In-Kernel Memory-Mapped I/O Device Emulation

[1,4]*V.Yu. Cheptsov<cheptsov@ispras.ru>*
[1,2,3,4] *A.V. Khoroshilov<khoroshilov@ispras.ru>*
[1] *Ivannikov Institute for System Programming of RAS*
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russia
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] *Higher School of Economics.*
20, Myasnitskaya Ulitsa, Moscow 101000, Russia

Abstract. Device emulation is a common necessity that arises at various steps of the development cycle, hardware migration, or reverse-engineering. While implementing the algorithms behind the device may be a nontrivial task by itself, connecting the emulator to an existing environment, such as drivers intended to work with the actual hardware, may be no less complex. Devices relying on memory-mapped input/output are of a particular interest, because unlike port-mapped input/output there is much less of a chance that the target platform provides a direct interface to intercept the transmissions. A well-known approach used in various virtual machine software is to put the entire operating system under a hypervisor and build the emulator externally. This may not be desirable for reasons like hypervisor complexity, performance loss, and additional requirements for the host hardware. In this paper we extend this approach to the kernel and explain how it may be possible to build the emulator by relying on the existing interfaces provided by an operating system. Given the common availability of an MMU unit as well as memory protection mechanisms, allowing the handling of page or segment traps at read or write access, we presume that a suggested technique of intercepting memory-mapped input/output could be implemented in a broad number of target platforms. To illustrate the specifics and show potential issues we provide the ways to simplify the implementation and optimize it in speed depending on the target capabilities, the protocol emulated, and the project requirements. As a working proof we created a SMC emulator for an x86 target, which makes use of this approach.

Keywords: device emulation; memory-mapped i/o; kernel modules

DOI: 10.15514/ISPRAS-2018-30(3)-9

For citation: Cheptsov V.Yu., Khoroshilov A.V. In-Kernel Memory-Mapped I/O Device Emulation. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 121-134. DOI: 10.15514/ISPRAS-2018-30(3)-9

1. Introduction

One of the common engineering demands is device emulation. It may arise during the software development cycle, for example, in testing or driver verification, at hardware migration, when there is no easy way to rewrite the existing software. Other than that, in the world of proprietary hardware and software it is not rare that the only way to understand and document the device abilities is to reverse-engineer it, and the ability to dynamically debug or reverse-engineer the code could be the key in security analysis or adding the device support to a virtual machine.

Speaking of virtual machines, or rather hypervisors, building the entire virtual stack for a single device one needs to emulate is often an overkill due to performance reasons, although it could be partially mitigated by hardware-assisted virtualization and software compatibility. The latter may involve working on completely unrelated parts of the driver stack and result in unnecessary costs for continuous support.

However, while the development of full platform emulators is a considerably common topic with abundance of existing papers and products like qemu, bochs, iOS simulator, etc., peripheral emulation is much less widespread. In some cases, virtual machine guest tools do try to mimic certain hardware, but even that is usually implemented as a part of a full scale platform emulation. The problem with the peripherals is not just in implementing the algorithms behind the device, which may be a nontrivial task by itself, but also connecting the emulator to an existing environment, such as other drivers above in the stack intended to work with the real hardware.

Since one of the important aspects of using any peripherals is the ability for the CPU to communicate to them, the common demand for a device emulator is to provide a way to do it. Presently there are two common low-level approaches to perform input and output operations: port-mapped I/O (PMIO) and memory-mapped I/O (MMIO). While there are other ways such as involving some dedicated hardware, they are relatively less widespread. High-level communications operating on a packet basis (like USB bus) usually go through the dedicated abstraction layer, and thus may be implemented with the standard APIs offered by the operating system without any special effort.

It is fairly easy to implement communication protocols with a hypervisor, the standard approach is to ensure that accessing certain memory exits the virtual machine context (vmexit), which is later handled by the implementation. However, as we mentioned previously, the use of a hypervisor may be impractical, and we have to look for other means of intercepting memory access. Since direct memory access is very common, yet quite problematic to intercept, in this paper we explain how one could implement a considerably portable MMIO emulator in the kernel and cover the details of emulating device communication protocols on common platforms.

