Vorwort

Die vorliegende Arbeit analysiert und visualisiert Schritt für Schritt auf elementarer Ebene die Konzepte und Probleme der Parallelverarbeitung für herkömmliche und echtzeitfähige Multitasking/Multithreading-Systeme, und liefert Lösungen in Form von C-Funktionen, unterstützt von einigen wenigen Assembler-Routinen. Am Ende steht ein kompletter Betriebssystem-Kern mit einer Reihe zugänglicher Funktionen für die angewandte parallele Programmierung zur Verfügung (Task-Kommunikation, Betriebsmittelnutzung, Echtzeitbetrieb etc.) und eine Reihe nicht-zugänglicher Funktionen, die den Betriebssystem-Kern realisieren und den sicheren, lebendigen und gerechten Ablauf paralleler Anwender-Programme ermöglichen.
Multitasking ist die Fähigkeit eines Betriebssystems, mehrere Programme (Tasks) quasi parallel auf einem Prozessor ablaufen zu lassen. Für den Betrachter sieht es so aus, als würde jedes Programm ständig laufen. So wird ihm die Illusion der Parallelität vermittelt.
Mit Windows NT/CE/XP..., VxWorks, QnX, Euros, Linux etc. sind eine ganze Reihe verschiedener Multitasking/Multithreading-Betriebssysteme für kommerzielle oder Echtzeit-Anwendungen verfügbar. Sie haben eines gemeinsam: Sie realisieren Parallelität nach den gleichen elementaren Prinzipien und Methoden z.B. zur Synchronisation von Tasks/Threads und deren Kommunikation.
Unterstützt wird die Multitasking-Fähigkeit dieser Betriebssysteme von ausgereiften Prozessor-Architekturen, wie z.B. den Protected Mode-Mechanismen der INTEL-Prozessoren (IA32-Familie). Weil Parallelität sehr schnelle Wechsel von Task zu Task erforderlich macht, übernimmt die dafür notwendigen Operationen die Hardware und befreit damit den Betriebssystem-Kern von dieser aufwendigen Arbeit.
Zusätzlich schützt die Hardware die Adressräume der Anwender-Tasks und des Betriebssystem-Kerns und schließt auf diese Weise gegenseitige Störungen aus. Die Hauptlast für den korrekten Betrieb eines Multitasking/Multithreading-Systems tragen jedoch seine Kern-Funktionen.
Sie sorgen dafür, daß parallele Programme sicher und lebendig ablaufen. So dürfen z.B. Botschaften, die zwischen den Tasks ausgetauscht werden, nicht verloren gehen (Sicherheit). Auch sollten Tasks am Gesamtgeschehen gleichmäßig beteiligt werden (Lebendigkeit). Und keine der Tasks sollte begünstigt oder gar ausgesperrt werden (Gerechtigkeit). Dies gilt in gleicher Weise auch für Parallel-Programme, die über Rechnergrenzen hinweg in sogenannten verteilten Systemen bzw. Multicomputersystemen ablaufen.

Das Buch besteht aus 7 Kapiteln.

Das l. Kapitel behandelt das Task-Management. Es stellt die Verbindung zur Protected Mode-Architektur des Prozessors her und zeigt die Zustände einer Task. Es realisiert ein Preemptiv-Multitasking-System mit Warteschlangen, Scheduling-Algorithmen (Prioritätsstufen, Round-Robin, mehrstufiges Herabsetzen der Priorität), Dispatcher, einfache Eingabe/Ausgabe.

Im 2. Kapitel werden alle Aspekte der Task-Kommunikation und Synchronisation behandelt. Es liefert Lösungen zum Problem des gegenseitigen Ausschlusses bei zwei und mehreren Tasks über Semaphore und Condition-Queues. Realisiert suspend- und resume-Operationen mit wait- und signal-Funktionen. Greift das Erzeuger/Verbraucher-Problem auf und nutzt das Konzept des Monitors, um Task-Kommunikationen über sende- und empfange-Dienste (einschließlich broadcasting) mit mailboxen zu realisieren.

Das 3. Kapitel ist der parallelen Numerik gewidmet. Es zeigt das Numerik-Management und demonstriert am Beispiel der numerischen Integration eine angewandte Parallelprogrammierung bei Nutzung der dafür geeigneten Zerlegungsmethode.

Das 4. Kapitel behandelt Realzeit-Multitasking. Es beinhaltet das Interrupt-Task-Management, verdeutlicht den Sinn der Idle-Task, implementiert eine Zeitverwaltung mit timer-Monitore und realisiert das Pausieren und das zeitbegrenzte Warten auf Nachricht.

