Universität Erlangen-Nürnberg

Lehrstuhl für Betriebssysteme
IMMD IV

Hauptseminar WS 1997/98
Aktuelle Trends in der Betriebssystem-Forschung

SPIN

Erweiterbarer Kern durch Event-Mechanismus

Vortrag am 14. Januar 1998

Alexander Lechner

Inhaltsverzeichnis:

1. Einleitung

SPIN ist ein Betriebssystem, das den Anforderungen kommender ressourcenintensiver Anwendungen gerecht werden will. Erreicht werden soll dies durch einen erweiterbaren Microkernel, den die jeweilige Anwendung sicher **und** effizient auf die eigenen Bedürfnisse anpassen kann.

1.1 Motivation

In den letzten Jahren hat sich in der Computerwelt ein radikaler Wandel vollzogen, der sich auch in den nächsten Jahren noch weiter fortsetzen wird. Anwendungen, die früher eher als Nischendienstleistungen angesehen wurden, wie z.B. große verteilte Datenbanken oder High-Quality-Multimedia-Anwendungen, bestimmen mehr und mehr den heutigen Rechneralltag. Den ständig wachsenden Ansprüchen der Anwendungssoftware stehen jedoch häufig immer noch Betriebssysteme entgegen, deren grundlegende Konzepte aus den 70er und 80er Jahren stammen.

Die meisten Betriebssysteme befinden sich gezwungenermaßen auf einer Gratwanderung zwischen Generalisierung und Spezialisierung. Je spezialisierter ein Betriebssystem aber ist, desto wahrscheinlicher ist es auch, daß einige Anwendungen darauf nur sehr langsam oder gar nicht mehr funktionieren. Bei generalisierten Betriebssystemen lassen sich Anpassungen auf bestimmte Anwendungsgebiete zwar zumeist mit mehr oder weniger großem Aufwand vornehmen, dabei tritt aber wieder dasselbe Problem auf. Spezialisierungen sollte deswegen jede Anwendung *individuell* vornehmen können, um bestimmte Funktionalität oder Performance zu erreichen, wie z.B. Interprozeß- und Netzwerkkommunikation, Thread-, Virtual Memory- und Cache-Management.

Natürlich muß bei diesen anwendungsspezifischen Anpassungen auch die Systemintegrität durch wohldefinierte Sicherheitsmechanismen gewährleistet bleiben, damit kein Konfigurationschaos à la MS-DOS, Windows oder MacOS entsteht. Ein solches zu verhindern, hat sich SPIN zur Aufgabe gesetzt.

1.2 Ziele und Ansätze von SPIN

SPIN wurde als universelles Betriebssystem konzipiert mit dem Ziel, Erweiterbarkeit, Sicherheit und hohe Performance in einem System zu vereinen. Die Interfaces zu Diensten und Ressourcen, die das System den Anwendungen zur Verfügung stellt, bestimmen dabei die *Erweiterbarkeit*. *Sicherheit* zwischen den einzelnen Anwendungen und dem System muß durch eine feinstrukturierte Zugriffskontrolle gewährleistet werden, *hohe Performance* kann nur erreicht werden, wenn die Kommunikation zwischen den Erweiterungen und dem Systemkern mit geringem Overhead abläuft.

In SPIN werden diese Ziele durch vier Mechanismen erreicht, die entweder bereits auf der Sprachebene oder aber zur Laufzeit greifen:

- *Co-location*: Betriebssystemerweiterungen werden dynamisch in den virtuellen Adreßraum des Kerns hinzugelinkt, wodurch der Kommunikationsaufwand zwischen Erweiterungs- und Systemcode äußerst gering gehalten werden kann.
- *Erzwungene Modularität*: Die Erweiterungen müssen in Modula-3 geschrieben werden, einer modularen Programmiersprache. Der Compiler fordert zwischen den einzelnen Modulen klar definierte Schnittstellen, so daß auch die Erweiterungen, die ja im virtuellen Adreßraum des Kerns ablaufen, nur über vom Kern exportierte Interfaces, auf dessen Interna zugreifen können. Module werden also schon auf Sprachebene untereinander und vom Kern getrennt.