2. State of the art

We admit to not being the first to experiment with peripheral device emulation. Every single year several published papers in the field of hardware virtualisation cover this topic to a certain level. Articles published by VMware Inc. researchers [1] [2] provide an in-depth coverage of x86- compatible hardware emulation. They explain the existing obstacles and necessary actions to be taken to implement a complete virtual stack from the CPU to network adapters. In their works they pay a lot of attention to performance optimization, hardware-assisted virtualization and show a visible performance penalty reduction over the new CPU generations in Virtualization nanobenchmarks section of the first referenced paper.

As a result of continuous contribution from different parties and competitive product development, the general hypervisor performance has dramatically improved. While GPU emulation is out of the scope of this paper, it should be admitted that there are several works which do manage to provide a complete GPU emulation at a reasonable performance [3] [4]. These works feature an open GPUvm platform in the Xen hypervisor.

Another related direction involves security analysis or reverse-engineering. While less frequently found in academic writing, there are several products, tools, and patches for Linux intended to log execution details from the Linux kernel for later analysis. One of the most well-known toolsets is Linux Trace Toolkit, and one of the most prominent cases of applying the approach in practice is for Nouveau driver development for NVIDIA GPUs. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack by Konstantinos Menychtas, Kai Shen, and Michael L. Scott [5] provides a good coverage in detail.

3. Basic I/O Introduction

Port-mapped I/O is usually more demanding to the CPU instruction architecture and requires a number of so-called ports the devices will be mapped to, and perhaps a dedicated instruction set to access these ports as well. Because the device memory is accessed indirectly, another name for PMIO is detached I/O.

As an example, one of the most popular architectures to implement PMIO is x86. It can be utilized by means of two dedicated instructions: in and out, which enable one to receive and send 8, 16, or 32 bits of data to a port from 0 to 65535. Since there are faster ways to perform I/O on x86 and PMIO is not recommended for use nowadays, in some literature it may be referred to as legacy I/O. This may not be the case for other architectures found in micro-controllers, but in general MMIO support is increasing.

Memory-mapped I/O involves direct mapping of the device memory to the host memory, enabling the software to access the device just like a normal chunk of noncacheable RAM with the use of the native instruction set. Since MMIO implementation is often faster than PMIO and sometimes simpler to use, it will be the one to opt for when implementing a device communication protocol. For example, on x86 various devices installed as PCI extension cards or system management controllers make a use of it.

Virtual devices are not supposed to be functionally different from real hardware. For this reason, emulators have difficulties supporting I/O communication protocols. The taken approach varies depending on the demands and available resources, but usually one of the following is used:

- Custom device development
- Driver reimplementation
- Building a hypervisor

Sadly, each of these has serious limitations, and most of them create obstacles for generic peripheral emulation, as observed in Table. 1.

Table 1. Pros and cons summary

	Device	Driver	Hypervisor
Software independency	+	-	±
Low costs	-	±	+
Legal issues	+	-	-
Infrastructure dependency	-	+	-
Forward compatibility	+	-	-
Performance	+	±	±
Other device support	+	+	-

Developing a new device by extending a microcontroller to offer a required interface or creating an entire chip mostly works for very simple devices when a single copy is going to be used for some kind of deep debugging or instrumentation. A good example could be removable BIOS chips for debugging or HDMI to VGA adapters with HDCP decoding. While this solution is very reliable for creating a test device, the results of mass-producing a customised device will likely be not worth the effort. It will be either more expensive or worse in quality. In addition, it is important to have the legal part of the question in mind and avoid patent infringement. However, this method could be most reliable when it comes to stability.

Reimplementing the driver to support another communication interface for the virtual device is very useful when working with performance-critical hardware such as GPUs. For them each extra communication layer may heavily affect the performance due to high bandwidth usage, and that is why virtualization software implements extended GPU support (like DirectX or OpenGL) in such a way. However, in our case it defeats the entire purpose of creating a virtual device. If the point is to test the driver, it will no longer stay the same. If the reason is to support a proprietary driver, one will have to reverse-engineer it and have issues every time it gets updated.