Das 5. Kapitel enthält eine Behandlung des IO-Managements in Systemen mit mehreren Betriebsmitteln wie z.B. Drucker, Scanner, CD ROM etc. In einem solchen System können schwierige Probleme auftauchen. Wenn z. B. zwei Tasks gleichzeitig den gleichen Drucker benutzen, führt dies zu keinem vernünftigen Ergebnis. Oder wenn die eine Task den Drucker besitzt und die andere Task den Scanner und beide das jeweils andere Betriebsmittel anfordern, dann sind beide Tasks blockiert und bleiben in diesem Zustand für immer. Ein solcher Zustand wird Deadlock oder Verklemmung genannt. Wie Verklemmungszustände entstehen und wie sie entdeckt und verhindert werden werden können, wird detailliert in diesem Kapitel analysiert. Es schließt ab, mit der Entwicklung und Implementierung von zugänglichen Kernel-Funktionen, deren verantwortungsbewußte Nutzung ein System mit beliebig vielen Tasks und Betriebsmitteln verklemmungsfrei hält. Alle Überlegungen in diesem Kapitel basieren hauptsächlich auf der Forschungsarbeit und dem Modell von Holt. Einer der ersten und einflußreichsten Beitragenden auf dem Gebiet der Verklemmungen war Dijkstra.

Das 6. Kapitel beschäftigt sich mit dem reader-writer-Problem, das als erstes von Courtois, Heyman und Parnas (1971) formuliert und mit Semaphoren gelöst wurde. Das Problem entsteht, wenn zwei Arten von Tasks, als reader und writer bezeichnet, ein gemeinsames Betriebsmittel, wie z.B. eine Datei in einem Datenbanksystem, benutzen müssen. Den writern ist es gestattet, das Betriebsmittel zu verändern. Folglich müssen sie exklusiven Zugriff auf das Betriebsmittel haben. Andererseits darf eine unbestimmte Vielzahl von readern das gemeinsame Betriebsmittel parallel lesen. Dies hat zur Folge, dass in dieser Situation eine gerechte Strategie dafür sorgen muss, dass weder die reader noch die writer noch beide für immer ausgeschlossen werden. In diesem Kapitel wird eine Monitorlösung vorgestellt, die von Hoare und Gorman (1974) stammt und schließt mit der Entwicklung von zugänglichen Kernel-Funktionen ab.

Das 7. Kapitel enthält eine elementare Einführung in verteilte Systeme.
Die Existenz leistungsstarker Mikroprozessoren auf der einen Seite und lokaler Netzwerke auf der anderen Seite, legen es nahe, beide Technologien zu vereinigen und ein Rechner-System zu konstruieren, das aus einer Vielzahl von Prozessoren besteht, die über ein Kommunikationsnetzwerk miteinander verbunden sind. Ziel eines solchen Systems ist es, dem Benutzer die Illusion zu vermitteln, dass er es mit einem einzigen Rechner zu tun hat und nicht mit einer Ansammlung von mehreren unterschiedlichen Rechnern. Die Task-Kommunikation spielt dabei eine wesentliche Rolle. Ein solches System verhält sich wie ein virtuelles Einprozessorsystem mit dem Makel, dass es dort keinen gemeinsamen Speicher und keine gemeinsamen Betriebsmittel gibt. So müssen ausgesuchte Rechner ihre lokale Einheiten wie Speicher und IO als virtuelle globale Datenstrukturen für den gemeinsamen Zugriff zur Verfügung stellen. Dies macht sie zu kritischen Regionen, die nur unter gegenseitigem Ausschluss betreten werden dürfen.

Vorwort

> Die Simulation der dafür notwendigen P/V-Operationen in einem zentralen Algorithmus und die Strategie zur Vermeidung von deadlocks bei vielfachem Betriebsmittelbedarf sind die Kernaspekte dieses Kapitels. Damit sich die Kommunikation nicht ausschließlich auf explizite Datenübertragungen beschränkt, greift es die Idee von Nelson und Birrell auf und schließt mit dem Mechanismus des entfernten Prozeduraufrufs (RPC) einschließlich den Input-/Output-RPCs ab.

So führt die vorliegende Arbeit die in der Literatur meist getrennt beschriebenen Welten, wie Betriebssysteme für Einprozessor- und Multicomputersysteme auf der einen Seite und Prozessor-Architektur auf der anderen Seite, **ganzheitlich** zusammen. Sie veranschaulicht ihr Zusammenwirken an über 400 Abbildungen und realisiert die typischen Funktionen eines Betriebssystem-Kerns mit dem Memory Management Programm (dem sogenannten Build-Programm) und mit vornehmlich C-Programmen und einigen wenigen Assembler-Routinen. Der Kern ist auf jeder Intel-IA32-Architektur im Protected Mode ablauffähig.

Alle Kernel-Funktionen wurden mit passender Entwicklungssoftware auf 32-bit Embedded-Systemen entwickelt und mit einem geeigneten Remote-Debugger getestet.

Klaus-Dieter Thies, Ph.D.