Streng gekapselt zu mehr Sicherheit
Vom Umgang mit Hypervisoren in Embedded-Systemen

Jens Braunes, PLS Programmierbare Logik & Systeme GmbH

Für die Realisierung sicherheitskritischer Anwendungen ist eine strikte Trennung von Applikationen oder Betriebssystemen, die sich eine gemeinsame Rechnerplattform teilen, unabdingbar. Deshalb rückt das Thema Virtualisierung und „Hypervisor“ auch im Embedded-Bereich immer mehr in den Vordergrund. Eine große Herausforderung vor allem für Entwickler, die sich in einem sehr hardwarenahen Umfeld bewegen.

Virtualisierung an sich ist nichts Neues. Im PC- und Serverumfeld wird sie schon seit vielen Jahren erfolgreich eingesetzt. Im Bereich der Embedded-Systeme wurde dem Thema bislang allerdings relativ wenig Beachtung geschenkt. Nachdem mit den immer leistungsfähigeren Multicore-SoCs mittlerweile genügend Performance zur Verfügung steht, um verschiedene Applikationen und Betriebssysteme parallel betreiben zu können, zeichnet sich hier inzwischen allerdings eine Trendwende ab.

Für Virtualisierungen im Embedded-Bereich gibt es grob unterteilt zwei grundlegende Motivationen: die Trennung echtzeitkritischer von weniger zeitkritischen Anwendungen auf einer gemeinsamen Rechnerplattform und die Gewährleistung der Sicherheit. Hierbei geht es hauptsächlich darum, weniger sicherheitskritische Anwendungen von sicherheitskritischen Anwendungen zu trennen. Verständlicherweise ist gerade letzteres Thema mit größter Umsicht zu behandeln. Vorfälle wie der Chrysler-Hack aus dem Jahr 2015 [1], bei dem Sicherheitsforscher über das Internet und eine Schwachstelle im Infotainment-System Zugriff auf so sicherheitskritische Fahrzeugsysteme wie die Bremse erhalten haben, führen eindrucksvoll vor Augen, dass eine strikte Kapselung von Funktionalität aus Sicherheitsgründen unabdingbar ist. Dies gilt nicht nur für Automotive-Systeme, sondern auch für andere stark vernetzte Anwendungen beispielsweise im IoT- oder SmartHome-Bereich

Trennung für mehr Sicherheit

Sollen mehrere Betriebssysteme – im nachfolgenden als Gastbetriebssysteme oder Gast bezeichnet – oder einzelne Bare-Metal-Anwendungen virtualisiert werden, benötigt man einen Hypervisor, der sich als Abstraktionsschicht über die reale Hardware, also die Prozessorkerne, die Speicher und nicht zuletzt die Peripherals schiebt. Dabei ist grundlegend zwischen Typ 1- und Typ 2- Hypervisoren zu unterscheiden (Abbildung 1).

Der Typ-1-Hypervisor setzt direkt auf der Hardware auf und wird aus diesem Grund auch Bare-Metal-Hypervisor genannt. Er benötigt kein separates Betriebssystem, muss aber folglich alle Treiber für die Hardware selbst bereitstellen.

Beim Typ-2-Hypervisor wird ein Host-Betriebssystem benötigt, welches die Gerätetreiber bereitstellt. Zwar können bei dieser Variante neben dem Hypervisor auch Applikationen laufen. Für embedded Systeme ist diese Variante aber eher uninteressant, weil man den Hypervisor so schlank wie möglich halten möchte und vor allem eine starke Separierung der einzelnen Gastbetriebssysteme oder Applikationen erreichen will. Konzentrieren wir also im Weiteren auf Funktion und Anwendung von Typ-1-Hypervisoren.