Bringing in a virtual machine with a hypervisor is a way to overdo it. While a decent virtual machine has a wide range of supported hardware, it adds a lot of downsides as well. In particular there will always be potential performance issues, even with hardware-assisted virtualization support. More than that, compatibility issues will likely become a blocker if the rest of the environment is not generic and well-known. It is unfortunate, but even the mainstream operating systems may be unwilling to expose new interfaces for virtual machines (like most of the graphical stack on Apple macOS).

4. Intercepting the I/O

As a result I/O interception comes out as a pragmatic way to achieve the goal. Despite not being very common, software and hardware have enough capabilities to intercept raw device communication without touching the higher-level drivers themselves.

For example, for the past 8 years the recent x86 firmwares contain a dedicated UEFI System Management Mode [6] protocol to intercept PMIO. This protocol originally existed as a EFI_SMM_IO_TRAP_DISPATCH_PROTOCOL protocol[1], but later on was extended with an additional IO_TRAP_EX_DISPATCH_PROTOCOL protocol[2]. Both protocols allow you to create direct handlers to intercept the portmapped access. By design, the management mode affects the operating system code as well, so it works throughout the boot process and is fully transparent to the higher level software implementations like OS kernel or drivers. However, aside from not being very well documented, third-party code execution in the System Management Mode is generally prohibited. So even if one is to reimplement the SMI handler similar to what Intel offers with the open source platform code, it will be of no use for anyone but UEFI firmware developers.

Fortunately, most of PMIO interface code is usually well abstracted in the kernel, and when it comes to intercepting you could just replace the underlying low level function implementation within the emulator context. However, devices relying on MMIO are of a particular interest, because unlike PMIO there is a much less chance that the target platform provides a direct interface to intercept the transmissions.

For embedded devices it may well be sufficient to statically analyze the firmware, find the instructions responsible for I/O, and either dynamically or statically overwrite them to jump to prepared thunks that will handle them accordingly. This approach is common for security analysis especially when very little is known not only about the explored peripherals but the whole controller. However, since the firmware or the driver may receive updates in the future, this approach is not very effective outside of security or code coverage analysis, and the like.

One of the first ideas that comes to mind due to the nature of MMIO writes is relying on CPU debug registers. These registers (e.g. DR on Intel or BP_CTRL/BP_COM on ARM Cortex) allow you to implement hardware breakpoints or rather watchpoints, which may trap read and write access. However, these registers are very few, and their scope area is small (i.e. a 32-bit or 64-bit word). Other than that, the kernel,

[1] GUID: 58DC368D-7BFA-4E77-ABBC-0E29418DF930

[2] GUID: 5B48E913-707B-4F9D-AF2E-EE035BCE395D

debuggers, or other software may use these registers for their own needs, which leads to them being simply impractical for this kind of work.

In general-purpose operating systems with defined kernel APIs there are much better ways, such as a page protection mechanism, which is used to implement watchpoints in software. While this is suitable for doing MMIO emulation, most of the known works relying on this technique either use it for tracing or just for debugging backends. The notable example is MMIO trace in Linux, which was originally developed to reverse-engineer proprietary NVIDIA drivers by tracing the register access [7]. Other than that, there are very few examples of how it can be utilized for device I/O emulation.

5. Proposed approach

The idea of general purpose I/O interception is very simple: catch reads and writes, make sure that the values read are correct, and the values written are accounted for. To apply it to MMIO we could limit page protection of the target area, and trap the faults as they happen. Due to bandwidth limitations and architecture simplicity the I/O sequences are generally serialized, even if they happen from different threads. It may not be the case for GPUs, yet GPUs likely will not need this kind of emulation due to performance reasons. Still, in general if serialized I/O is not guaranteed even within a single memory page (which is rare) one could always implement it manually by utilizing the synchronization primitives.

