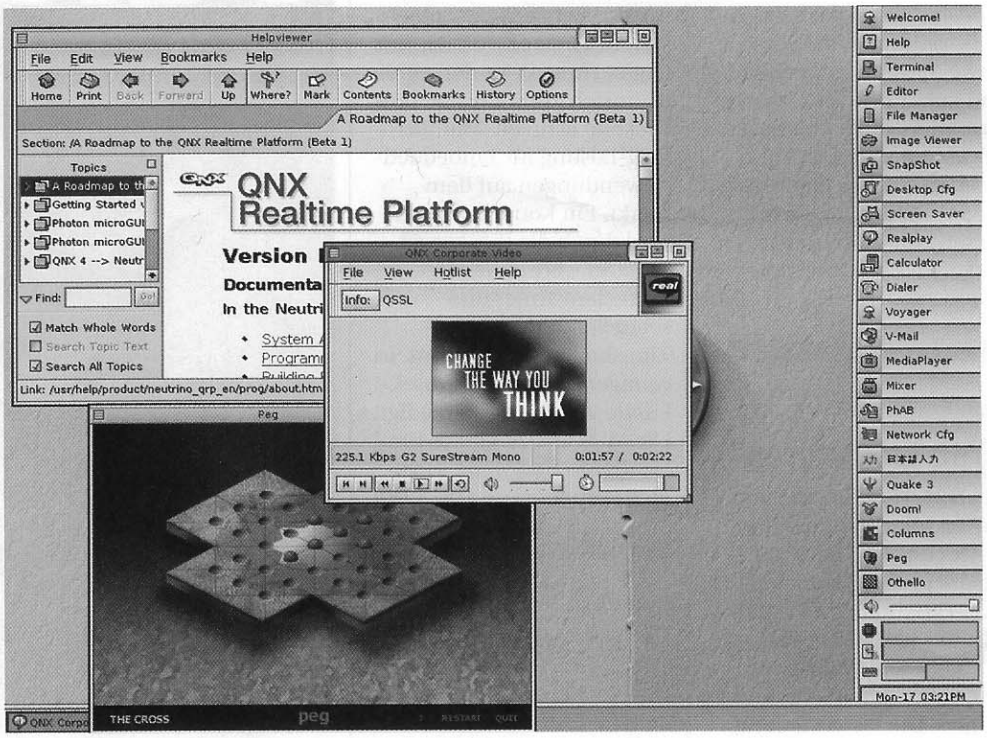


Mehr Funktionalität

Beschleunigung der Software-Entwicklung bei gleichzeitiger Steigerung der Produktqualität

Mehr Funktionalität heißt das Stichwort in der Embedded-Softwareentwicklung. Nicht nur bei Geräten der Post-PC-Ära wie PDA oder Set-Top-Box ist dieser Trend bemerkbar, sondern auch bei Kfz-Fahrerinformationssystemen. Die meisten Betriebssysteme sind für einen schnellen Zuwachs an neuen Features nicht ausgelegt. Das Problem hat seine Ursache im traditionellen Embedded-Design, wo die Software während des kompletten Produktlebenszyklus nur wenig oder gar nicht geändert wird.

*Matthias Stumpf**

Gerade in der Automobilbranche wird der Wettbewerb Hersteller dazu zwingen, Systeme mit immer mehr Features in immer kürzeren Abständen einzuführen. Kfz-Fahrerinformationssysteme, die Navigation, Spracherkennung, drahtlose Kommunikation, Web Zugang, E-Commerce und andere Online Services kombinieren, werden in den nächsten Monaten zum Standard werden. Für Fahrzeughersteller und Zulieferer bedeutet dies jedoch, dass sie über kurz oder lang vor dem selben Problem stehen wie die Designer anderer Information Appliances: Die Betriebssysteme machen diese Entwicklung nicht mehr mit.

