The Case for Active Device Drivers

Leonid Ryzhyk[‡§] Yanjin Zhu[‡§] Gernot Heiser[‡§¶]
[‡]NICTA* [§]University of New South Wales [¶]Open Kernel Labs
Sydney, Australia
leonid.ryzhyk@nicta.com.au

ABSTRACT

We revisit the device-driver architecture supported by the majority of operating systems, where a driver is a passive object that does not have its own thread of control and is only activated when an external thread invokes one of its entry points. This architecture complicates driver development and induces errors in two ways. First, since multiple threads can invoke the driver concurrently, it must take care to synchronise the invocations to avoid race conditions. Second, since every invocation occurs in the context of its own thread, the driver cannot rely on programming-language constructs to maintain its control flow. Both issues make the control logic of the driver difficult to implement and even harder to understand and verify.

To address these issues, we propose a device-driver architecture where each driver has its own thread of control and communicates with other threads in the system via message passing. We show how this architecture addresses both of the above problems. Unlike previous message-based driver frameworks, it does not require any special language support and can be implemented inside an existing operating system as a kernel extension. We present our Linux-based implementation in progress and report on preliminary performance results.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Input/Output; B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices

General Terms

Reliability, Performance

Keywords

Device Drivers, Concurrency, Stack Ripping

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

APSys 2010, August 30, 2010, New Delhi, India.

1. INTRODUCTION

Device drivers in most operating systems (OSs) today are passive objects. A driver constitutes a collection of entry points that are invoked by the OS when it needs to perform an I/O or a configuration transaction or deliver an interrupt notification to the driver. The driver does not have its own thread of control, but rather executes in the context of external OS threads. Even if the driver creates one or more helper threads to perform auxiliary tasks, the main logic of the driver is still invoked from external threads.

The passive model enables the OS to efficiently communicate with the driver via simple function calls. However it suffers from serious drawbacks that complicate driver development and lead to errors. First, modern OS kernels are multithreaded, hence the driver must be prepared to handle concurrent invocations by multiple threads. Concurrency management accounts for approximately 20% of all driver bugs [10]. Second, since every driver invocation occurs in the context of its own thread, the driver must maintain its execution state across invocations using state variables. This makes the control logic of the driver difficult to implement and even harder to understand and verify—the phenomenon known as stack ripping [1]. As discussed in Section 3, approximately 10% of driver bugs are related to stack ripping.

To address these issues, we propose a device-driver architecture where the driver has its own thread of control. Communication between the driver thread and other OS threads occurs via message passing. The driver/OS interface consists of a set of *mailboxes* where each mailbox is used for a particular type of request. The OS sends an I/O request or an interrupt notification to the driver by placing it into the appropriate mailbox. The driver receives the request by performing a blocking wait on one or more mailboxes. The driver notifies the OS about a completed request via a reply mailbox.

In contrast to the passive architecture, where the driver can be invoked at any time, an active driver determines when it can process the next request itself. Since the driver handles all requests in the context of its own thread, it does not have to worry about thread synchronisation.

An active driver is structured as a sequential program with explicitly defined control flow. The state of the program is automatically maintained by the compiler, resulting in much clearer code. Furthermore, the order in which the driver handles OS requests is explicit in the structure of the program. In contrast, a passive driver is merely a collection of functions without any indication of the order in which these functions are invoked.

Some types of drivers lend themselves naturally to multiple concurrent control flows. For example, receive and transmit paths of a network driver are largely independent and can be implemented using two parallel threads. Our architecture allows such multi-

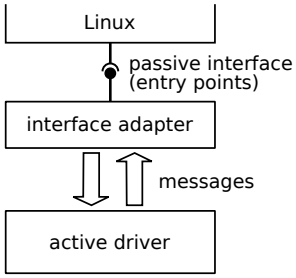

Figure 1: Architecture of the active driver framework for Linux. The lollipop connector represents the passive driver interface consisting of entry points.

threaded drivers. Unlike multithreading in passive drivers, this kind of multithreading captures the control structure of the driver and does not lead to stack ripping. In addition, driver threads are scheduled cooperatively, thus avoiding most of the problems related to thread synchronisation.

