Wann genau benötigt man ein Echtzeitbetriebssystem?

Paul Leroux, Technology Analyst
Malte Mundt, Field Application Engineer

QNX Software Systems
paull@qnx.com, mmundt@qnx.com

Inhalt

Durch die Leistungsfähigkeit moderner Prozessoren und die Verfügbarkeit von Echtzeit-Patches für Standardbetriebssysteme stellt sich die Frage, wann man für Embedded-Systeme tatsächlich noch ein Echtzeitbetriebssystem braucht. Wir schauen uns deshalb an, welche Verhaltensweisen nur ein wirkliches Echtzeitbetriebssystem (RTOS, „Real Time Operating System“) garantieren kann, egal ob auf High-End-CPUs oder auf kleineren Prozessoren. Damit wird klar, warum Echtzeitbetriebssysteme aus der Embedded-Welt nach wie vor nicht wegzudenken sind.

Einleitung

Benötigt man für übliche Embedded-Projekte wirklich noch ein Echtzeitbetriebssystem? Angesichts der Leistung moderner Prozessoren und der Verfügbarkeit von Echtzeit-Versionen bzw. -Erweiterungen für Linux, Windows und anderen universell einsetzbaren Standard-Betriebssystemen muss man sich diese Frage natürlich stellen.

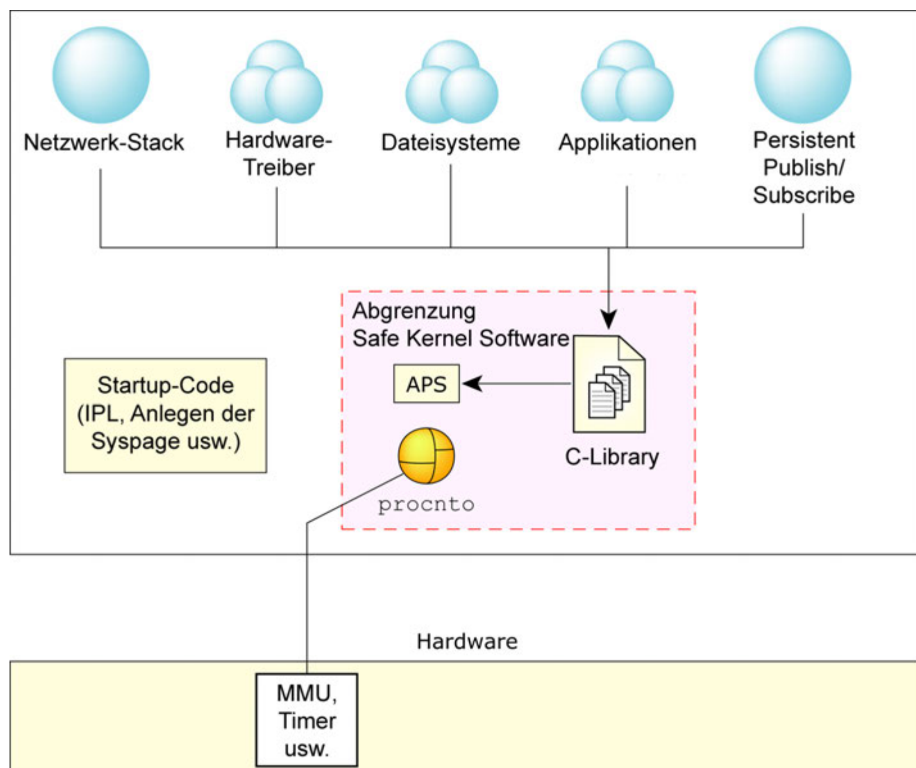

Abbildung 1. Auch im sicherheitskritischen Umfeld haben Echtzeitbetriebssysteme bestimmte Vorteile. Zu sehen ist hier die Abgrenzung der sicherheitszertifizierten Softwarekomponenten im QNX Neutrino RTOS Safe Kernel.

Die Antwort liegt unter anderem in der Natur der Geräte: Werden diese zu Tausenden oder gar in Millionen-Stückzahlen produziert, kann bei der Hardware selbst eine Kostenersparnis von nur einem Euro über kommerziellen Erfolg oder Misserfolg entscheiden. Superschnelle Multi-Gigahertz Prozessoren sind dabei häufig schlicht zu teuer (von der unerwünschten Hitzeentwicklung mal ganz abgesehen).

Im Bereich der Fahrzeugtelematik beispielsweise läuft ein typischer 32-Bit Prozessor mit 600 MHz viel langsamer als in modernen Desktop- oder Serversystemen. Die kurzen, exakt definierbaren Reaktionszeiten eines Echtzeitbetriebssystems, auch auf weniger leistungsfähiger Hardware, sind hier ein entscheidender Faktor. Neben der Kostenersparnis vereinfacht ein RTOS auch viele der technischen Herausforderungen, speziell wenn mehrere Abläufe im System gleichzeitig um Ressourcen konkurrieren.

Stellen Sie sich ein System vor, das in jedem Fall unmittelbar auf Benutzereingaben reagieren muss. Mit einem RTOS kann ein Entwickler garantieren, dass die Reaktionen auf Eingaben vor allen anderen Aktivitäten durchgeführt werden. Es sei denn, eine wichtigere Aktion muss (z.B. aus Sicherheitsgründen) noch früher ausgeführt werden.