Das bedeutet, dass sich die Struktur von Betriebssystemen ändern muss, um heutigen Anforderungen von seiten der Entwicklung zu entsprechen und damit die Time-to-Market zu verkürzen. Ziel ist es, den verschiedensten Branchen die Möglichkeit zu eröffnen, Applikationen „von der Stange" anbieten zu können. Installierte Systeme dürfen nicht mehr „geschlossen" sein, sondern „offen", um sie so einfach wie möglich pflegen, aktualisieren oder erweitern zu können. Denn Features wie Web-Zugang mit XML oder Multimedia Plug-Ins sind oft mit einem Upgrade versehbar, wenn die Information Appliances nicht bereits wenige Monate nach dem Kauf veraltet sein sollen.

Viele kommerziell und firmenintern entwickelten Embedded-Betriebssysteme verwenden eine traditionelle „flache" Architektur, d.h. mit

*Matthias Stumpf ist European Manager Corporate Communications bei QNX Software Systems, Hannover.

wenig oder gar keinen Hierarchien. Wie in Bild 1 dargestellt, werden bei dieser Architektur alle Softwaremodule in denselben Adressraum gelegt wie der Betriebssystemkernel; es besteht absolut kein Speicherschutz. Das führt dazu, dass jedes noch so triviale Modul den vom Kernel verwendeten Speicher überschreiben kann – und somit das ganze System zum Absturz bringt. Es reicht schon ein einziger Programmierfehler – etwa ein ungültiger C-Pointer.

Flache Architektur: Engpass bei der Entwicklung

Diese Architektur lässt offensichtlich wenig Spielraum für Fehler zu. Bei einem einfachen Embedded-Design ist das kein Problem, denn die meisten Probleme können dort bei der Integrationsprüfung ermittelt werden. Was aber, wenn das Projekt weiter wächst und 50, 100 oder 1000 Module umfasst? Mit Tausenden von Programmcodezeilen und Hunderten von aktiven Threads. Wo mit der Suche beginnen, wenn das System von einem verirrten Pointer zum Absturz gebracht wird? Selbst mit den besten Werkzeugen kann nicht immer das fehlerhafte Modul ermittelt werden. Wurde der Fehler behoben, muss die komplette Software wahrscheinlich neu getestet werden. Der Grund ist einfach: Wird unter einer „flachen" Architektur eine Programmcodeänderung durchgeführt, muss das komplette Abbild des Moduls im Speicher neu gelinkt werden, wodurch ein neues Abbild mit anderen Speicheradressen-Offsets gebildet wird. Ein Modul, das bisher einen unbenutzten Datenbereich überschrieben hat, könnte nun einen kritischen Datenbereich überschreiben, der von einem anderen Modul – eventuell sogar dem Betriebssystemkernel – verwendet wird. Dieses Problem kann durch Hinzufügen einer einzigen Programmcodezeile entstehen.

Durch eine „flache" Architektur ist der Test neuer Design-Ideen eingeschränkt. Denn jedesmal wenn der Programmcode geändert wird, muss das komplette Abbild des Moduls im Speicher neu aufgebaut werden. Und in größeren Anwendungen kann dieser Neuaufbau mehrere Stunden dauern. Darüber hinaus kann sich die Fehlersuche so zeitaufwendig gestalten, dass das QS-Team gezwungen wird, instabilen Programmcode freizugeben.

Weshalb aber arbeiten noch so viele Betriebssysteme mit dieser Architektur? Der Grund lässt sich historisch erklären: Bis vor kurzem verfügten die meisten Embedded-Prozessoren noch über keine integrierte MMU. Ein weiterer Grund ist die Leistung: Viele kommerzielle und firmeninterne Betriebssystementwickler haben Schwierigkeiten, den zusätzlichen Overhead der MMU zu unterstützen. Manche behaupten sogar, dass Speicherschutz aus Leistungsgründen geopfert werden muss. Dies entspricht jedoch nicht unbedingt den Tatsachen. Wie später noch zu sehen ist, kann ein gut ausgelegtes Betriebssystem nicht nur umfangreichen MMU-Schutz gewähren, sondern eine Leistung bieten, die einer konventionellen „flachen" Betriebssystemarchitektur entspricht oder sie übertrifft.