In order to evaluate the proposed architecture, we are implementing an active driver framework and sample drivers for Linux. Our framework transparentlly integrates active drivers into the Linux driver stack. To this end, it associates an adapter with each driver, which exports a conventional passive interface to the Linux kernel and translates invocations received through this interface into messages to the driver (Figure 1). This architecture allows active device drivers to co-exist with conventional passive drivers.

The improved programming model should not come at the cost of performance. In our previous work [10], we showed that request serialisation does not noticeably affect the performance of the majority of drivers. For the remaining few drivers that can benefit from parallel request processing on a multicore system, we are working on an extension of the active architecture where multiple driver threads can run and receive OS requests concurrently. This extension is beyond the scope of this paper.

Another potential source of performance overhead is message-based communication, since sending a message between two threads is much more expensive than invoking a driver entry point. Existing mechanisms in the Linux kernel do not allow efficient implementation of the communication semantics required by active drivers. We describe our implementation of a high-performance message-passing mechanism for Linux in Section 5 and present initial performance evaluation in Section 6.

This paper makes three contributions. First, we identify and analyse two types of device-driver architecture: passive and active drivers. While both types have been used in the past, no systematic comparison between them has been undertaken. Second, we propose an active device-driver architecture that addresses the shortcomings of the passive architecture. Unlike previous approaches, this architecture does not rely on programming-language extensions, thus eliminating a major barrier to its adoption by mainstream OSs. Third, we present an implementation and initial evaluation of an active driver framework for Linux that does not require any special compiler support nor modifications to the existing kernel code.

2. RELATED WORK

The work presented here is a follow-up to our previous research on the Dingo driver architecture [10]. Dingo reduces the amount of concurrency in drivers by serialising driver invocations. In order to avoid performance degradation due to serialisation, all blocking operations must be rewritten using completion chaining, which exacerbates the stack ripping problem. Dingo addresses the problem by introducing a C language extension that allows writing event-driven programs using the sequential style. In contrast, the active device-driver architecture proposed here addresses both the concurrency and the stack-ripping problem without requiring any special language support.

Previous studies of driver errors [4, 10] identified concurrency as a major source of software defects in drivers. Adya et al. [1] studied the stack-ripping problem in the context of arbitrary event-driven programs and proposed a user-level runtime framework that avoids stack ripping.

In the context of device drivers, stack ripping was studied by Chandrasekaran et al. [3]. They consider stack ripping as an inherent property of event-driven programs and do not attribute it to the specific device-driver architecture supported by current OSs. They point out that stack ripping complicates static analysis of the driver control flow and propose a new programming language, called Clarity, that enables more effective static analysis for drivers.

A device-driver architecture where a driver has its own thread of control was introduced by Dijkstra [5] for the THE OS. However, this approach was not adopted by mainstream OSs. Research systems that support active drivers include Singularity [6] and RMoX [2]. Similarly to Clarity and Dingo, these systems rely on programming language support for threading and communication.

User-level device-driver architectures [8, 9] encapsulate each driver in a separate process with its own thread of control, which puts them in perfect position to support active drivers. Instead of doing so, they provide runtime environments that simulate the passive programming model for user-level drivers.

3. PROBLEMS WITH PASSIVE DRIVERS

3.1 Concurrency

Multithreaded programming is inherently error-prone. The kernel environment imposes additional constraints that further complicate concurrency management. For example functions invoked by the kernel in the interrupt context are not allowed to perform blocking operations, since blocking in the interrupt context can lead to a deadlock.

Figure 2 shows a fragment of the RTL8169 Ethernet controller driver for Linux. The `tx_timeout` entry point (line 1) is invoked by the networking code when a packet transmission fails to complete within a reasonable period, indicating a hardware lock-up. The driver handles this request by resetting the device, which involves blocking operations, such as waiting for an interrupt from the device. The `tx_timeout` function is invoked from the timer interrupt handler and hence should not block. Therefore, instead of calling the `reset` function directly, it schedules this function for execution in the context of a kernel worker thread (line 3), thus introducing another concurrent activity that the driver must handle.