Oder stellen Sie sich ein System vor, das Quality of Service (QoS) Anforderungen erfüllen muss, zum Beispiel die Wiedergabe eines Live-Videostreams. Ist Software involviert, kann es unter Umständen zu Darstellungsfehlern kommen (verzögerte oder übersprungene Bilder) - aus Sicht des Anwenders ist das System unzuverlässig. Mit einem RTOS hingegen kann man die Reihenfolge der Prozessausführung exakt steuern und so eine stets ausreichende und konsistente Wiederholrate für die Darstellung garantieren.

Echtzeitsysteme sind nicht „fair"

Die Forderung nach „harter" Echtzeit hat in der Embedded-Industrie nicht an Bedeutung verloren. Aber was genau unterscheidet ein „echtes" RTOS von einem universellen Standard-OS? Und wie sind die heute verfügbaren Echtzeit-Erweiterungen für Standard-Betriebssysteme zu bewerten? Stellen sie eine brauchbare Alternative zur Leistung eines RTOS dar?

Fangen wir beim Scheduling an: In einem Standard-OS herrscht üblicherweise Fairness bei der Zuteilung der CPU an Threads und Prozesse. Dies ermöglicht eine maximale Gesamtleistung des Systems für Desktop- und Serveranwendungen. Aber: Man hat keine wirkliche Garantie, dass wichtige, zeitkritische Threads mit Vorzug vor unwichtigeren Aufgaben ausgeführt werden.

Ein Standard-OS kann beispielsweise die Priorität eines wichtigen Threads im Sinne der Fairness gegenüber anderen Threads im System herabsetzen oder dynamisch anpassen. Dieser kann somit dann von eigentlich niedriger priorisierten Threads unterbrochen werden. Zudem gibt es bei Standard-Betriebssystemen meist keine Obergrenze für Dispatch-Latenzzeiten. Je mehr Threads laufen, desto länger dauert es, bis ein Thread wieder zur Ausführung kommt. Jeder dieser Gründe kann dazu führen, dass ein wichtiger Thread seine Deadline nicht einhalten kann – selbst auf einer schnellen CPU.

In einem RTOS werden die Threads streng nach Priorität ausgeführt. Wenn ein High-Priority-Thread ausführbereit wird, wird er innerhalb eines kleinen, begrenzten Zeitintervalls die CPU von einem niedriger priorisierten Thread übernehmen (niedrige Context Switch Zeiten). Zudem darf der wichtigere Thread dann ohne Unterbrechung laufen, bis er seine Aufgabe erfüllt hat - es sei denn, er wird seinerseits von einem noch höher eingestuften Thread unterbrochen. Dieser als "Prioritäten-gesteuertes präemptives Scheduling" bekannte Ansatz stellt sicher, dass wichtige Threads ihre Deadlines zu 100% einhalten, selbst wenn viele Threads um CPU-Zeit konkurrieren.

Unterbrechbarer Kernel?

Die meisten Standard-Betriebssysteme haben keinen unterbrechbaren OS-Kernel. Das bedeutet: Selbst ein wichtiger Thread mit hoher Priorität kann einen gerade laufenden Systemaufruf (Kernel Call) nicht unterbrechen, sondern muss auf dessen Beendigung warten - selbst wenn dieser Aufruf vom unwichtigsten Thread im System kam. Außerdem: Wenn ein Treiber oder anderer System-Dienst im Auftrag eines Client-Threads zu arbeiten beginnt (oft eben auch als Kernel Call implementiert), geht jede Priorisierung verloren. Denn der Kernel selbst ist in der Regel ja kein Prozess und hat damit keine Priorität. Dieses Verhalten führt zu unkalkulierbaren Verzögerungen und kritische Aktivitäten werden ggf. nicht rechtzeitig abgeschlossen.

In einem RTOS hingegen sind auch Kerneloperationen unterbrechbar. Wie bei jedem Standard-OS gibt es Zeitfenster, in denen es keine Unterbrechung geben darf. Aber in einem RTOS sind diese Zeitfenster extrem kurz, in der Größenordnung von einigen hundert Nanosekunden. Dadurch ergeben sich Obergrenzen für nicht unterbrechbare Bereiche, in denen Interrupts gesperrt sind. Das ermöglicht den Entwicklern, eine maximal mögliche Verzögerung zu bestimmen.

Um konsistente Berechenbarkeit und zeitlich korrekte Ausführung kritischer Aktivitäten garantieren zu können, muss der RTOS-Kernel so einfach und elegant wie möglich sein. Diese Einfachheit erreicht man am besten, wenn der Kernel selbst nur Services enthält, die schnell ausgeführt werden können. Indem man aufwändige Operationen (z.B. Laden von Prozessen) an externe Prozesse bzw. Threads auslagert, kann der RTOS-Designer sicherstellen, dass die zeitliche Obergrenze für den längsten nicht unterbrechbaren Code-Pfad im Kernel möglichst gering ist.