Monolithische Architektur: Die halbe Miete

Bei dem Versuch, das Problem der „flachen" Architektur anzusprechen, greifen einige Programmierer von Embedded-Betriebssystemen auf eine monolithische Kernelarchitektur zurück (Bild 2). Bei dieser Architektur läuft jedes Anwendungsmodul in einem eigenen speichergeschützten Adressraum. Versucht eine

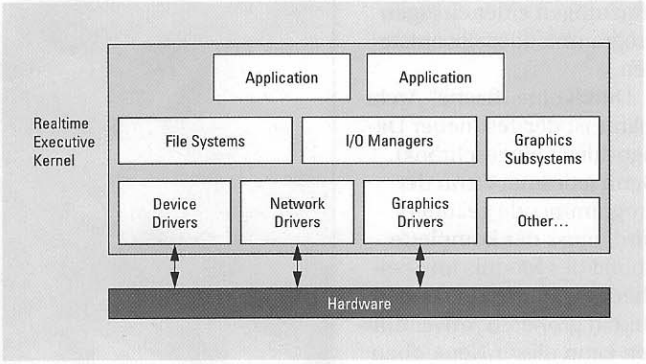


Bild 1: Eine „flache" Architektur bietet keinen Speicherschutz

Anwendung den von einem anderen Modul genutzten Speicher zu überschreiben, fängt die MMU den Fehler ab, sodass der Entwickler die fehlerhafte Stelle erkennt. Auf den ersten Blick sieht das gut aus. Der Entwickler muss nicht länger in Sackgassen nach subtilen Fehlern im Anwendungscode suchen. Alle „low-level"-orientierten Module wie Dateisysteme, Protokollstacks oder Treiber bleiben mit demselben Adressraum wie der Kernel verbunden. Eine einzige Speicherschutzverletzung in nur einem einzigen Treiber kann das System immer noch zum Absturz bringen, wobei eine nur schwache oder gar keine Spur des Fehlers übrig bleibt.

Dies stellt ein echtes Problem dar, da Entwickler von Embedded-Systemen viel Zeit mit der Entwicklung von „low-level"-orientierten Modulen verbringen. So verfügt beispielsweise ein Switch einer Telekommunikationsanlage über einen großen Anteil an Protokollen und Treibern für kundenspezifische Hardware. Das führt dazu, dass die mit einer „flachen" Architektur einhergehenden Probleme den Entwickler immer noch plagen, Tage oder Wochen werden damit verbracht, korrupte C-Pointer aufzuspüren. Umfangreiche Prüfmaßnahmen sind für jede Programmcode-Änderung erforderlich und es besteht die Gefahr, dass sogar ein triviales Modul das System zum Absturz bringt.

Um diese Engpässe in der Entwicklung zu beseitigen, muss ein Betriebssystem die Architektur des Universal Prozess Modells implementieren. Wie aus Bild 3 zu erkennen ist, implementiert ein Universal Prozess Modell einen kleinen Satz essentieller Dienste innerhalb des Kernels selbst wie z.B. Scheduling, Interprozesskommunikation und die Umleitung von Hardware-Interrupts. Alle übrigen Systemdienste werden durch optional hinzufügbare Prozesse bereitgestellt. Das führt dazu, dass jeder Treiber, jedes Protokoll, jedes Dateisystem, jeder I/O-Manager und jedes Grafiksubsystem in seinem eigenen speichergeschützten Adressraum laufen kann.

UPM-Architektur: Hier endet der Fehler

Dass diese Architektur die Zuverlässigkeit erhöht, ist offensichtlich. Erstens enthält der Betriebssystemkernel nur wenig Programmcode, der Fehler verursachen könnte. Zweitens ist es sehr unwahrscheinlich, dass irgend ein Modul – auch ein schlecht programmiertes mit höchstem Zugriffsrecht – den Kernel korrumpieren könnte. Darüber hinaus läuft nun jedes Modul als unabhängiger Prozess, sodass sich jeder Teil des Softwaresystems beliebig starten, stoppen, verändern oder upgraden lässt, ohne einen Neustart durchführen oder den Kernel neu erstellen zu müssen. Wie lässt sich ein Entwicklungszyklus dadurch rationalisieren?