Therefore, the most obvious approach will be:

1. mark the relevant page as neither writable nor readable (not present in x86 terms);
2. catch a fault and decode the fault address and the direction (in or out);
3. disassemble the instruction that caused the fault and obtain its operands from the frame;
4. handle the operands for the emulation;
5. update the destination registers or memory for the reads as necessary;
6. return to the location after the instruction, which caused the fault.

While it indeed solves the problem and looks very straightforward, the implementation itself could be very convoluted. While the saved context is likely to contain the fault and return addresses, bringing a full-scale disassembling framework to the kernel is inflexible due to extra architecture dependencies and considerable amounts of code required for instruction emulation. Even more, it may impose additional performance penalties, which are already tough enough.

For these reasons we tried to alter the algorithm in a way that would be simpler, less platform-dependent, and similarly performant. After examining several real-world examples, we consider the following model of a MMIO-based I/O protocol, which could be applied to quite a number of devices:

1. host ensures that the target is ready for an I/O operation;

2. host performs the I/O operation (by reading or writing at a defined address space);
3. host ensures that the operation is complete and repeats the process.

The 1st and 3rd steps are usually implemented as a write-and-poll, a write-and-interrupt or just as a poll. Another advantage comes out from common differences in frequencies between the host and the peripheral. Since communications usually happen between the devices with different clock bases, most of the protocols are synchronous, and the host generally does not overwrite the areas it has just written to without making a read to confirm it was successful. Even more, most of the protocols are stateful, and it is uncommon to see subsequent reads from the same place expecting the value to change more than once. A write operation will most likely appear in-between.

Under these assumptions we use a simple satisfactory transaction model as an example:

1. write operation type (read or write);
2. read acknowledge status until status ready;
3. handle the values:
   - 3.1. read the value for read operations;
   - 3.2. write the value for write operations and read acknowledge status until status ready.

5.1 With write-only page support

If write-only pages are supported, in a number of cases one may implement a flip-flop approach that will switch page protection from read-only to write-only and backwards as the process goes.

To emulate the proposed transaction we could start the communication process with the page marked as read-only, which will then trap on operation type. Here we will initiate the transaction and switch the protection to write-only. After the operation is written the trap on the status read will trigger, where we will read the written operation type, update the value for read operations and set its status. Afterwards the page protection is returned to read-only and the control is transferred back to the driver. For read operations that is all of it, for write operations the driver will read the status and attempt to perform the actual write, which should trigger the trap again. From there on one could repeat the process as described for the operation type. In the end for both reads and writes page protection returns back to read-only, eliminating any platform-specific disassembling and relying on generic approach.

Expectedly one does not have any easy access to write-only pages on popular architectures such as ARM [8] or x86. Perhaps, if these architectures were originally designed at present, when the demand for better memory protection management is much higher and when features like W^X memory and execute-only memory have already become commonplace, we would have had finer memory management that would support write-only pages. However, nowadays write-only pages are not very common in both hardware and software implementations. Certain PowerPC implementations [9] or processor extensions may provide access to them, so it remains a good idea to check CPU manuals before abandoning the try. For example, Intel x86 processors starting from Nehalem technically support write-only memory via EPT (Extended Page Tables [10]), yet it can hardly be used for anything but virtualization.

5.2 Without write-only page support

When write-only pages are not available, we may still be able to work out a simpler approach, and this is where memory patching comes in hand. The idea is to let the original instruction perform the I/O just as normal, but to encode a jump-back instruction right afterwards to ensure that page protection is limited again to trap the next I/O operation. Initially this approach may appear to have too many issues to be considered in practice, however, they could all be solved with enough effort, and some of them could even be turned into benefits.

The first issue to solve is the length of the faulted instruction. A number of architectures provide fixed-length instruction sets, so the next instruction address to encode our jumpback instruction could be calculated even without knowing anything about the current instruction. For others one could write or find simple instruction fetchers, to only decode the length without operand or operation details. Such software may also go under the name of length disassemblers, and various implementations exist for popular platforms [11]. It may become a little more involved when the I/O instruction results in non-linear control flow, but in general I/O and branching instructions belong to separate classes and are not mixed together.