Bei einigen Standard-Betriebssystemen wurde der Kernel mittlerweile um ein gewisses Maß an Unterbrechbarkeit erweitert. Aber die nicht unterbrechbaren Zeiträume sind immer noch viel länger als bei einem typischen RTOS. Deren Länge hängt nämlich vom längsten kritischen Ausführungspfad in jedem einzelnen Modul (z.B. Netzwerkstack) ab, das Teil des Standard-OS-Kernels ist. Zudem werden auch bei einem unterbrechbaren Standard-Kernel bestimmte andere Situationen nicht abgedeckt, die zu völlig unvorhersehbaren Latenzzeiten führen können, wie z.B. der oben erwähnte Verlust der Prioritätsinformation beim Aufruf von Treiber- oder Systemdienst-Funktionen seitens der Applikationsthreads.

Vermeidung von Prioritätsinversion

Bei vielen Betriebssystemen kann ein Thread mit niedriger Priorität einen wichtigeren Thread unbeabsichtigt von der Zuteilung von CPU-Zeit abschneiden – ein Effekt, der als Prioritätsinversion bekannt ist. Tritt eine zeitlich unbegrenzte Prioritätsinversion ein, können kritische Deadlines nicht eingehalten werden. Das kann zu unerwünschtem Verhalten oder sogar zu komplettem Systemversagen führen. Leider wird die Gefahr der Prioritätsinversion beim Systementwurf häufig übersehen. Es gibt zahllose Beispiele für Prioritätsinversion, eines davon ließ 1997 sogar die "Pathfinder"-Mission der NASA scheitern[1].

Allgemein bedeutet Prioritätsinversion, dass zwei Tasks unterschiedlicher Priorität auf eine gemeinsame Ressource zugreifen und der höher priorisierte Task die Ressource nicht vom anderen Task übernehmen kann. Ein RTOS kann, im Gegensatz zu einem Standard-OS, mehrere Maßnahmen ergreifen, um die dadurch entstehende Verzögerung zu begrenzen. Eine ist die Prioritäts-

---

[1] Barr, Michael. “Introduction to Priority Inversion," *Embedded Systems Programming,* Volume 15: Number 4, April 2002.

vererbung, eine andere die „Priority Ceiling Emulation“. Es soll hier nicht um einen Vergleich der beiden Mechanismen gehen, deshalb wollen wir uns als Beispiel auf die Prioritätsvererbung konzentrieren.

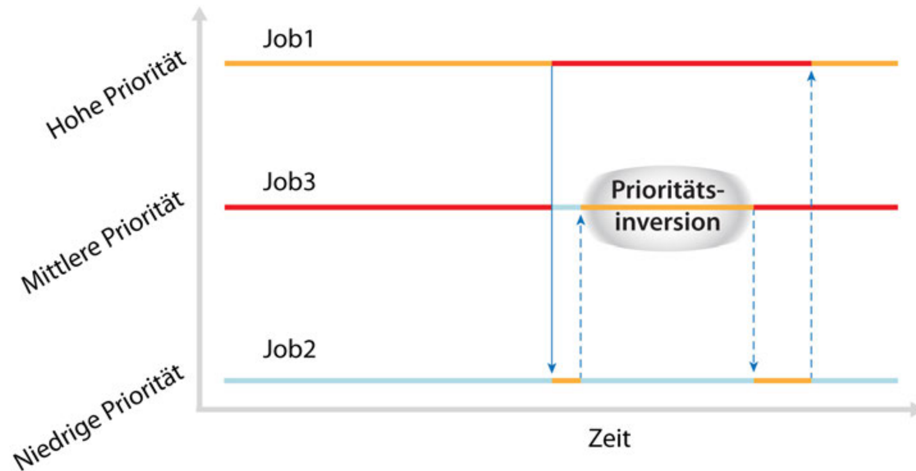

Abbildung 2. Job1 wartet auf Job2, dann wird Job2 von Job3 unterbrochen. Job1 wird nun noch länger in seiner Ausführung verzögert.

Vorab müssen wir uns klar machen, wie die Synchronisation von Tasks zu Blockaden führen kann und letztlich eine Prioritätsinversion auftritt. Angenommen es laufen zwei Jobs, Job1 und Job2, Job1 sei höher priorisiert. Wenn Job1 bereit ist, aber auf Job2 warten muss, dann haben wir eine Blockade. Diese kann durch Synchronisation ausgelöst werden, z.B. wenn der Zugriff auf eine gemeinsame Ressource durch ein Lock oder ein Semaphor geregelt wird und Job1 auf die Freigabe der Ressource durch Job2 wartet. Oder wenn Job1 einen Service anfordert, der gerade von Job2 genutzt wird.

Durch die Blockade kann Job2 so lange laufen, bis die Bedingung eintritt, auf die Job1 wartet (z.B. dass Job2 die gemeinsame Ressource freigibt). Nun kann Job1 ausgeführt werden. Die Wartezeit von Job1 wird als Blockadefaktor bezeichnet. Wenn Job1 zwingend seine Deadlines einhalten muss, darf der Blockadefaktor nicht von Parametern wie der Anzahl der Threads oder Eingaben an das System abhängen. Kurz, der Blockadefaktor muss begrenzt sein.