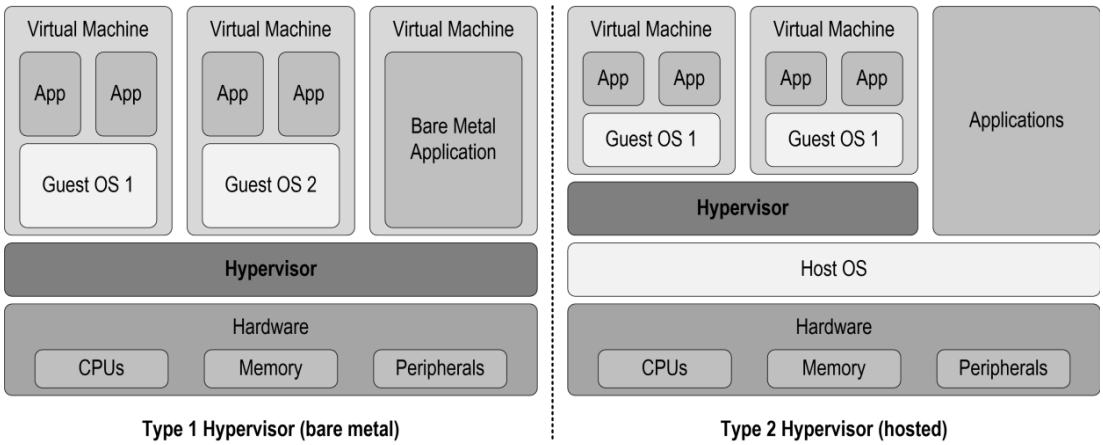

Abbildung 1: Klassifizierung der Hypervisor-Ansätze

Der Typ-1-Hypervisor übernimmt neben grundlegenden Betriebssystemaufgaben wie die Verwaltung der Hardware-Ressourcen vor allem die Zuordnung, welches Gastbetriebssystem innerhalb einer virtuellen Maschine (VM) auf welchen physischen Rechenkernen ausgeführt wird und wie deren Scheduling erfolgen soll. Die Gastbetriebssysteme sehen die physischen Cores nicht, vielmehr werden ihnen durch die jeweilige VM virtuelle Cores zur Verfügung gestellt. Dabei ist es nicht zwingend, dass die Gesamtzahl der virtuellen Cores mit der Anzahl physischer Cores übereinstimmen muss. Mehrere Gastbetriebssysteme können sich durchaus einen oder mehrere physische Cores teilen. In diesem Fall muss der Hypervisor allerdings ein Scheduling durchführen. Jeder Gast bekommt eine Zeitspanne für die Abarbeitung auf den Cores zur Verfügung gestellt. Beim Wechsel des aktiven Gastbetriebssystems muss der Verarbeitungskontext, also Core-Register, Systemregister, Status der Interrupt-Abarbeitung, etc., gesichert und wiederhergestellt werden. Dies ist vergleichbar mit dem Taskwechsel in einem Betriebssystem.

Bei der Virtualisierung von Echtzeitbetriebssystemen werden die virtuellen den physischen Cores typischerweise statisch zugeordnet. Der Entwickler entscheidet also selbst, wie sich die Verarbeitungslasten der einzelnen Gastbetriebssysteme auf die Prozessorressourcen, sprich auf die Kerne, verteilen. Damit wird ein deterministisches Verhalten des Gesamtsystems garantiert. Ob die Echtzeitfähigkeit bezüglich des zuzusichernden Zeitverhaltens auch wirklich erreicht wird, ist zwar in großem Maße von der gewählten Zuordnung und Verteilung abhängig, hängt jedoch natürlich auch von der Performance der eingesetzten Prozessorplattform ab.

Die Aufteilung der Gastbetriebssysteme auf verschiedene Cores erfolgt zusammen mit weiteren Konfigurationsinformationen wie beispielsweise der Zuordnung, in welche physikalischen Speicherbereiche die Gastbetriebssysteme geladen werden sollen oder welche Peripherals ihnen zur Verfügung stehen und wird häufig fest in den Hypervisor einkompiliert bzw. gelinkt. Gleiches gilt auch für die Binaries der virtuellen Maschinen und die Gastbetriebssysteme. Zwar könnte dafür auch auf externe Files zurückgegriffen werden, die sich beispielsweise auf einem Massenspeicher oder auf einem Netzlaufwerk befinden. Doch so ein Vorgehen birgt natürlich immer auch ein gewisses Risiko hinsichtlich Integrität und Vertrauenswürdigkeit der zu ladenden virtuellen Maschinen. Wenn soweit alles geklappt hat, steht am Ende ein aus allen Gastbetriebssystemen und dem Hypervisor bestehendes monolithisches

Binary bereit, um vom Bootloader in den Speicher des Zielsystems geladen und letztendlich gestartet zu werden.