Interessant sind zunächst die Unterschiede beim Schreiben von Gerätetreibern, womit Entwickler von Embedded-Systemen viel Zeit verbringen. Bei einer „flachen" oder monolithischen Betriebssystemarchitektur ist sämtlicher Treibercode an den Kernel gebunden (Bild 4). Es ergibt sich folgendes Bild:

- Wird der Treiber verändert, muss gewöhnlich der Kernel neu kompiliert und das System neu gestartet werden?
- Wird der Kernel in einem Multi-Usersystem neu kompiliert, müssen sich alle anderen Entwickler ausloggen?
- Bei der Fehlersuche müssen „low-level"-orientierte Kernelwerkzeuge verwendet werden, statt einfacher zu bedienende Quellcode-orientierte Werkzeuge.
- Wenn eine Speicherkorruption auftritt, gibt es keine zuverlässige Möglichkeit sie genau zu bestimmen.

Nun der Vergleich mit der UPM-Architektur, bei der jeder Treiber als eigenständiger Prozess in einem separaten, speichergeschützten Adressraum läuft:

- Wird ein Treiber verändert, muss nur er neu kompiliert werden – eine Sache von wenigen Sekunden. Es ist nicht erforderlich den kompletten Kernel neu zu kompilieren.
- Da der Kernel niemals neu kompiliert wird, wird niemand ausgeloggt. Multiple Programmierer können das selbe Zielsystem gleichzeitig benutzen und Treiberprogrammierer dürfen nachts schlafen.
- Zur Fehlersuche in einem Treiber oder in praktisch jedem anderen traditionellen Kernelmodul lassen sich Quellcode-orientierte Testhilfen und Profilierungstools verwenden. Die Programmierung eines Treibers oder sogar eines kundenspezifisches Betriebssystems wird so einfach die einer Standardanwendung.
- Tritt eine Speicherverletzung auf, kann das Betriebssystem sofort das verant-

Wie sieht es mit der Leistung aus?

Die Vorteile der UPM bei der Entwicklung und während der Ausführungszeit wurden zwar aufgezeigt, doch bleibt die Frage: Hat die Tatsache, dass jede Anwendung, jeder Treiber und jedes Betriebssystemmodul in ein eigenes MMU-Segment gelegt wird, Nachteile für die Leistung? Die Antwort ist nein, nicht wenn das UPM richtig implementiert wurde: Beispielsweise kann ein UPM-Betriebssystem wie QNX eine Kontextumschaltung, das heißt die erforderliche Zeit zum Anhalten eines Prozesses und zum Neustart, auf einem 133-MHz-Pentium in nur 1,95 µs durchführen. Das ist mehr als schnell genug für praktisch jede leistungskritische Anwendung. *(hh)*

wortliche Modul identifizieren. Statt sich Wochen mit der Fehlersuche zu beschäftigen, ist das Problem in Minuten gelöst.

Weniger Aufwand durch Wiederverwendung

Wie bereits erwähnt, kann bei einem konventionellen Betriebssystem eine einzige Code-Korrektur für einen Treiber zu einer anderen Kernelabbildung führen, sodass umfangreiche Prüfungen erforderlich sind. Beim UPM umfasst der Kernel jedoch nur bestimmte essentielle Dienste, sodass derselbe Binärcode verwendet werden kann, der bereits vom Betriebssystemlieferanten laborgeprüft und von jedem Anwender feldgetestet worden ist (Bild 5). Darüber hinaus besitzt jedes Modul einen linearen virtuellen Adressraum, der mit 0 beginnt, sodass sogar der Binärcode jeder unveränderten Anwendung sowie jedes unveränderten Treibers, Betriebssystemmoduls und Protokollstacks wiederverwendbar ist. Das Ergebnis: Bei den meisten Codeänderungen müssen nur die betroffenen Module getestet werden, nicht die komplette Software.

Durch den geringeren Zeitaufwand für Prüfungen hat der Entwickler mehr Möglichkeiten neue Features hinzuzufügen oder bestehende zu verbessern – auch während späteren Phasen des Entwicklungszyklus. Auch lassen sich verschiedene Versionen eines Produktes schneller freigeben. Zu beachten ist, dass einige Embedded-Betriebssysteme zwar ein eingeschränktes Prozessmodell unterstützen, aber nicht die obige Methode verfolgen. Statt jedem Prozess eine virtuelle Adresse mit dem Startpunkt 0 zuzuordnen, verlässt sich das Betriebssystem auf Fixups und Offsets um Prozesse und Treiber im Speicher zu positionieren. Das führt dazu, dass ein Binärcode nicht immer für die komplette Produktpalette verwendet werden kann. Findet dennoch ein und derselbe Binärcode Verwendung, muss sichergestellt sein, dass er in das vorhandene Schema der Speicherzuordnung passt.