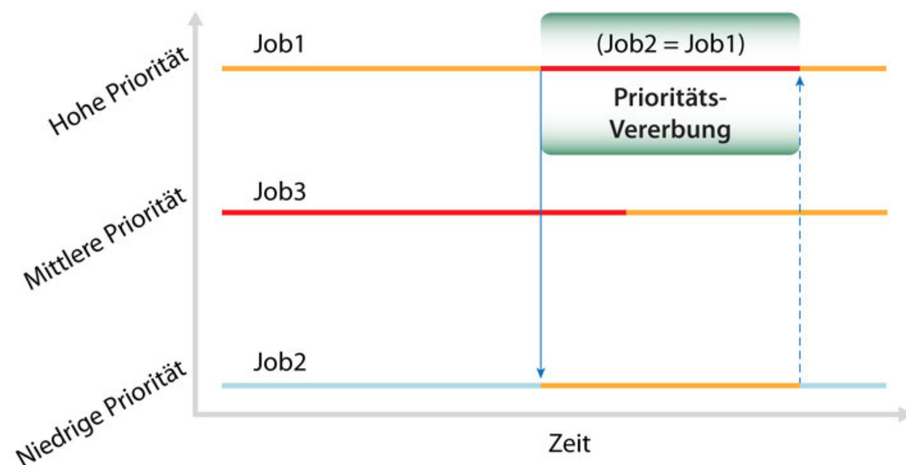

Abbildung 3. Job2 erbt die höhere Priorität von Job1 und verhindert, dass Job3 Job2 unterbricht. Job3 verzögert nicht mehr die Ausführung von Job1.

Nun wollen wir einen dritten Job – Job3 – einführen, der höher priorisiert ist als Job2, aber niedriger als Job1 (siehe Abbildung 2). Wenn Job3 bereit ist, während Job2 läuft, wird er Job2 unterbrechen und dieser kann erst wieder

laufen, wenn Job3 anhält oder terminiert. Der neue Job wird somit den Job1 weiter in der Ausführung verzögern. Die gesamte Verzögerung durch die Unterbrechung ist die Prioritätsinversion.

Im schlimmsten Fall könnte Job2 mehrfach unterbrochen werden, das führt dann zu einer Blockadekette. Job2 könnte für eine unvorhersehbare Zeit unterbrochen werden, somit eine unbeschränkte Prioritätsinversion nach sich ziehen und Job1 könnte keine seiner Deadlines einhalten.

Das ist der Punkt, an dem zur Lösung des Problems die Prioritätsvererbung auf den Plan tritt. Betrachten wir das obige Beispiel noch einmal, versehen aber nun während der Synchronisationsphase Job2 mit der Priorität von Job1 (Vererbung), dann kann Job3 Job2 nicht unterbrechen und die Prioritätsinversion tritt nicht ein (siehe Abbildung 3).

Zeitpartitionierung

In vielen Systemen ist die garantierte Verfügbarkeit von Ressourcen unabdingbar. Wenn einem kritischen Subsystem beispielsweise die CPU entzogen wird, ist dessen Funktionalität für den Anwender plötzlich nicht mehr verfügbar. Bei einer Denial-Of-Service (DoS) Attacke könnte ein Angreifer ein System so lange mit Anfragen überschütten, die mit hoher Priorität behandelt werden müssen, bis das System unter der CPU-Last zusammenbricht und für den Anwender unbrauchbar wird.

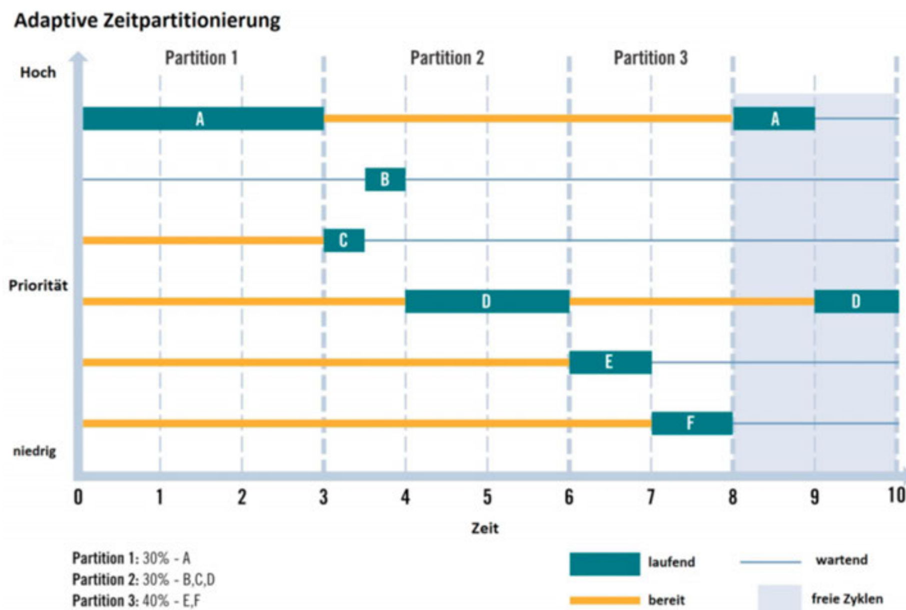

Abbildung 4. Adaptive Zeitpartitionierung verhindert, dass Tasks mehr CPU-Zeit verbrauchen, als ihrer Partition zugeteilt wurde - es sei denn, es gibt nicht genutzte CPU-Zeit. Beispielsweise können die Tasks A und D Zeit von Partition 3 verwenden, da E und F die zugeteilte Zeit nicht vollständig nutzen.