Another problem is related to the use of kernel locking primitives. Since the driver is embedded into the OS kernel, it must be aware of the context in which each entry point is invoked, i.e., which locks are held by the kernel and which locks can be safely acquired by the driver. In the RTL8169 driver example, the `reset` function must be synchronised with other driver functions that modify the network interface configuration, such as `open` and `close`. The implementation in lines 5–11 leverages an existing kernel lock used to serialise all network interface re-configurations. The kernel holds this lock when invoking the `open` and `close`

```
1 void tx_timeout(struct net_device *dev){
2   ...
3   schedule_work(reset_work);
4 }
5 void reset(struct work_struct *work){
6   rtnl_lock();
7   if (!netif_running(dev)) goto unlock;
8   <reset the device>
9 unlock:
10  rtnl_unlock();
11 }
12 int open(struct net_device *dev){
13  <initialise the device>
14 }
15 int close(struct net_device *dev){
16  <shut down the device>
17 }
```

Figure 2: A fragment of the RTL8169 Ethernet controller driver for Linux.

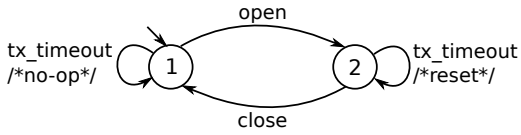

Figure 3: Ethernet driver interface state machine.

entry points. By acquiring the same lock in line 6, the `reset` function avoids race conditions with other configuration functions.

Coordinating multiple concurrent activities while coping with the constraints of the kernel environment has proved an impossible challenge for most driver developers. Concurrency errors are particularly common in corner-case scenarios, e.g., when the driver receives a hot unplug notification while handling a suspend request.

3.2 Control-flow management

Device drivers are stateful objects, i.e. the driver's reaction to a request depends on the history of previous requests and responses. When implementing the driver, the developer determines various sequences of requests that the driver must handle and how the driver responds to these sequences. In a sequential program, these scenarios can be directly encoded using programming language constructs and can be later analysed by following the program structure. In contrast, a driver comprised of entry points invoked from external threads must maintain its control flow using explicit state variables. The resulting driver implementation does not fully capture the developer's intention and is hard to understand and maintain.

The state machine in Figure 3 describes some possible sequences of Ethernet driver invocations in Linux. The `tx_timeout` function can be invoked in both states 1 and 2. In state 2, the network interface is active and the `tx_timeout` function must perform a device reset. In state 1, the interface is inactive and `tx_timeout` must behave as a no-op.

The high-level structure of the driver in Figure 2 does not provide any information about sequences of commands that the driver can handle. In particular, there is no indication that a close request must always follow an open, while a timeout notification can be handled in any state. All three operations are implemented as driver entry points and can, in principle, occur in any order.

Digging deeper into the implementation, we observe that the `reset` function performs the `netif_running` check in line 7 before resetting the device. The `netif_running` kernel API checks whether the network interface is in the active state. It returns a non-zero value in state 2 and zero in state 1. A similar check could be performed by using a driver state variable to keep track of the interface status.

In effect, the driver correctly implements the reset logic described above; however, due to stack ripping, this logic is spread across different parts of the driver. Understanding the driver logic requires an in-depth knowledge of the communication protocol between the driver and the OS. This complicates driver development and leads to errors. Our earlier study showed that errors related to the ordering of driver-OS interactions, including situations when the driver does not correctly handle a sequence of OS requests or invokes OS function in the wrong order, constitute approximately 10% of all driver errors [10].

4. THE ACTIVE DRIVER ARCHITECTURE

4.1 Scheduling and communication

The proposed device-driver architecture extends the OS kernel with two new abstractions: cooperative domains and mailboxes.

A *cooperative domain* is a collection of one or more threads. Threads inside the domain are scheduled cooperatively with respect to each other, i.e., only one of them can be runnable at any given instance. The cooperative domain is scheduled preemptively with the rest of the kernel, i.e., a domain thread can run concurrently with other kernel threads and other cooperative domains. A cooperative domain is created with one initial thread. Additional threads can be spawned at runtime.

A *mailbox* is a message-based communication medium. Messages are typed objects implemented as C structures. A mailbox is declared to only accept a particular type of message. Multiple message instances can be queued in the mailbox and delivered in the FIFO order.