- *Logical Protection Domains*: Erweiterungen existieren innerhalb sogenannter *logical protection domains*, Namensräume innerhalb des Kerns, die ablaufbaren Code und exportierte Interfaces umschließen. Diese Interfaces sind Elemente der Sprachebene und repräsentieren den Blick auf Systemressourcen, die durch das Betriebssystem geschützt sind. Ein dynamischer Linker innerhalb des Kerns verbindet zur Laufzeit Code aus unterschiedlichen logical protection domains, so daß die Kommunikation zwischen Erweiterungen aus verschiedenen Domains als simpler Prozeduraufruf abläuft.
- *Dynamic call binding*: Erweiterungen laufen grundsätzlich als Antwort auf ein System-Event ab. Dabei kann ein solches Event jedmögliche Zustandsänderung im System beschreiben, wie z.B. einen Page Fault, die Deblockierung eines Threads, oder den Aufruf eines System-Calls. Zur Verteilung dieser Events, die in den Interfaces der Erweiterungen deklariert werden, reicht wiederum ein simpler Prozeduraufruf aus.

2. Überblick über das SPIN-Betriebssystem

In diesem Abschnitt wird zunächst ein Überblick über das SPIN Operating System gegeben, in dem die SPIN-Philosophie und der Event-Mechanismus kurz umrissen werden.

Spindles

In SPIN können Applikationen den Mikrokern, der in diesem System sich auch wirklich so bezeichnen läßt, sicher und dynamisch spezialisieren, um die durch die Applikation geforderte Funktionalität und Performance zu erreichen.
Diese spezialisierten Programmstücke, in der SPIN-Terminologie *spindles* (*SPIN Dynamically Loaded Extensions*) genannt, werden in der typ-sicheren Programmiersprache Modula-3, in derselben Sprache wie der Mikrokern, geschrieben. Diese Spindles, im weiteren oft auch nur mit *Erweiterungen* bezeichnet, werden dynamisch in den virtuellen Adreßraum des Kerns gelinkt, wodurch sie schnellen Zugriff auf andere Systemdienste haben.

Der Microkernel

Der Kern selbst stellt nur Implementierungen bzw. Interfaces von Thread-, Geräte- und virtuellem Speichermanagement zur Verfügung, letzteres wird auch zur Erzeugung von Adreßräumen benutzt. Anwendungen laufen jeweils in ihrem eigenen virtuellen Adreßraum ab und können somit in jeder beliebigen Sprache verfaßt werden, denn unerlaubter Speicherzugriff führt zu einer System-Exception, Speicherschutz und Systemintegrität bleiben dadurch gewahrt.

Im Gegensatz zu traditionellen Mikrokernen sind im SPIN-Mikrokern die Abstraktionen der physikalen und logischen Ressourcen des Systems strikt von den Managementpolicies, die durch Spindles implementiert werden, getrennt.

Neben diesen konventionellen Kerndiensten bietet SPIN auch noch Erweiterungsdienste an, mit denen anwendungsspezifische Dienste in das laufende System integriert werden können. Diese Erweiterungsdienste übernehmen zwei Aufgaben: Zum einen muß neuer Code sicher im virtuellen Adreßraum des Kerns installiert werden, zum anderen muß dieser neue Code auch sicher mit bestehenden Diensten des Systems oder anderer Spindles verbunden werden.

Dynamisches Linken

Das *Installieren* in den Adreßraum des Kerns übernimmt der dynamische Linker von SPIN, der nur Objektcode akzeptiert, der vorher auch vom Modula-3 Compiler gezeichnet worden ist. Der Linker ersetzt alle noch ungelösten Referenzen der Erweiterung durch Symbole aus der *logical protection domain*, gegen die die Erweiterung gelinkt wird. Diese *logical protection domain* ist eine Menge von Interfaces, die Zugriff auf Variablen und Prozeduren erlauben, z.B.

zur Speicherverwaltung. Sollte ein Symbol nicht aufgelöst werden können (weil es nicht in der *logical protection domain* enthalten ist), so schlägt der gesamte Link-Prozeß fehl.

Events und Event-Dispatcher

Das *Verbinden* des neuen Codes mit bereits bestehenden Diensten des Systems geschieht durch SPINs *dynamischen Event-Dispatcher*, der aufgetretene Events an entsprechende Event-Handler verteilt. Wie schon erwähnt wird jede Zustandsänderung im System als ein Event betrachtet, das entweder durch den Kern selbst oder durch eine Erweiterung ausgelöst werden kann. Jede Anwendung bzw. ein entsprechendes Spindle, das über das Auftreten eines bestimmten Events informiert werden will, kann sich als *Event-Handler* registrieren lassen und wird dann durch den Dispatcher aufgerufen. Dabei sind aber noch diverse Sicherheitsmechanismen involviert, auf die später im Detail eingegangen wird.

Die folgende Abbildung soll den Aufbau von SPIN verdeutlichen: Man erkennt den Minimalkern mit seinen wenigen Diensten, ebenso die Erweiterungen (Spindles), die im virtuellen Adreßraum des Kerns installiert sind und dort entweder bestimmte anwendungsspezifische Dienste verrichten oder aber von mehreren Anwendungen zusammen benutzt werden können.