Bei konventionellen Betriebssystemarchitekturen muss jeder Entwickler möglicherweise die Funktionsweise jedes Moduls detailliert kennen lernen, einfach um zu verhindern, dass man sich nicht im Adressraum eines anderen aufhält. Bei zunehmender Komplexität einer Anwendung ist es manchmal erforderlich, mehr Zeit zum Erlernen des Quellcodebaums zu verwenden, als ihn zu entwickeln. Beim UPM müssen die Programmierer das System nicht in- und auswendig kennen. Tritt eine Speicherkorruption auf, wird sie vom Betriebssystem identifiziert. Erfahrene Entwickler haben mehr Zeit, das zu tun, wofür sie bezahlt werden: Kernprobleme lösen und den Produkten echten Mehrwert hinzufügen. Auch Programmierer mit weniger Erfahrung werden schon viel früher produktive Lösungen schaffen. Da beim UPM die Fehlersuche in Maschinencode-orientierten Modulen mit Hilfe von Quellcode-orientierten Werkzeugen durchgeführt werden kann, können Programmierer, die nur Anwenderapplikationen geschrieben haben, nun auch Treiber, Dateisysteme usw. entwickeln.

Kürzere Entwicklungszyklen bei großen Projekten

Da sich Entwickler nicht mehr ausführlich mit dem gesamten System auskennen müssen, kann beim UPM ein großes Projekt in kleinere, unabhängige Teams aufgeteilt werden. Dadurch wird die Entwicklung mindestens auf zweierlei Weise beschleunigt: Erstens sind kleinere Teams naturgemäß effizienter. Zweitens kann jedes Team einfach ein Softwaresubsystem entwickeln und testen, auch wenn die anderen dazugehörigen Subsysteme noch nicht fertiggestellt sind. Nehmen wir beispielsweise an, dass ein Team an Modul A arbeitet und ein anderes Team an Modul B, und dass diese beiden Module im endgültigen Produkt in enger Wechselbeziehung stehen werden.

Da jedes Modul oder jede Modulgruppe als unabhängiger Prozess laufen kann, kann das erste Team mit der Programmierung, Prüfung und Fehlerbehebung für Modul A beginnen, ohne dass es darauf warten muss, dass das zweite Team mit Modul B beginnt. Das einzige was beide Teams vorab machen müssen, ist, darüber zu entscheiden, welche Form die gemeinsam genutzten Daten haben werden. Diese gemeinsame Datennutzung könnte unterschiedlich implementiert werden: Message-Passing, Shared Memory, POSIX Message-Queues, etc.

Das erste Team kann dann eine einfache „Testumge-

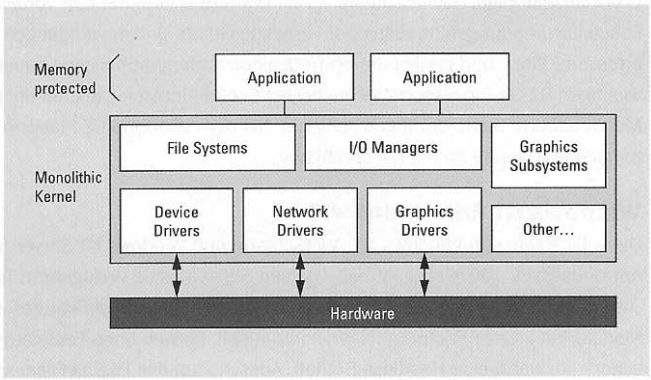


Bild 2: Eine monolithische Architektur bietet Speicherschutz, aber nur für Anwendungen