The second issue occurs when the device memory is mapped to userspace and the communication happens in userspace as well. In this case a direct jump to protection restoration code is not possible, and a breakpoint or similar instruction will have to be encoded to trigger the context switch, return to the kernel and pass the control to our handler.

The third and probably the most serious issue happens when I/O operations are performed through shared code. By assuming serialized I/O we consider no cases of simultaneous code execution from the same area (unless there are multiple devices). Therefore, we could safely patch it. However, nothing prohibits the driver from utilizing generic memory primitives like memcpy or memset to bulk-write or read the dedicated area. These primitives generally have no effect on the I/O itself, and we do not need to intercept every byte they touch. To avoid the issue one could examine the stack trace and modify the instruction at the return address. Not only this does not require disassembling but also reduces the penalty from trapping extra I/O operations, so a quick stack unwinding that can often be implemented with compiler intrinsics easily pays off.

With all the pieces put together it creates a solid approach for a large chunk of I/O protocols. In addition to these general improvements platform-specific optimizations could be applied. For example, extra page protection changes may be avoided for write operations, if the hardware may ignore interrupts caused by write protection violation (CR0 WP bit on x86). It should be noted that one is to pay extra attention to the scheduler (e.g. disable preemption) not to let it switch the task to another core, where write protection is on.

6. Evaluation

To apply the proposed solution in practice we created a software-based emulator for the 2nd generation Apple SMC in a form of a kernel extension for Apple macOS. System Management Controller (SMC) is a chip commonly found in Intel-based Apple Macintosh computers or certain Google Chromebooks. This chip is responsible for computer power management, display backlight control, HDD monitoring, thermal control, hybrid sleep and hibernation support, external device current regulation (AirPort, USB, FireWire), charging the battery, trackpad controls, screen mirroring, etc. This chip is not essential for computer functioning, and could be viewed as a convenience feature for a vendor to rely on to centralize and simplify hardware management.

There are two main generations of SMC controllers in Apple computers. The 1st generation was built on a 16-bit Renesas H8S/2117 controller and exposed port-mapped I/O interfaces to communicate with the operating system. The 2nd and subsequent generations are based on 32-bit ARMv7-A processors, and expose memory-mapped and port-mapped I/O interfaces. Both approaches are used to implement the same functionality within a single synchronous stateful protocol. Initially the communication happens via the PMIO protocol, and then a switch to MMIO protocol happens if the device supports it. The whole communication process happens within the kernel and the existing drivers for the 2nd generation hardware are closed-source. Fortunately, due to side researchers the communication protocols are mostly documented [12].

The reasons for taking this particular device into consideration was not only because it is a challenging task compared to devices with open specifications and decent documentation, but also for the importance of having better control of the hardware you use. Apple SMC has complete access to every device in the system and could monitor the bus communications. Other than that it stores temporary encryption keys for hibernation images or user action free restarts (authenticated restarts), when full disk encryption is enabled. Apple SMC drivers expose a dedicated protocol to userspace. This protocol provides a way to obtain SMC data and configure both SMC and onboard devices. Given its direct connection to the hardware, it may be possible to inflict damage on the computer by overheating or causing power surges. Moreover, previous researches discovered that it was very easy to modify SMC firmware, which is also a very serious concern [13].

The actual implementation follows the proposed approach without write-only page support with all the suggested optimizations and certain platform-specific adjustments. SMC MMIO protocol covers a 64 KB area, which we split into pages with the dedicated handlers based on the page index. Since the access to each page is serialized, no additional I/O wrapping is necessary.

In the XNU kernel, which powers all modern Apple hardware including Macs, Intel CPU exceptions are routed through a dedicated kernel_trap function. To let the driver communicate with the emulated device we added a SMC nub via the standard I/O Kit APIs with mapped memory regions with restricted protection and extended the kernel_trap function in EXC_I386_PGFLT handling code specifically for our memory.