Aber nicht nur Sicherheitslücken können zur Überlastung führen. Häufig liegt es schlicht an zusätzlich implementierter Funktionalität, dass ein System überlastet wird und bestehende Software-Komponenten unzureichend CPU-Zeit bekommen. Bislang problemlos laufende Anwendungen und Services reagieren nach der Erweiterung dann evtl. nicht mehr oder zumindest nicht wie erwartet. Die klassische Lösung wäre nun die Hardware aufzurüsten oder aber ein Neudesign der Software vorzunehmen – beides keine wünschenswerten Alternativen.

Um dieses Problem in den Griff zu bekommen, brauchen Systemdesigner ein Partitionierungsschema, das mittels Hardware oder Software CPU-Zeitbudgets

überwacht. Damit können einzelne Prozesse oder Threads nicht mehr alle CPU-Zyklen für sich monopolisieren. Da ein RTOS Prozessor, Speicher und andere Ressourcen zentral verwaltet, ist es für die Einführung von CPU-Budgets ein guter Kandidat.

Manche Echtzeitbetriebssysteme bieten eine statische Zeitpartitionierung. Bei so einem Scheduler kann man die Tasks in Gruppen oder Partitionen einteilen und jeder Partition einen festen Prozentsatz der CPU-Zeit zuteilen. Bei diesem Ansatz kann kein Task in irgendeiner Partition mehr CPU-Zeit konsumieren, als der Partition fest zugeteilt wurde. Beispielsweise erhalte eine Partition 30% der CPU-Zeit. Wenn irgendein Prozess in dieser Partition Opfer eines Denial-Of-Service Angriffs wird, so wird er dennoch nie mehr als 30% der CPU-Zeit beanspruchen. Somit können die anderen Prozesse weiterhin ihre Aufgaben erfüllen. Zum Beispiel würde das sicherstellen, dass das Userinterface (z.B. ein Remote-Terminal) weiterhin verfügbar ist und der Operator das Problem beheben kann, ohne das System komplett neu starten zu müssen.

Leider hat dieser Ansatz ein Problem: Da dieser Algorithmus fest zuteilt, kann eine Partition nie CPU-Zeit einer anderen Partition verwenden, auch wenn diese die ihr zugeteilten CPU-Zyklen gar nicht benötigt. Somit wird Leistung verschwendet und das System kann Peaks in der CPU-Nutzung nur eingebremst abarbeiten. Als Systemdesigner muss man also teurere Hardware einkalkulieren oder mit einem langsameren System leben - oder man beschränkt die Anzahl der Features.

Adaptive Zeitpartitionierung

Ein weiteres Zuteilungsverfahren, „Adaptive Zeitpartitionierung“ genannt, vermeidet die Nachteile der festen Zuteilung durch einen dynamischen Algorithmus. Wie beim statischen Partitionieren kann man hier CPU-Zeit für einen Prozess oder eine Gruppe von Prozessen reservieren. Somit kann man sicher sein, dass die Last auf einem Subsystem nicht die Verfügbarkeit des Gesamtsystems beeinträchtigt. Im Gegensatz zu statischen Ansätzen können hier aber dynamisch CPU-Zyklen von Partitionen mit wenig Last abgezogen werden um sie solchen zuzuweisen, die mehr Rechenzeit benötigen. Zuteilungs-budgets werden nur dann erzwungen, wenn die CPU komplett ausgelastet ist. Somit kann man zum einen Höchstlasten bewältigen und 100% Ausnutzung erreichen, zum anderen hat man die Sicherheit garantiert zugeteilter CPU-Ressourcen.

Was besonders wichtig ist: Die adaptive Zeitpartitionierung kann ohne Codeänderungen auf ein bestehendes System aufgesetzt werden. Beim QNX® Neutrino® RTOS beispielsweise kann ein Systemdesigner einfach bestehende POSIX-Anwendungen starten und der Scheduler stellt sicher, dass jede Partition ihr Budget zugeteilt bekommt. Innerhalb jeder Partition werden die Tasks nach wie vor nach den Regeln des prioritätsbasierten präemptiven Multitaskings verwaltet, Anwendungen müssen also in Bezug auf Scheduling nicht verändert werden. Außerdem kann man die Partitionen auch dynamisch feintunen um so das System auf optimale Leistungsfähigkeit zu bringen.