Nicht ohne geeignete Hardware
Für eine wirklich sichere Abgrenzung der Gastbetriebssysteme muss der Hypervisor freilich auf einige wichtige Hardwarefunktionen des Zielprozessors zurückgreifen können. Schauen wir uns diese zwingend erforderlichen Features am konkreten Beispiel eines ARM Coretx-A53 einmal näher an. Dieses SoC bietet mit seinen bis zu vier Armv8-A Cores nicht nur die nötige Rechenleistung; der auf dem Chip integrierte *Hardware Virtualization Support* stellt gleichzeitig auch alle notwendigen Hardwarefunktionen für den Hypervisor zur Verfügung:

- vier Exception Levels (EL0 bis EL3), wobei EL2 explizit dem Hypervisor vorbehalten ist
- eine Memory Management Unit (MMU) mit zweistufiger Adressumrechnung
- Unterstützung für Device Emulation
- exklusive Zuweisung von physischen Devices zu einer bestimmten virtuellen Maschine
- Weiterleitung von Exceptions und virtuelle Interrupts

Die Exception Level der Armv8-A Architektur legen die Privilegierung der aktuellen Software-Ausführung fest. Je höher das Exception Level, desto höher privilegiert ist die Ausführung. Konsequenterweise sieht die Armv8-A Architektur deshalb für den Hypervisor einen eigenen Exception Level (EL2) vor, um ihn sicher von den Gastbetriebssystemen (EL1 und EL0) zu separieren [2].

Damit die Gastbetriebssysteme den physikalischen Speicher des Cortex-A53 nutzen können, ohne sich der Existenz des Hypervisors bewusst zu sein und ohne unerlaubt auf dessen Speicher zuzugreifen, ist eine zweistufige Adressumrechnung in Form einer MMU implementiert. Der physikalische Speicher, den das Gastbetriebssystem zu sehen glaubt, ist eigentlich die sognannte *Intermediate Physical Address Map*. Diese wiederum liegt in der Verantwortung des Hypervisors. Nach der Übersetzung der virtuellen Adresse durch das Gastbetriebssystem in die *Intermediate Physical Address* ist also noch eine weitere Übersetzung in die eigentliche physische Adresse notwendig (Abbildung 2).

Über Device Emulation oder Device Assignment erlangen die Gastbetriebssysteme Zugang zu den Hardware Devices bzw. Peripherals. Device Emulation ist dann notwendig, wenn ein Device für mehr als nur für einen Gast verfügbar sein soll. Ein direkter Zugriff auf das Hardware-Device ist in diesem Fall nicht möglich, da sonst Konflikte vorprogrammiert sind. Im Hypervisor werden die Devices deshalb in Software so nachgebildet, dass auch Zugriffe durch mehre Gastbetriebssysteme behandelt werden können. Greift ein Gast auf ein solches in den Speicher eingeblendetes emuliertes Device zu, resultiert dies in einen Trap in den Hypervisor (EL2) hinein. Dieser muss nun entsprechend reagieren und gegebenenfalls das reale Hardware Device ansprechen. In entgegengesetzter Richtung, wenn also der Gast mit Daten aus dem Device versorgt werden soll, werden virtuelle Interrupts durch den Hypervisor ausgelöst. Aus Sicht der Gastbetriebssysteme sehen diese wie normale Hardware Interrupts aus. Auch der Interrupt Controller steht als virtuelles Device zur Verfügung. Reagiert der Gast dann mit einem Lesezugriff auf den Speicher oder auf