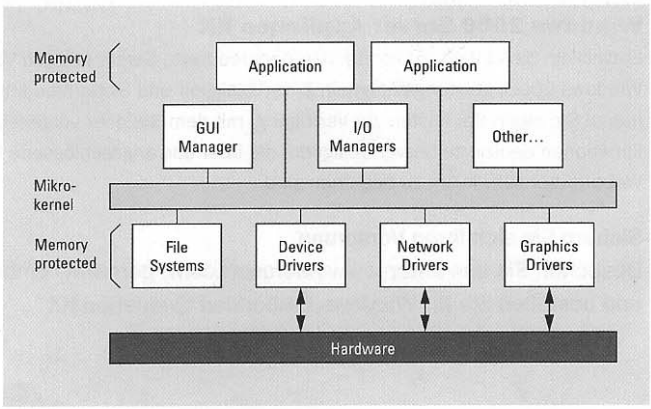


Bild 3: Die UPM-Architektur schützt den Speicher für sämtliche Softwarekomponenten einschließlich Betriebssystemmodule und Treiber

bung" erstellen, um zu prüfen, ob das (noch nicht implementierte) Modul B funktionsgerecht mit Modul A interagieren wird. Wie bereits erwähnt, wird durch das UPM ermöglicht, dass mehrere Entwickler dasselbe Zielsystem gleichzeitig nutzen können. Das bedeutet, dass die Prüf- und Fehlerbeseitigungsmaßnahmen des einen Teams die Arbeit der anderen Teams nicht stören oder verzögern wird. Die Teams arbeiten somit parallel und nicht nacheinander.

Das UPM beseitigt praktisch Kernel-Faults, verfügt aber über eine Anzahl weiterer eigener Features, durch die die Zuverlässigkeit und Verfügbarkeit von Embedded-Systemen erhöht wird. Dazu gehören eine automatische Wiederherstellung nach Softwarefehlern, ein schnelles Austauschen sowohl der Hardware als auch der Software sowie die Möglichkeit, Komponenten der Anwendung über multiple CPUs zu verteilen.

Vorteile bei der Ausführungszeit erkennbar

Ganz egal wie sorgfältig programmiert wurde, einige Fehler werden erst zur Ausführungszeit auftreten. Bei einer „flachen" Architektur ist ein Reset die einzige Möglichkeit zur Erholung des Systems. Bei einer monolithischen Architektur ist eine Erholung ohne Neustart möglich, aber nur wenn der Fehler auf der Anwendungsebene liegt. Beim UPM ist eine Erholung ohne Neustart möglich, und zwar auch dann wenn der Fehler in einem Treiber, einem Protokollstack oder in einem kundenspezifischen Betriebssystemmodul auftritt. Um das zu erreichen, nutzen Sie intelligente, anwendergeschriebene Mechanismen, die als „Software Watchdogs" bezeichnet werden.

Nehmen wir beispielsweise an, dass ein Treiber fehlerhaft ist. Statt einen vollständigen Reset zu erzwingen könnte ein Software Watchdog folgendes tun:

- den Treiber einfach neu starten

ODER

- den Treiber sowie eventuell dazugehörige Prozesse neu starten.

In beiden Fällen kann der Softwaredesigner genau bestimmen, welche Prozesse neu gestartet werden sollen.

Beim Durchführen eines teilweisen Neustarts oder, falls erforderlich, eines koordinierten System-Resets, kann der Software Watchdog auch Informationen über den Softwarefehler sammeln. Hat das System beispielsweise Zugriff auf einen Massenspeicher, wie Flashspeicher oder Festplatte kann der Watchdog eine Speicherauszugsdatei des Prozesses erzeugen, die Sie mit Quellcode-orientierten Testhilfen betrachten können. Diese Speicherauszugsdatei

- kennzeichnet genau die Codezeile, die den Fehler verursacht hat und
- ermöglicht das Ansehen der Ressourcen wie z.B. Variablen und eine History von Funktionsaufrufen.

Vergleicht man diese Vorgehensweise mit den konventionellen Hardware Watchdogs, die das System einfach zurücksetzen, ohne auch nur eine Spur des Fehlers zurückzulassen, ist die Wahl eindeutig. Der Fehler kann nicht nur nachvollzogen, sondern auch behoben werden.