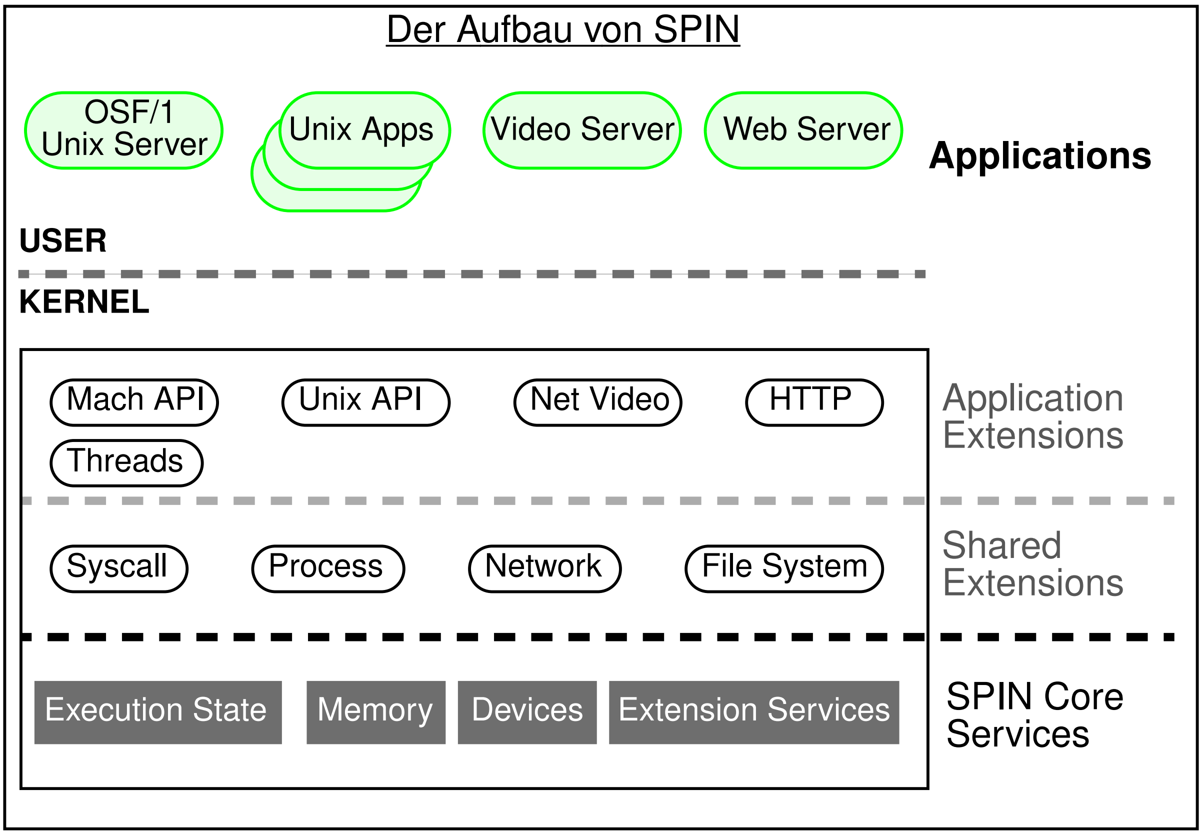

Abbildung 1. Der Aufbau von SPIN

3. Die Architektur von SPIN

Die SPIN Architektur bietet eine gute Infrastruktur, um System- und Anwendungscode sicher zu kombinieren. Dabei werden an die darunterliegende Hardware jedoch nur wenige Anforderungen gestellt, vielmehr verläßt man sich auf Mechanismen der Sprachebene, auf denen das folgende Sicherheitsmodell und Erweiterungsmodell durch Events beruhen.

Grundlegende Eigenschaften von Modula-3

Das SPIN-System sowie seine Erweiterungen sind in Modula-3 geschrieben, dessen wichtigste Merkmale die Unterstützung von Interfaces, Typsicherheit, automatischer Speicherverwaltung, Objekten, generischen Schnittstellen, Threads und Exceptions sind. Dabei werden die vier letztgenannten nur aus ästhetischen Gründen genutzt, um den Entwurf eines großen Systems zu vereinfachen, die ersten drei Merkmale aber spielen die entscheidende Rolle:

- *Interface*: Ein Interface deklariert den nach außen sichtbaren Teil einer Modul-Implementierung. Alle anderen Teile des Modules sind von außen nicht zugreifbar. Diese Einschränkung überprüft der Compiler zur Übersetzungszeit.
- *Typsicherheit*: Dadurch wird verhindert, daß auf irgendwelche Speicherbereiche willkürlich (z.B. durch “Pointer-Umbiegung”) zugegriffen wird. Neben dieser statischen Typüberprüfung zur Kompilierungszeit, geschieht die Feldgrenzenüberprüfung bei Arrays zusätzlich auch noch zur Laufzeit.
- *Automatische Speicherverwaltung*: Hierbei wird verhindert, daß Speicherbereich, auf den es noch einen Zeiger gibt, wieder auf den Heap gelangt und dort für ein neues Objekt eines anderen Typs wiederverwendet wird.