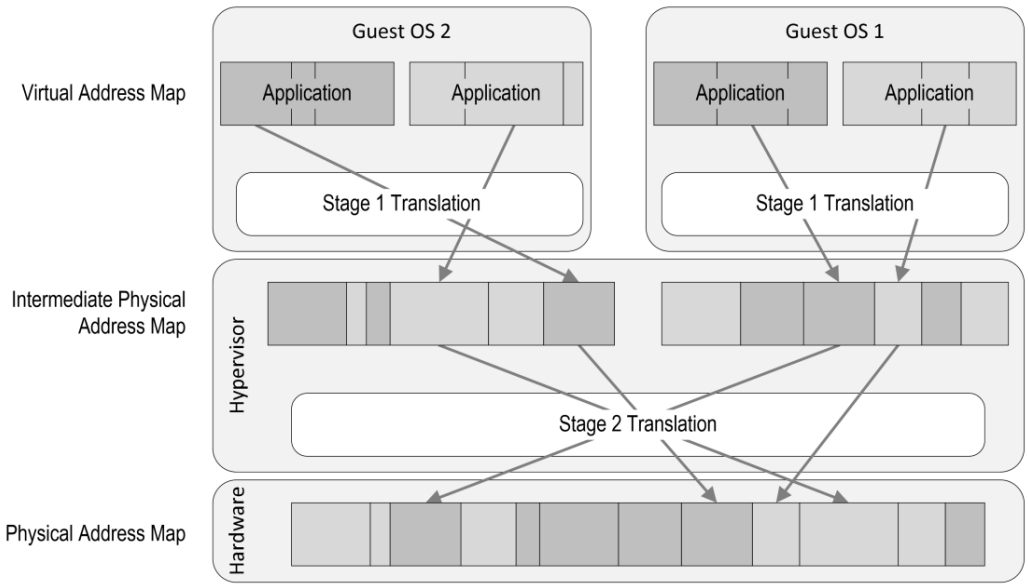

Abbildung 2: Zweistufiger Adressumrechnung

Register des Devices, wird wie bereits beschrieben ein Trap in den Hypervisor ausgelöst. Dieser stellt dann die angefragten Daten zur Verfügung.
Der Hypervisor kann selbstverständlich auch Devices vor den Gastbetriebssystemen komplett verbergen oder nur für einen Gast exklusiv verfügbar machen. Letzteres erlaubt die direkte Nutzung von Hardware Devices. Lediglich eine Adressumrechnung und die Anpassung der Interrupt-IDs durch den Hypervisor ist nötig. Der große Vorteil dabei ist natürlich, dass man sich den Overhead der Device Emulation spart.

Auch für Interrupts bzw. Exceptions gilt, dass diese aus Sicht der Gastbetriebssysteme direkt aus der Hardware kommen. Dem ist aber nicht so. Vielmehr werden Hardware Exceptions oder Interrupts durch den Hypervisor bearbeitet und gegebenenfalls als virtuelle Exceptions bzw. virtuelle Interrupts an den jeweiligen Gast weitergereicht.

Hypervisor Awareness in der Entwicklung –
Herausforderungen beim Debugging
Die Präsenz eines Hypervisors wirkt sich auch auf das Debugging aus, wobei es hier mehrere Sichtweisen und Aufgabenstellungen zu beachten gilt. Eine besondere Herausforderung stellt vor allem das hardwarenahe Debugging dar, bei dem der Entwickler naturgemäß immer mit dem Hypervisor bzw. den VMs in Berührung kommt:

1. **Entwicklung einer virtualisierten Bare-Metal Applikation**
   Die Zugriffe auf Speicher und Peripherals werden nicht durch ein Betriebssystem gekapselt, stattdessen wird direkt auf die virtualisierte Hardware zugegriffen. Für das Debugging sind also Speicherinhalte und Zugriffe auf die Device Register interessant.
2. **Treiberentwicklung für ein Gastbetriebssystem**
   Dieser Anwendungsfall ist im Großen und Ganzen identisch mit Punkt 1.
3. **Laufzeitanalyse des Gesamtsystems**
   Hierbei geht es vorrangig um das Messen von Laufzeiten und der Lastverteilung. Ziel dieser Messungen ist eine Optimierung der Hypervisor-

Konfiguration, damit beispielsweise ein zugesichertes Zeitverhalten eines Echtzeitbetriebssystem als Gast auch tatsächlich eingehalten wird.

Auf das reine Debugging von Applikationen, die unter einem Gastbetriebssystem laufen, gehe ich an dieser Stelle ganz bewusst nicht näher ein, weil hier ja eher selten direktes Hardware-Debugging betrieben wird. Schauen wir uns stattdessen die drei oben genannten Anwendungen noch etwas genauer an.

Während sich der Entwickler beim letzteren Anwendungsfall explizit mit dem Hypervisor auseinander setzt, sollte ihm in den ersten beiden Fällen die Tatsache, dass es sich um eine Virtualisierung handelt, eigentlich verborgen bleiben. In der Praxis sieht dies jedoch meist anders aus. Setzt man zum Debuggen einen JTAG-Debugger wie die Universal Debug Engine (UDE) von PLS ein, dann hat dieser direkten Zugang zum Speicher, zu den Registern und zu den Prozessorkernen des Zielprozessors; also wenn man so will zu allem, was unterhalb des Hypervisors angesiedelt ist. Die Sicht der Bare-Metal-Applikation oder des Treibers, der entwickelt werden soll, endet aber oberhalb des Hypervisors. Für diese sind nämlich nur die virtuellen Hardware-Komponenten, also virtualisierter Speicher, emulierte Devices und virtuelle Prozessorkerne sichtbar. Das hat in der Praxis natürlich Konsequenzen.

Die Hypervisor-Awareness des Debuggers erlaubt zwar prinzipiell eine Umrechnung der virtuellen und Intermediate-Physical-Adressen in physische Adressen, allerdings funktioniert das nur bei realem Speicher (z.B. RAM oder FLASH) nicht aber bei emulierten Devices. Diese sind zwar in den virtuellen Speicher eingeblendet, haben aber keinen ihnen zugeordneten physischen Speicherbereich. Wenn ein Gast auf diesen Device-Speicher zugreift, ist das natürlich nicht problematisch. Es folgt wie weiter oben beschrieben ein Trap in den Hypervisor und die Verarbeitung durch die Device-Emulation. Die Adressübersetzung durch den Debugger jedoch liefert einen ungültigen Speicherbereich zurück, weil er die Emulation des Devices nicht kennt. Die Verwendung von emulierten Devices innerhalb der virtuellen Maschine ist also nicht direkt debugbar. Das gilt natürlich nicht bei Devices, die direkt und exklusiv einer VM zugeordnet. Hier existiert reale Hardware, auf die der Debugger Zugriff hat.