Ein zweiter Kernel

Standard-Betriebssysteme wie Windows, Linux und verschiedene UNIX-Systeme verfügen üblicherweise über keinen der bislang diskutierten Mechanismen[2]. Um diese Lücke zu schließen, haben verschiedene Anbieter einige Echtzeiter-weiterungen und Patches entwickelt. Zum Beispiel ein Design mit zwei Kerneln,

---

[2] Mehr Informationen zu den diversen Aspekten von verschiedenen Echtzeitbetriebssystemen finden Sie z.B. in den unabhängigen Evaluierungsreports des Dedicated Systems Instituts: <http://download.dedicated-systems.com/>

bei dem der Standard-Kernel als Task auf dem RTOS-Kernel läuft (siehe Abbildung 5). Alle Tasks, die ein deterministisches Scheduling brauchen, laufen ebenfalls auf diesem Kernel, aber mit höherer Priorität als der Kernel des Standard-OS. Somit können diese Tasks den Standard-OS-Kernel jederzeit unterbrechen und sicherstellen, dass die CPU erst dann wieder freigegeben wird, wenn sie ihre Arbeit erledigt haben.

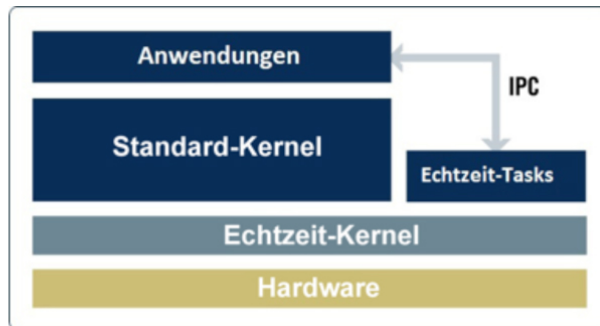

Abbildung 5. In einer typischen Dual-Kernel-Implementierung läuft das Standard-OS als Task mit niedrigster Priorität auf einem Echtzeit-Kernel.

Leider können die im Echtzeit-Kernel laufenden Tasks Services des Standard-OS, wie z.B. Filesystem, Netzwerk, usw. nur eingeschränkt nutzen: Wenn ein im Echtzeit-Kernel laufender Task irgendeinen Service des Standard-OS aufruft, treten wieder genau die gleichen Probleme auf, die dafür sorgen, dass ein Standard-OS sich nicht deterministisch verhält. Somit müsste man speziell für den RTOS-Kernel neue Treiber und Services implementieren, auch wenn es diese Komponenten für das Standard-OS bereits gibt. Im Echtzeit-Kernel laufende Tasks profitieren auch nicht von der Robustheit einer MMU-Speicherverwaltung des Standard-OS. Somit kann ein Programmfehler in einem Echtzeit-Task, beispielweise ein falsch gesetzter C-Pointer, zu einem fatalen Absturz führen. Das ist definitiv ein großes Problem, da Systeme, die Echtzeit benötigen, normalerweise auch ein sehr hohes Maß an Zuverlässigkeit benötigen.

Um es noch ein wenig komplizierter zu machen, verwenden unterschiedliche Implementierungen dieser "Zwei-Kernel-Lösung" unterschiedliche APIs. In den meisten Fällen kann ein für das Standard-OS geschriebener Service nicht einfach auf den Echtzeit-Kernel portiert werden. Tasks, die für eine Echtzeiterweiterung von Anbieter A geschrieben werden, laufen noch lange nicht auf der Implementierung des Anbieters B.

Solche Ansätze zeigen, wie schwierig und aufwändig es ist, ein Standard-OS um Echtzeitfähigkeiten zu erweitern. Das hat gar nichts mit „gutes RTOS“ gegen „schlechtes Standard-OS“ zu tun – Standard-Betriebssysteme wie Linux, Windows oder die zahlreichen UNIX-Derivate funktionieren auf Desktop- oder Serversystemen hervorragend. Aber sie kommen eben an ihre Grenzen, wenn sie in deterministischen Umgebungen eigesetzt werden, für die sie nie konzipiert wurden: Fahrzeugtelematik, Medizingeräte, Industriesteuerungen, usw.

Das OS erweitern und anpassen

Bei allen Nachteilen bezgl. deterministischer Umgebungen haben Standard-Betriebssysteme natürlich auch ihre Vorteile: Support für gängige und überall verwendete APIs beispielsweise. Für Linux spricht zudem das Open-Source-Modell. Hat man den Quellcode verfügbar, kann ein Entwickler Komponenten des Betriebssystems an die eigenen Anforderungen anpassen und Zeit bei der Fehlersuche sparen. Hersteller von Echtzeitbetriebssystemen sind sich dieser Aspekte natürlich inzwischen bewusst und manche setzen deshalb z.B. auf die POSIX-API, die jedem vertraut ist, der schon mal mit Linux oder einer UNIX-Variante zu tun hatte. Ebenso wird von einigen Anbietern Source Code zu OS-Komponenten zur Verfügung gestellt.