A simplified version of this code is shown in Listing 1. ioRegionStart and ioRegionEnd locate the emulated I/O area starting and ending addresses, appleSmcStart and appleSmcEnd point to the AppleSMC driver address range. instrSize function calculates the instruction length at the return address to later write the jump-back code via writeTrampoline function, which not only writes the trampoline code (by disabling the WP bit and interrupts) but additionally disables CPU preemption to avoid the scheduler switch.

```
auto faultAddr = state->cr2;
if (faultAddr >= ioRegionStart &&
    faultAddr < ioRegionEnd) {
  auto retAddr = state->rip;
  if (retAddr >= appleSmcStart &&
      retAddr < appleSmcEnd) {
    // Simple case (from AppleSMC)
    retAddr += instrSize(retAddr, 1);
  } else {
    // Complex case (from e.g. memcpy)
    retAddr = unwindToSMC(state->rsp);
  }

  auto faultType = FaultTypeRead;
  if (state->err & T_PF_WRITE) {
    faultType = FaultTypeWrite;
  }
  updateProtection(faultType, faultAddr);
  saveOrgCode(retAddr, TrampolineSize);
  writeTrampoline(faultType, faultAddr);
  return;
}
```

Listing 1. Sample code

To transfer the control flow to the protocol emulator updateProtection is performing the actual protection upgrade of the emulated I/O area and invokes the read access handler. It should be noted that a dedicated procedure may be needed for platforms with delayed physical mapping update. For example, with XNU it is necessary to trigger virtual memory fault twice when the page is not present. Similarly, the protection restoration routine invoked from the trampoline preserves the registers and calls the write handler.

As a result, it was possible to emulate all the existing SMC protocols at no issue and avoid the use of the original device.

7. Conclusion

Emulating peripheral devices within the existing operating system is not a new problem. Different solutions and approaches have appeared over the years. The industrial demand for full-stack operating system virtualization brought their performance to a completely different level, and the needs for better customization resulted in operating system developers providing more flexible interfaces with the possibility to create virtual hardware out of the box. Programmable microcontrollers made the process of building a device clone with the necessary features a much simpler task to accomplish.

However, there are numerous cases, where in-kernel peripheral emulation is highly anticipated, such as driver development needs, testing and verification, hardware migration, security analysis, etc. As we stated, it is often not possible or extremely impractical to attempt to incorporate virtual machines due to development costs or performance penalties. While virtual machines succeed in emulating CPUs of the same architecture at almost the same speed with hardware assisted virtualization, the performance of other CPUs without the use of JITs, commonly used in video game console emulators but rarely found in generic virtualization software, is often much worse. And in terms of I/O emulation, which is the primary concern of this paper, the situation is no better.

Furthermore, all the solutions heavily depend on the target architecture. While it was possible to think of x86 as the main architecture for personal computers in the beginning of 2000- s, today the concept of personal computers has shifted away, and other major players, e.g. ARM, appeared on the market. With this in mind the classical approach to virtualizing the whole operating system could face severe issues in the future.

The idea of using page protection faults to handle device I/O events without a hypervisor may be known but not widespread anywhere out of I/O tracing. In this paper we described a way to implement a complete MMIO protocol emulator in the kernel with the use of a generic approach that has few dependencies on the target architecture and relies on platform features such as MMU and paging. We showed that certain target architecture capabilities and device protocol specifics may affect the implementation, and effectively allow or disallow a broad range of optimizations. We believe that a suggested device I/O protocol model is applicable to various hardware, and give examples on how to simplify and optimize its implementation. After exploring the existing hardware, we built a SMC emulator in the XNU kernel to illustrate the suggested approach.

Acknowledgements

The authors thank ISP RAS and SYRCoSE staff for review and comment, Nikita Golovliov for aid in SMC emulator development, and Marvin Häuser for reverse-engineering Apple SMC UEFI drivers.

