *Communication between User and Kernel Space*

In this section we will explain a typical communication session between the user and kernel with an example. Here the test application will invoke a function call on the driver, which will call the udaemon to execute a procedure for him, which in turn wants to execute a procedure in the kernel driver as show in the Figure 3.

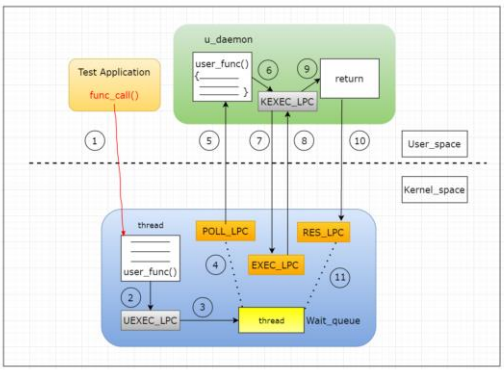

Figure 3. Communication

1. The Test application calls a driver function like write/read which is executed in the kernel driver
2. The kernel driver executes the write system call and now wants to execute a procedure in the user driver or udaemon
3. The kernel driver can do this by calling the uexec_lpc function, which enqueues the job request in the Request Queue in the kernel and puts the requesting thread to sleep on a wait queue and posts the semaphore.
4. The leader thread in udaemon which is sleeping on a semaphore for the Request Queue, wake up, dequeues the job from the request queue and returns the POLL_LPC function
5. The leader thread now goes back to the udaemon and enqueues the job received by the kernel driver in the Request queue in the udaemon
6. One of the worker thread in udaemon, dequeue this request and start executing the procedure.
7. When executing this procedure if the worker threads encounter a procedure in the kernel driver. They request the execution of this procedure using kexec_lpc function.
8. The kexec_lpc function calls the EXEC_LPC ioctl call and executes the procedure in the kernel driver. This is a blocking call and the function returns to the worker thread when the kernel driver finishes the execution of the procedure.
9. The worker thread finishes the execution of the rest of the procedure and performs an RESPONSE_LPC call to signal the kernel driver that the procedure has been completed
10. The RESPONSE_LPC call will wake up the respective kernel thread which is sleeping on the wait queue.
11. The kernel driver completes the execution of the remaining procedure and finally returns to the Test Application

B. Synchronization

Our communication mechanism allows the user driver to synchronize data structures in the kernel driver. The synchronization primitives supported are mutexes and semaphores. We use a static analysis tool to find all the shared data structures between the user and kernel driver and map every procedure with their dependent data structure. There is a one to one mapping between each data structure and its corresponding mutex. Now it's important to know beforehand about the functions and their dependent data structures. Hence, comes the concept of pointer analysis. Although it's not included in the implementation, its approach is included for future work for the analysis of data structure and its caller functions. The idea is similar to Andersen's pointer analysis where every pointer is mapped with the memory address it is pointing to. A graph is then generated that shows which pointer are alias and where they are pointing. The similar approach can be used for getting a pointer-graph for functions (since they are stored in a fixed place in the memory) and the data structure they are using. Therefore, instead of statically getting the information about which data structure to lock and send to the user space, it can be dynamically figured out to when and where to place the mutex locks for synchronization.

We have implemented four ioctl calls to facilitate synchronization between the user driver and kernel driver.

LOCK_LPC: This function is called by the user driver when it wants to place a lock in the kernel. The user daemon passes the name of the mutex as arguments to LOCK_LPC, the function then uses the hash table and to index into the array of mutex pointers and places a lock in the kernel space. LOCK_LPC then returns the data structure locked to the user driver by using a copy to user function. Now the user driver has successfully placed a lock in the kernel driver.

UNLOCK_LPC: The user driver calls UNLOCK_LPC when it wants remove a lock initially placed on the kernel. UNLOCK_LPC gets the mutex name and the data structures as arguments form the user driver, this function first uses the hash table to find the corresponding mutex, copies the data structure in the kernel with the data structure passed by the user driver and then unlocks the corresponding mutex.

Two more ioctl functions namely UP_LPC and DOWN_LPC have been implemented to synchronize semaphores between kernel driver and user driver. The implementation for UP_LPC is very similar to LOCK_LPC and the implementation for DOWN_LPC is similar to UNLOCK_LPC. UP_LPC is the equivalent of sem_post in pthreads and DOWN_LPC is the equivalent of sem_wait in pthreads.

C. Locking and Unlocking

Below is an example of sequence of execution to place a lock and unlock that lock in the kernel from the user driver.