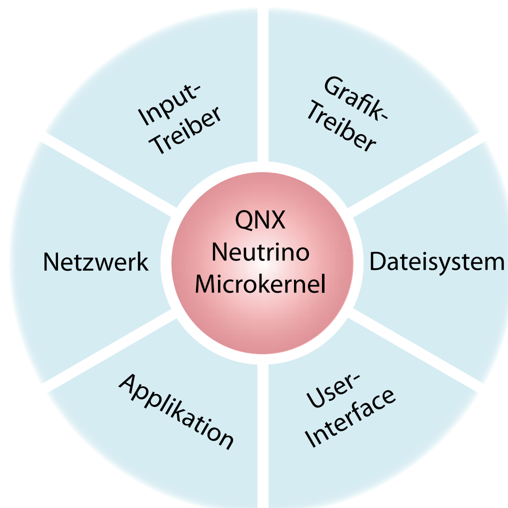

Abbildung 6. Bei einem Microkernel-RTOS laufen Services des Systems als normale Prozesse im Modus für Anwendungen. Anpassungen werden somit viel einfacher.

Auch die Architektur eines RTOS spielt eine wichtige Rolle: Basiert das RTOS auf einem Microkernel-Design, werden Anpassung und Erweiterung von Betriebssystemfunktionalität wesentlich einfacher als bei anderen Architekturen. Bei einem Microkernel liegt nur ein kleiner Teil der allerwichtigsten Services im Kernel (z.B. Signale, Timer, Scheduler). Alle anderen Komponenten, wie Treiber, Dateisystem, Netzwerk-protokolle, Anwendungen laufen außerhalb des Kernels als separate, speichergeschützte Prozesse (siehe Abbildung 6). Somit benötigt man für die Entwicklung eigener Treiber oder sonstiger Betriebssystem-Erweiterungen weder Kernel-Debugger noch spezifisches Wissen über den Kernel. Tatsächlich lassen sich solche Erweiterungen genau so einfach wie normale Anwendungen entwickeln, da sie mit normalen Source-Level Debuggern und Werkzeugen entwickelt werden können.

Ein Beispiel: Wenn ein Treiber auf Speicher außerhalb des dem Prozess zugewiesenen Bereichs zugreifen möchte, kann das Betriebssystem den Prozess identifizieren, die Stelle des Fehlers lokalisieren und einen Dump des Prozesses erzeugen, der mit einem Source-Level Debugger untersucht werden kann. Die Dump-Datei enthält alle Informationen, die der Debugger zur Identifizierung der fehlerhaften Codezeile benötigt und zusätzliche Diagnoseinformation: den Wert von Variablen oder die Historie der Funktionsaufrufe beispielsweise.

Eine solche Architektur bietet zudem hervorragende Fehlerisolierung und Recovery-Fähigkeiten: Wenn ein Treiber, ein Netzwerk-Protokoll oder irgendein anderer Service versagt, schlägt das nicht auf andere Komponenten oder gar den Kernel durch. Man kann solche Ereignisse kontinuierlich mit „Software Watchdogs“ überwachen und betroffene Services dynamisch neu starten, ohne dass das Gesamtsystem neu gebootet werden muss oder der Anwender in irgendeiner Weise eingreifen müsste. Äquivalent kann man Treiber oder andere Services dynamisch stoppen, aktualisieren und neu starten, ohne dass man einen Shutdown des Gesamtsystems durchführen müsste.

Man sollte solche Vorteile nicht unterschätzen - die größtmögliche Störung eines Echtzeitbetriebes ist ein ungeplanter Neustart! Auch ein geplanter Neustart im Fall von Softwareupdates legt den Betrieb still, aber wenigstens kontrolliert. Um Deadlines auch im Dauerbetrieb einhalten zu können, muss man ein Betriebssystem verwenden, das ohne Unterbrechungen verfügbar ist, sogar bei Softwarefehlern oder Service-Upgrades.

Eine strategische Entscheidung

Ein RTOS verhilft komplexen Applikationen zu einem berechenbaren und zuverlässigen Verhalten. Die präzise Kontrolle des Timings durch ein RTOS ist mit einem Standard-OS unerreichbar (wenn sich ein auf einem Standard-OS aufbauendes System aufgrund von Timing-Problemen nicht korrekt verhält,

können wir es mit Recht als unzuverlässig bezeichnen). Die Auswahl des richtigen RTOS ist allerdings nicht einfach. Die Architektur des Systems ist ein wichtiges Kriterium, aber daneben gibt es noch weitere:

- *Flexible Wahlmöglichkeit bei den Scheduling-Algorithmen* – Unterstützt das RTOS verschiedene Algorithmen (FIFO, Round-Robin, Sporadic, usw.)? Kann der Entwickler den Algorithmus für einzelne Threads auswählen, oder zwingt ihn das RTOS zu einem einzelnen Algorithmus für alle Threads im System?

- *Zeitpartitionierung* – Unterstützt das RTOS Zeitpartitionierung, die Prozessen einen definierten Prozentanteil der CPU-Zeit garantiert? Solche Garantien vereinfachen die Integration von Subsystemen anderer Entwicklungsteams oder Hersteller erheblich. Sie stellen zudem sicher, dass kritische Tasks im Fall von Denial-of-Service Attacken (DoS) oder anderen Angriffen verfügbar bleiben und ihre Deadlines einhalten.