References

[1]. Jeremy Sugerman, Ganesh Venkitachalam, Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In Proceedings of the General Track: 2001 USENIX Annual Technical Conference, 2001, pp. 1-14. Available at: http://static.usenix.org/legacy/publications/library/proceedings/usenix01/sugerman/sugerman.ps, accessed 12.06.18

[2]. Keith Adams, Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006, pp. 2-13. Available at: https://www.vmware. com/pdf/asplos235 adams.pdf, accessed 09.06.18

[3]. Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In Proceedings of the 2014 USENIX Annual Technical Conference, 2014, pp. 109-120. Available at: https://www.usenix.org/system/files/conference/ atc14/atc14-paper-suzuki.pdf, accessed 12.06.18

[4]. Hangchen Yu, Christopher J. Rossbach. Full Virtualization for GPUs Reconsidered. In Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking, 2017.

[5]. Konstantinos Menychtas, Kai Shen, Michael L. Scott. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In Proceedings of the 2013 USENIX conference on Annual Technical Conference, 2013, pp. 291-296. Available at: https://www.usenix.org/system/files/ conference/atc13/atc13-menychtas.pdf, accessed accessed 12.06.18

[6]. Unified EFI, Inc. Platform Initialization (PI) Specification. Version 1.6. 2017. Available at: http://www.uefi.org/sites/default/files/resources/PI Spec 1 6.pdf, accessed 09.06.18

[7]. Jeff Muizelaar, Pekka Paalanen. In-kernel memory-mapped I/O tracing Available at: https://www.kernel.org/doc/Documentation/trace/mmiotrace.txt, accessed 12.06.18

[8]. Arm Holdings. ARM1176JZ-S Technical Reference Manual. Available at: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/ Caceaije.html, accessed 12.06.18

[9]. NXP Semiconductors. e500mc Core Reference Manual. Available at: http://cache. freescale.com/files/32bit/doc/ref manual/E500MCRM.pdf, accessed 09.06.18

[10]. Intel. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Available at: http://www.ece.cmu.edu/_ece845/sp17/ docs/vt-overview-itj06.pdf, accessed 12.06.18

[11]. BeaEngine. Length Disassembler Engine for Intel 64-bit processors. Available at: https://github.com/BeaEngine/lde64, accessed 12.06.18

[12]. CupertinoNet. EfiPkg, AppleSmcIo protocol. Available at: https://github.com/ CupertinoNet/EfiPkg, accessed 12.06.18

[13]. Crowdstrike. Alex Ionescu. ”Spell”unking in Apple SMC Land. 2013. Available at: http://www.nosuchcon.org/talks/2013/D1 02 Alex Ninjas and Harry Potter.pdf, accessed 09.06.18

Эмуляция ввода-вывода оборудования с отображением в ОЗУ внутри ядер операционных систем

[1,4] *В. Ю. Чепцов <cheptsov@ispras.ru>*
[1,2,3,4] *А. В. Хорошилов <khoroshilov@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25;
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1,
[3]*Московский физико-технический институт,*
141700, Московская область, г. Долгопрудный, Институтский пер., 9
[4] *Высшая школа экономики,*
101000, Россия, г. Москва, ул. Мясницкая, д. 20.

Аннотация. Необходимость эмуляции оборудования часто возникает на различных стадиях цикла разработки, миграции оборудования или обратной разработки. Реализация алгоритмов, связанных с конкретным устройством, сама по себе является нетривиальной задачей, но интеграция эмулятора с существующей средой, например, драйверами, предназначенными для работы с реальным оборудованием, зачастую оказывается не менее сложной. Устройства, полагающиеся на ввод-вывод с отображением в оперативную память, представляют особый интерес, так как в этих случаях, в отличие от использования портов ввода-вывода, гораздо меньше вероятность, что целевая платформа предоставит интерфейс для перехвата операций. Один из распространённых подходов, широко используемый в ПО виртуальных машин, состоит в том, чтобы поместить всю операционную систему под гипервизор и создать внешний эмулятор. Однако это может быть нежелательно по причинам сложности гипервизора, потери производительности, дополнительных требований к аппаратному обеспечению и пр. В данной статье такой подход распространяется на ядро, и предлагается описание возможности построить эмулятор, прибегая лишь к существующим интерфейсам, предоставляемым операционной системой. Ввиду частой доступности MMU и механизмов защиты страниц, позволяющих перехватывать доступ записи и чтения, предполагается, что предлагаемый подход может быть использован на значительном количестве целевых платформ. В статье приводится подробное рассмотрение проблем, возникающих при написании конкретной реализации, и приводятся способы её упрощения и оптимизации в зависимости от возможностей целевой платформы, эмулируемого протокола и иных требований к задаче. В качестве экспериментального доказательства работоспособности предлагаемого подхода приводится реализация эмулятора SMC для платформы x86.