Mailboxes can be of three types. Incoming mailboxes are used to send messages from normal kernel threads to threads inside a cooperative domain. Outgoing mailboxes are used to send message from the domain to the kernel. Internal mailboxes are used for communication among threads inside the domain. A mailbox is associated with a cooperative domain at creation time.

Message exchange occurs using two main operations. The `SEND` operation places a message in a mailbox. The sending thread continues without blocking. The `WAIT` operation takes references to one or more mailboxes and blocks until a message arrives to one of these mailboxes. It returns a reference to the mailbox containing the message. When a thread becomes blocked in a `WAIT` call, another thread from the same domain can be made runnable. Multicast communication is currently not supported, i.e., at most one thread can be waiting on a mailbox.

4.2 The active driver architecture

An active driver occupies its own cooperative domain. When the driver is being instantiated, the active-driver runtime framework creates a new cooperative domain and calls the main function of the driver in the context of the domain's initial thread. The driver can spawn additional cooperative threads and set up mailboxes for communication among them at runtime.

The active driver architecture is supported by a runtime framework implemented as a kernel extension. An active driver must register a pointer to its main function and the list of devices that it can handle with the framework. Once a supported device has been detected, the framework creates a new cooperative domain and runs

```
1 int main (void * arg) {
2   ...
3   while (1) switch (state) {
4     case state_closed: /* state 1 */
5       mb = WAIT(open, tx_timeout, ...);
6       if (mb == open) {
7         <initialise the device>
8         SEND(open_complete, status = 0);
9         state = state_open;
10      };
11      break;
12
13    case state_open: /* state 2 */
14      mb = WAIT(close, tx_timeout, ...);
15      if (mb == tx_timeout) {
16        <reset the device>
17      } else if (mb == close) {
18        <shut down the device>
19        SEND(close_complete, status = 0);
20        state = state_closed;
21      };
22      break;
23  };
24};
```

Figure 4: An active implementation of the RTL8169 driver.

the driver's main function in the context of its initial thread. The driver can spawn new cooperative threads at runtime and create internal mailboxes for communication among internal threads.

The framework creates an interface adapter (Figure 1) for each interface of the driver. Most drivers support at least two interfaces (and hence need two adapters): a device-class interface that the driver implements in order to allow the rest of the OS to use the device and a bus-transport interface used to communicate with the device over an I/O bus, such as USB or PCI.

The device-class interface adapter receives I/O requests from the OS and translates them into messages to the driver. Depending on request type, the adapter either blocks the calling thread waiting for a reply message from the driver or returns control to the kernel immediately. Requests that can occur in the primary interrupt context and are therefore not allowed to block are delivered to the driver as an asynchronous notification, i.e., a message that does not require a response message.

The bus-transport interface adapter accepts bus transfer request messages from the driver and translates them into corresponding bus transactions. It also translates interrupt notifications from the device into messages to the driver.

4.3 Example

Figure 4 shows a fragment of an active version of the RTL8169 controller driver that implements the behaviour shown in Figure 3. The only entry point exported by the driver is `main`; all interactions between the driver and the OS occur via messages.

When in state 1 (line 4), the driver accepts open and timeout requests by waiting on the corresponding mailboxes (line 5). If an open request arrives (line 6), the driver initialises the device, sends a completion notification back to the OS, and moves to state 2 (lines 7–9). If a timeout request arrives, the driver simply ignores this request and waits for the next message. In state 2, the driver accepts close and timeout requests (line 14). It handles a timeout by resetting the device (line 16). In response to a close request, the driver shuts down the device and returns to state 1 (lines 18–20).

This implementation is more verbose than the original (passive) one. It explicitly specifies the control flow of the driver and lists requests that the driver handles in every state. While this results

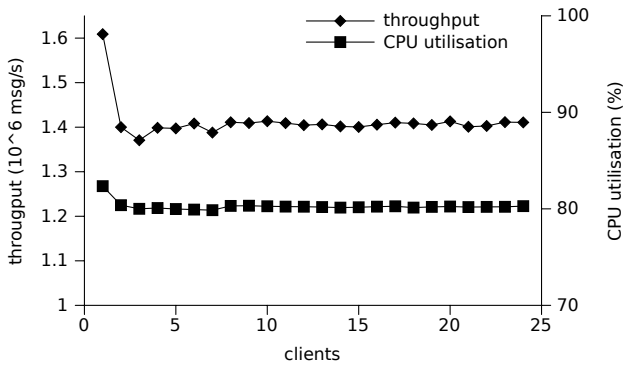

Figure 5: Message throughput and aggregate CPU utilisation over 8 CPUs for varying number of clients.