- *Unterstützung für Multi-Core-Prozessoren* – Die Möglichkeit, Multi-Core Systeme nutzen zu können, ist für High-Performance Entwicklungen enorm wichtig geworden. Unterstützt das RTOS verschiedene Multiprocessing-Strategien (symmetrisches / asymmetrisches / gebundenes Multiprocessing)? Gibt es vernünftige Tracing-Tools, mit denen man die Leistung des Systems untersuchen und optimieren kann? Ohne Werkzeuge, die einen über Ressourcenkonflikte, häufige Thread-Migration und andere typische Multi-Core-Probleme informieren können, wird die Optimierung eines solchen Systems schnell zu einer schwierigen und zeitintensiven Aufgabe.

- *Werkzeuge zur Remote-Diagnose* – Downtime ist bei vielen Embedded-Systemen im Feld nicht tolerierbar. Deshalb sollte der Hersteller Werkzeuge anbieten, die das Verhalten des Systems im laufenden Betrieb untersuchen können, so dass die Funktionalität des Systems weiter verfügbar bleibt. Suchen Sie sich einen Hersteller, der Laufzeit-Analysewerkzeuge für System- und Anwendungs-Profiling und zur Heap-Analyse anbietet.

- *Offene Entwicklungsumgebung* – Bietet der Hersteller eine Entwicklungsumgebung an, die auf einer offenen Plattform wie Eclipse aufsetzt, in die Entwickler Plug-Ins für ihre favorisierten Werkzeuge für Modellierung oder Versionskontrolle integrieren können? Oder basiert die gesamte Umgebung auf proprietärer Technologie?

- *Graphische Benutzeroberflächen* – Bietet das RTOS nur einfache Grafikbibliotheken oder verfügt es über hochentwickelte Grafiktechnologien, wie z.B. Multi-Layer-Unterstützung, Multi-Head-Displays, schnelles 3D-Rendering und die Möglichkeit, Standards wie Qt oder HTML5 einzusetzen? Kann man das Erscheinungsbild der Oberfläche einfach anpassen? Kann die Oberfläche gleichzeitig verschiedene Sprachen anzeigen und verarbeiten (Chinesisch, Koreanisch, Japanisch, English, Russisch, usw.)? Können 2D- (z.B. HTML5) und 3D- (z.B. OpenGL ES) Anwendungen simultan auf dem Bildschirm dargestellt werden?

- *Standard APIs* – Zwingt das RTOS die Entwickler, eine proprietäre API zu verwenden, oder bietet es zertifizierte Unterstützung für Standard-APIs wie POSIX oder OpenGL ES? Das macht es sehr viel einfacher, Programme von und auf andere Umgebungen zu portieren. Bietet das RTOS auch wirklich eine vollständige Unterstützung für eine API, oder kann man womöglich nur einen Teil der Aufrufe verwenden?

- *Middleware für digitale Medien* – Eine flexible Unterstützung für digitale Medien wird für viele Embedded-Systeme zunehmend wichtiger, z.B. für Autoradios, Medizingeräte, Industriesteuerungen, Mediaserver und natürlich

Unterhaltungselektronik. Systeme müssen mit unterschiedlichsten Quellen (USB-Flashdrives, MP3-Player, Internetstreams, Bluetooth-Handys, usw.) zurechtkommen, verschiedene Formate verstehen und eine Reihe von DRM-Technologien unterstützen. Ein RTOS-Hersteller kann den softwareseitigen Aufwand für die Anbindung der verschiedenen Datenquellen und die Verarbeitung signifikant verringern, indem er eine geeignete Middleware für digitale Medien bereitstellt. Mit einer sauber entworfenen Middleware ist es zudem viel einfacher, neue Datenquellen, z.B. ein zukünftiges iPhone, zu integrieren, ohne dass man das Userinterface oder andere Komponenten dafür ändern müsste.

Die Wahl eines RTOS ist eine strategische Entscheidung für jedes Projektteam. Wenn ein Hersteller auf jede der oben aufgeworfenen Fragen klare Antworten liefern kann, sind Sie bei der Auswahl eines für Sie passenden und zukunftssicheren Betriebssystems einen entscheidenden Schritt weiter gekommen.

Über QNX Software Systems

QNX Software Systems ist Hersteller innovativer Embedded-Technologien, dazu gehören Middleware, Entwicklungswerkzeuge und Betriebssysteme. Die komponentenbasierte Architektur des QNX® Neutrino® RTOS, die QNX Momentics® Tool Suite und die QNX Aviage® Middleware-Reihe bilden gemeinsam das zuverlässigste und skalierbarste Fundament für leistungsfähige Embedded-Systeme. Weltweit bekannte Firmen wie z.B. Cisco, Daimler, General Electric, Lockheed Martin oder Siemens nutzen QNX-Technologie für Telematik- und Infotainmentsysteme, Industrieroboter, Netzwerk-Router, medizinische Geräte, Sicherheits- und Verteidigungssysteme und viele andere betriebs- und sicherheitskritische Applikationen. Die QNX-Firmenzentrale ist in Ottawa, Kanada, die deutsche Niederlassung befindet sich in Hannover.

www.qnx.com

© 2012-2013 QNX Software Systems Limited. QNX, QNX CAR, Momentics, Neutrino, Aviage are trademarks of QNX Software Systems Limited, which are registered trademarks and/or used in certain jurisdictions. All other trademarks belong to their respective owners. 302234 MC411.110