1. First a user test process performs a driver call like open/write on the kernel
2. Now the write function in the kernel driver wants to execute a function which is in the user driver or user daemon
3. The kernel request the udaemon to execute this function
4. One of the threads in the udaemon starts executing the procedure requested by the kernel
5. This worker thread in udaemon wants to place a lock on the data structure in the kernel and does so by using the LOCK_LPC ioctl call
6. The LOCK_LPC call, will use the hash table to find the corresponding mutex and places a lock on that mutex in the kernel. It also sends the corresponding data structure to the user using the copy to user function

7. The worker thread in the udaemon has now effectively placed a lock in the kernel driver and has acquired the latest copy of the data structure from the kernel driver.
8. The worker thread now done executing on the data structure and wants to unlock the mutex in the kernel. It does so by using the UNLOCK_LPC ioctl call.
9. The UNLOCK_LPC call receives the mutex name and the data structure as arguments to the function.
10. The ioctl call update the data structure in the kernel driver with the data structure received from udaemon and performs a mutex unlock

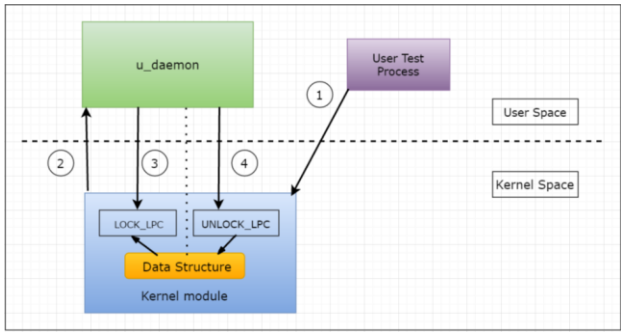

Figure 4. udaemon can place a mutex lock and unlock in the kernel driver. A worker thread in udaemon can do a sem_post or sem_wait on a semaphore in the kernel driver in the same fashion.

V. Performance Evaluation

We have evaluated the performance of the hybrid driver by first calculating the time taken for a NULL function to execute as a traditional driver only in the kernel and the hybrid driver in the user and kernel space. This will give us an estimate of the overhead caused by the hybrid driver.

The time taken to execute NULL function as a traditional kernel driver is 1-2uS. The time taken for a null function to execute from the user driver is 28-170uS. Here the kernel driver places a request for the NULL function to execute in the user driver and then the user driver notifies the kernel driver of its completion and the kernel driver then finishes execution. To make it representative of a real workload we pass a 4K buffer from kernel driver to the user driver before executing the NULL function.

NULL FUNCTION:
Traditional Driver: User application -> kernel driver
Hybrid Driver: User application -> Kernel driver -> user driver -> kernel driver -> return

Driver	Time
Traditional Driver	1 – 2uS
Hybrid Driver	28-170us

In the second section we create a dummy workload which computes the factorial of numbers 10,000 times and evaluates its performance:

Factorial of numbers in loop:

Driver	Time
Traditional Driver	1.025ms
Hybrid Driver: Workload in user	3.076ms
Hybrid Driver : Workload Split equally between user and kernel	2.027ms
Run as a C program	2.80ms

We notice that the code in the kernel runs almost 3x faster than the user application code. This is because the executable execution is managed by the kernel scheduler. While there is no monitoring for the code that runs in kernel space. The time required for execution in user space is dependent upon the time needed for actual code execution and the kernel management time. While the time required for execution in the kernel space is dependent only on the time needed for actual code execution. All the above implementation have been done using a virtual machine. We expect better performance when running on native Linux OS.

References

[1] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan†, Michael M. Swift, Somesh Jha, “The Design and Implementation of Microdrivers,” Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008

[2] Michael M. Swift, Brian N. Bershad, and Henry M. Levy, “Improving the Reliability of Commodity Operating Systems”, Department of Computer Science and Engineering, University of Washington

[3] Shakeel Butt Vinod Ganapathy Michael M. Swift Chih-Cheng Chang, “Protecting Commodity Operating System Kernels from Vulnerable Device Drivers” Proceedings of the 25th Annual Computer Security Applications Conference, Honolulu, Hawaii, December 2009.

[4] Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift and Somesh Jha, “Microdrivers: A New Architecture for Device Drivers,” Published in Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS XI), San Diego, California, May 2007.

[5] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, “Failure Resilience for Device Drivers,” Computer Science Dept., Vrije Universiteit, Amsterdam, The Netherlands.

[6] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, “Fault Isolation for Device Drivers,” Dept. of Computer Science, VU University Amsterdam, The Netherlands.