Stoppt der Debugger an einem Breakpoint oder direkt von Anwender ausgelöst das System, werden neben allen physischen Cores auch der gesamte Hypervisor und somit auch alle virtuellen Maschinen angehalten. Das ist durchaus sinnvoll und konsequent, denn andernfalls müsste sich der Hypervisor plötzlich mit einer virtuellen Maschine auseinandersetzen, die nicht mehr reagiert. Er käme unweigerlich außer Tritt und würde womöglich abstürzen.

Anders sieht es aus, wenn der Hypervisor einen Debug-Monitor anbietet. Dann ist er selbst in der Lage und dafür verantwortlich, einzelne virtuelle Maschinen und deren Gastbetriebssysteme anzuhalten, während die anderen weiterlaufen können. Wenn der Debugger diesen Debug-Monitor nutzt, darf er allerdings selbstverständlich keinen direkten Einfluss auf die Hardware mehr nehmen.

Das Setzen eines Breakpoints innerhalb des Kontextes einer virtuellen Maschine bei laufendem System erfordert vom Debugger einiges an Aufwand und lässt sich leider auch nicht vollständig reibungsfrei realisieren. Der Debugger muss dafür nämlich

erst einmal den aktuellen Zustand des Gesamtsystems ermitteln, also welche virtuellen Maschinen gerade aktiv sind und welche nicht. Wirklich sicher funktioniert das jedoch nur im angehaltenen Zustand, weshalb der Debugger auch kurz das gesamte System anhält. Zwar bleibt dieser kurze Zwischenstopp dem Nutzer weitestgehend verborgen auf das Laufzeitverhalten der einzelnen virtuellen Maschinen hat er aber natürlich schon Einfluss.
Um das Scheduling des Hypervisors zu optimieren und feststellen zu können, ob Zusicherungen von Echtzeitverarbeitung von Gastbetriebssystemen eingehalten werden, ist es notwendig, Laufzeitmessungen durchzuführen. Im einfachsten Fall liefert der Hypervisor diese Informationen selbst und der Debugger kann den Debug-Monitor dazu befragen. Wenn nicht, bleibt als Alternative eigentlich nur Trace. Letzteres erfordert jedoch sowohl vom Debugger als auch vom Anwender eine genaue Kenntnis der Verwaltungsstrukturen des Hypervisors, um die aufgezeichneten Trace-Daten nutzbringend auswerten zu können.

Fazit
Ganz ohne Virtualisierung wird eine strikte Trennung von sicherheitskritischer oder echtzeitkritischer Software auf einer gemeinsamen Embedded-Prozessorplattform künftig kaum zu bewältigen sein. Allerdings gilt es hierbei zu berücksichtigen, dass vor allem das hardwarenahe Debuggen von Treibern oder Bare-Metal-Applikationen in einer virtualisierten Umgebung auch unerwünschte Effekte für das Gesamtsystem mit sich bringen kann. Zwar sind die Hersteller von Debug-Werkzeugen bestrebt, solche prinzipbedingten Rückwirkungen so gering wie möglich zu halten, gänzlich verhindern lassen sie sich jedoch leider nicht. Nur wer sich dessen vorab bewusst ist, wird unerwartete Effekte während der Systementwicklung auch richtig einordnen können.

Referenzen

[1] https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
[2] ARM Ltd: *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*

Autor
Jens Braunes ist Product Marketing Manager bei der PLS Programmierbare Logik & Systeme GmbH. Er studierte Informatik an der TU Dresden und arbeitete dort als wissenschaftlicher Mitarbeiter. 2005 wechselte er zum Softwareteam von PLS und ist dort maßgeblich an der Entwicklung der Universal Debug Engine beteiligt. Er erweiterte 2016 sein Tätigkeitsfeld auf das Produktmanagement und technische Marketing. Jens Braunes ist regelmäßig als Autor von Fachartikeln und als Referent auf Kongressen tätig.

Kontakt
Internet: www.pls-mc.com
Email: jens.braunes@pls-mc.com
Tel. +49 35722-384-0
PLS Programmierbare Logik & Systeme GmbH
Technologiepark,02991 Lauta