Darüber hinaus kann ein Software Watchdog Systemereignisse überwachen, die für einen Hardware Watchdog „unsichtbar" sind. Beispielsweise kann ein Hardware Watchdog dafür sorgen, dass ein Treiber die Hardware bedient. Aber er kann nur sehr schwer feststellen, ob andere Programme korrekt mit dem Treiber kommunizieren. Ein Software Watchdog dagegen kann auch diese Lücke abdecken und Maßnahmen ergreifen, bevor der Treiber selbst Anzeichen für Probleme aufweist.

Bild 4: Bei konventionellen Betriebssystemarchitekturen ist die Treiberentwicklung unpraktisch und zeitaufwändig

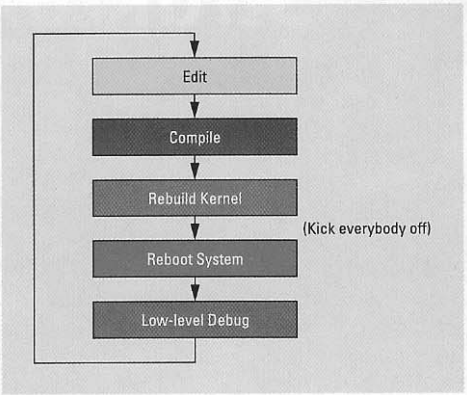
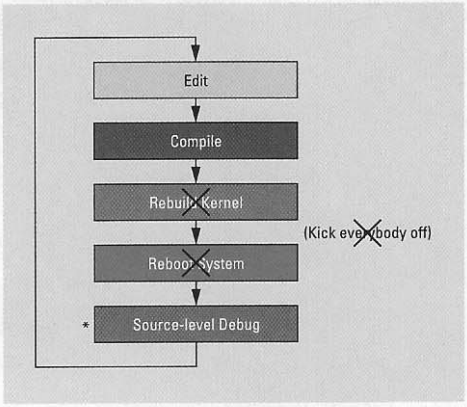


Bild 5: Durch die UPM-Architektur wird die Treiberentwicklung drastisch rationalisiert



Hohe Verfügbarkeit durch Software Hot-Swap

Natürlich sind Softwarefehler nicht die einzige Ursache dafür, dass ein System abstürzt. Anwender müssen häufig ihre „flachen" und monolithischen Systeme herunterfahren um Treiber, Protokollstacks oder Betriebssystemmodule auf den neuesten Stand zu bringen. Mit der UPM-Architektur können Sie jedoch diese Module im Betrieb austauschen, ohne das System neu starten zu müssen. Zu beachten ist dabei, dass damit nicht das einfache Hinzufügen von Modulen bei einem aktiven System gemeint ist. Bei einigen monolithischen Betriebssystemen lassen sich beispielsweise Treiber am Kernel dynamisch anbinden. Da diese Treiber dann jedoch im Kernelraum laufen, können sie nicht entfernt, neu gestartet oder ersetzt werden. Dagegen kann beim UPM jede Komponente beliebig hinzugefügt oder entfernt werden.

Mit dieser flexiblen Kontrolle über „low-level"-orientierte Module lässt sich auch veraltete oder defekte Hardware austauschen, und zwar wiederum ohne das System neu starten zu müssen. Versagt beispielsweise eine Ethernetkarte, kann der Treiber beendet werden, eine neue Karte eingesetzt und der Treiber mit den Parametern der neuen Karte gestartet werden. Es lässt sich sogar eine neue Karte einstecken, die einen anderen Chipsatz verwendet, und dann wird der entsprechende Treiber gestartet. Durch eine ähnliche Vorgehensweise kann man sogar so katastrophalen Situationen wie einem Festplattenabsturz begegnen. Versagt eine lokale Festplatte, lassen sich Dateioperationen auf eine an-

dere Festplatte im Netz umdirigieren. Das ist etwas völlig anderes als bei konventionellen Betriebssystemen, wo sogar eine triviale Wartungsaufgabe wie das Upgraden eines Maustreibers ein System-Reset oder ein Neukompilieren des Kernels notwendig machen kann.