in an increased amount of code, this code is easier to understand and validate than the shorter passive version. For example, if the driver developer forgets to handle the `tx_timeout` request in either state 1 or state 2, the omission can be detected by inspecting the list of mailboxes that the driver listens to in every state. In addition, since all driver operations are serialised, it does not contain any of the synchronisation complexity of the original implementation.

The switch statement provides a natural way to structure state-machine-based code. Every case of the switch describes how the driver behaves in a particular state. In contrast, in a passive driver this information is spread across multiple entry points.

5. IMPLEMENTATION

We are implementing support for active device drivers along with several sample drivers for Linux. The critical component of the active driver infrastructure is the scheduling and communication mechanism.

Our implementation associates a *message queue* and a *message dispatcher* with each cooperative domain. When a Linux thread sends a message to a mailbox associated with the domain, the message is placed on the queue. If the driver is currently idle, the dispatcher is invoked. It inspects the first message on the queue, locates its target mailbox, and checks if there is a thread waiting on this mailbox. If so, it makes this thread runnable. The thread runs for some time before terminating or blocking on a `WAIT` operation. At this point, the dispatcher is invoked again. It marks the current thread as stopped and chooses the next runnable thread by inspecting the message queue.

6. PRELIMINARY EVALUATION

We evaluate the performance of the initial implementation of the message passing mechanism on a machine with 2 quad-core 1.5GHz Xeon CPUs. In the first set of experiments, we measure the communication throughput by sending a stream of messages from a Linux thread to a thread inside a cooperative domain. This simulates streaming of UDP packets through a network driver. The achieved throughput is $2 \cdot 10^6$ messages/s with both threads running on the same CPU and $1.2 \cdot 10^6$ messages/s with the two threads assigned to different CPU cores on the same chip. To put these numbers in context, a Gigabit NIC can send up to 10^5 1Kbyte packets per second. Hence, we expect message passing to introduce less than 8% CPU overhead without affecting the network throughput for such drivers.

Second, we run the same experiment with varying number of Linux threads distributed across available CPU cores (without enforcing CPU affinity), with each Linux thread communicating with the Dingo thread through a separate mailbox. As shown in Figure 5, we do not observe any noticeable degradation of the throughput or CPU utilisation as the number of clients contending to communicate with the single server thread increases (the drop between one and two client threads is due to the higher cost of inter-CPU communication).

Third, we measure the communication latency between a Linux thread and an active driver thread running on the same CPU by bouncing a message between them in a ping-pong fashion. The average measured roundtrip latency is 1.8 microseconds. For comparison, the roundtrip latency of a Gigabit network link is at least 55μs [7].

7. CONCLUSION

We proposed an active device driver architecture that simplifies concurrency management and eliminates stack ripping in drivers. Our ongoing research focuses on validating this architecture by applying it to develop high-performance drivers for a wide range of devices.

8. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *2002 USENIX*, Jun 2002.

[2] F. Barnes and C. Ritson. Checking process-oriented operating system behaviour using CSP and refinement. *Operat. Syst. Rev.*, 43(4):45–49, Oct 2009.

[3] P. Chandrasekaran, C. L. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with CLARITY. In *6th ESEC*, pages 65–74, Dubrovnik, Croatia, 2007.

[4] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[5] E. W. Dijkstra. The structure of the "THE" multiprogramming system. *CACM*, 11:341–346, 1968.

[6] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *1st EuroSys Conf.*, pages 177–190, Apr 2006.

[7] R. Hughes-Jones, P. Clarke, and S. Dallison. Performance of 1 and 10 gigabit ethernet cards with server quality motherboards. *Future Generation Computer Systems*, 21(4):469–488, 2005.

[8] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sep 2005.

[9] Microsoft Corporation. Architecture of the user-mode driver framework, 2007.

[10] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *4th EuroSys Conf.*, Apr 2009.