Ключевые слова: эмуляция оборудования; ввод-вывод с отображением в ОЗУ; модули ядра

DOI: 10.15514/ISPRAS-2018-30(3)-9

Для цитирования: Чепцов В.Ю., Хорошилов А.В. Эмуляция ввода-вывода оборудования с отображением в ОЗУ внутри ядер операционных систем. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 121-134 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-9

Список литературы

[1]. Jeremy Sugerman, Ganesh Venkitachalam, Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In Proceedings of the General Track: 2001 USENIX Annual Technical Conference, 2001, pp. 1-14. Режим доступа: http://static.usenix.org/legacy/publications/library/proceedings/usenix01/sugerman/sugerman.ps, дата обращения 12.06.18

[2]. Keith Adams, Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006, pp. 2-13. Режим доступа: https://www.vmware. com/pdf/asplos235 adams.pdf, дата обращения 09.06.18

[3]. Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In Proceedings of the 2014 USENIX Annual Technical Conference, 2014, pp. 109-120. Режим доступа: https://www.usenix.org/system/files/conference/ atc14/atc14-paper-suzuki.pdf, дата обращения 12.06.18

[4]. Hangchen Yu, Christopher J. Rossbach. Full Virtualization for GPUs Reconsidered. In Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking, 2017.

[5]. Konstantinos Menychtas, Kai Shen, Michael L. Scott. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In Proceedings of the 2013 USENIX conference on Annual Technical Conference, 2013, pp. 291-296. Режим доступа: https://www.usenix.org/system/files/ conference/atc13/atc13-menychtas.pdf, дата обращения 12.06.18

[6]. Unified EFI, Inc. Platform Initialization (PI) Specification. Version 1.6. 2017. Режим доступа: http://www.uefi.org/sites/default/files/resources/PI Spec 1 6.pdf, дата обращения 09.06.18

[7]. Jeff Muizelaar, Pekka Paalanen. In-kernel memory-mapped I/O tracing Режим доступа: https://www.kernel.org/doc/Documentation/trace/mmiotrace.txt, дата обращения 12.06.18

[8]. Arm Holdings. ARM1176JZ-S Technical Reference Manual. Режим доступа: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/ Caceaije.html, дата обращения 12.06.18

[9]. NXP Semiconductors. e500mc Core Reference Manual. Режим доступа: http://cache. freescale.com/files/32bit/doc/ref manual/E500MCRM.pdf, дата обращения 09.06.18

[10]. Intel. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Режим доступа: http://www.ece.cmu.edu/_ece845/sp17/ docs/vt-overview-itj06.pdf, дата обращения 12.06.18

[11]. BeaEngine. Length Disassembler Engine for Intel 64-bit processors. Режим доступа: https://github.com/BeaEngine/lde64, дата обращения 12.06.18

[12]. CupertinoNet. EfiPkg, AppleSmcIo protocol. Режим доступа: https://github.com/ CupertinoNet/EfiPkg, дата обращения 12.06.18

[13]. Crowdstrike. Alex Ionescu. ”Spell”unking in Apple SMC Land. 2013. Режим доступа: http://www.nosuchcon.org/talks/2013/D1 02 Alex Ninjas and Harry Potter.pdf, дата обращения 09.06.18