Skalierbarkeit durch verteilte Verarbeitung

Die Zuverlässigkeit eines Systems lässt sich erheblich verbessern, indem man die Komponenten der Anwendung über mehrere CPUs verteilt. Auf diese Weise kann nicht einmal der Ausfall einer CPU die Anwendung daran hindern, ihren Dienst zu tun. Tatsächlich hat man durch das Anwachsen einer Anwendung häufig gar keine andere Wahl, als eine Aufteilung auf mehrere CPUs. Nicht wegen der Zuverlässigkeit, sondern da die Anwendung mehr physikalische Schnittstellen benötigt, oder auch ganz einfach mehr Rechenleistung als eine einzige CPU liefern kann. Leider ist diese Aufgabe bei konventionellen Betriebssystemen sehr umständlich, da die meisten Softwaremodule an den Kernel gebunden sind. Wird beispielsweise ein Protokollstack von einer CPU zu einer anderen bewegt, müssen eventuell zwei neue Kernelabbildungen erzeugt und überprüft werden – eine für jede CPU. Verfügt das Betriebssystem über keine transparenten Möglichkeiten, um mit einem Modul zu kommunizieren, das auf eine andere CPU bewegt wurde, muss sowohl dieses Modul selbst als auch die Module, mit denen es kommuniziert, neu kompiliert werden. Erschwerend kommt hinzu, dass es oft schwierig ist, festzulegen, welche Prozesse welcher CPU zugeordnet werden sollten. Manchmal weiß der Entwickler bis zur Integrationsphase nicht, dass die Prozesse so verteilt wurden, dass gar keine optimale Leistung erreicht wird. Und dann ist es oft zu spät, die Software neu zu codieren, neu zu kompilieren und nochmals zu prüfen.

Das UPM umgeht diese Probleme indem es alles vom Kernel abkoppelt. Jedes Softwaremodul bildet ein unabhängiges, bewegliches Objekt.Und wenn das UPM so implementiert ist, dass die Interprozesskommunikation (IPC) transparent über das Netz läuft, dann kann ein Prozess weiterhin mit einem anderen kommunizieren, auch wenn dieser andere Prozess zu einer anderen CPU bewegt wurde. Es sind keine Codeänderungen oder Neuverknüpfungen erforderlich. Tatsächlich kann der exakte Binärcode jedes Prozesses jederzeit verschoben werden, auch während der Ausführungszeit.

Das bedeutet natürlich, dass Programmierer nicht eine bestimmte Systemkonfiguration im Sinn haben müssen, um ein Programm zu entwickeln. Egal wie groß das endgültige Zielsystem sein wird, Entwickler brauchen Ihre Programme nur einmal zu schreiben. Es ist beispielsweise unwichtig, ob eine Festplatte und der dazugehörige Treiber letztendlich auf dem lokalen Rechner oder auf einem fernen, im Netzwerk verbundenen Rechner platziert sein wird. So oder so wird jeder Prozess, sofern er über die richtigen Zugriffsrechte verfügt, auf die Festplatte transparent und ohne besonderen Code zugreifen können. *(hh)*

Kennziffer: 309

www.elektronikpraxis.de

- Alle Infos zur Embedded-Echtzeit-Plattform von QNX
- Die Entwicklungswerkzeuge von QNX im Detail
- Was ist eigentlich eine Echtzeit-Plattform?
- QNX Developer's Net – Das Entwickler-Forum im Web

Vorausgesetzt, die Architektur stimmt ...

Zusammenfassend lässt sich sagen, dass die Wahl des richtigen Betriebssystems entscheidend ist. Nicht nur in Bezug auf die Zuverlässigkeit und die Leistung des Systems, sondern auf die eigentlichen Möglichkeiten, neue Produkte unter engen Zeitvorgaben bereitzustellen. Natürlich können auch Werkzeuge einen Unterschied machen, doch können sie den erheblichen, durch konventionelle Betriebssysteme auferlegten Aufwand für Fehlersuche, Prüfung und Wartung nicht ausgleichen. Bleiben Innovation, Time-to-Market und Zuverlässigkeit Wunschdenken? Kann ein Betriebssystem dabei helfen? Die Antwort ist definitiv: Ja, vorausgesetzt natürlich, es hat die richtige Architektur. *(hh)*