3.1 Das Sicherheitsmodell

Das Sicherheitsmodell soll den Zugriff auf die Ressourcen des Systems kontrollieren. Oft werden dazu auch virtuelle Adreßräume eingesetzt, die aber nur sehr aufwendig zu erzeugen und darüberhinaus noch langsam im Zugriff sind. SPIN geht hierbei einen anderen Weg:

Capabilities

Alle Ressourcen des Kerns werden ausschließlich über sogenannte Capabilities angesprochen, eine unfälschbare Referenz auf eine Ressource, die ein Systemobjekt (z.B. physikalische Seite), ein Interface (z.B. zum Allokieren einer physikalischen Seite) oder eine Kollektion von Interfaces (z.B. das gesamte virtuelle Speichersystem) sein kann. Dadurch ist gewährleistet, daß Erweiterungen nur auf solche Ressourcen zugreifen können, zu denen sie eine Capability besitzen.

Da auch alle Erweiterungen in Modula-3 geschrieben werden müssen, also derselben Sprache wie der Kern, kann man Capabilities in SPIN jetzt einfach mittels ganz normaler Zeiger implementieren und ist nicht auf Hardwareunterstützung angewiesen. Diese Zeiger sind Referenzen auf Speicherbereiche, deren Typ innerhalb eines Interfaces deklariert wurde. Der Compiler verhindert dabei, daß ein Zeiger “umgebogen” oder inkonsistent gemacht wird. Zur Laufzeit jedoch könnte z.B. ein C-Programm eine Capability fälschen, würde es einfach einen normalen C-Zeiger als Capability erhalten. Deswegen werden Capabilities vom Kern an UserLevel-Anwendungen nur als *externe Referenzen* übergeben, das sind *Indizes* in eine Tabelle mit Referenzen, die für jede Applikation im Kern getrennt zur Verfügung steht.

Protection Domains

Neben den Capabilities bilden die sogenannten *Protection Domains* eine weitere Grundlage des Sicherheitsmodells. Eine *Protection Domain* definiert eine Menge von Namen oder Programmsymbolen, die von Programmcode, der Zugang zu dieser Domain hat, benutzt werden kann. Damit Erweiterungen sicher innerhalb des Kerns ablaufen können, exisitieren sie immer innerhalb solcher Protection Domains, haben also nur Zugriff auf bestimme Symbole bzw. Code, die sich in der jeweiligen Domain befinden, gleichzeitig sind sie von unbefugtem Zugriff durch andere Module geschützt. Protection Domains spielen für das dynamische Linken von Erweiterungscode in den Adreßraum des Kerns eine wichtige Rolle:

Der dynamische Linker muß zuerst feststellen, ob die Objektdatei der Erweiterung als **sicher** angesehen werden kann. Dies ist dann der Fall, wenn die Objektdatei zwar dem Kern unbekannt ist, aber vom Modula-3 Compiler angenommen wurde, oder aber der Kern die Sicherheit andersweitig gewährleisten kann (so gibt es zum Beispiel bestimmte Gerätetreiber, die zwar in C geschrieben wurden, aber direkt vom Hersteller stammen und deswegen als sicher gelten). Ist die Sicherheit garantiert, so kreeiert der dynamische Linker mittels **Domain.Create()** aus der Objektdatei des Erweiterungsmoduls eine neue Domain. Alle Namen und Symbole, die in den Interfaces der Objektdatei *exportiert* werden, werden auch von der Domain *exportiert*. Symbole, die von Interfaces der Objektdatei *importiert* werden, werden zunächst nicht aufgelöst und repäsentieren Interfaces, die zwar vom Programmcode innerhalb der Objektdatei importiert werden, deren Implementierung aber noch nicht gefunden wurde.

Diese nichtaufgelösten Symbole werden schließlich durch die *Resolve*-Operation aufgelöst, die als Basis für das dynamische Linken fungiert. Hierbei werden eine Quell- und Ziel-Domain angegeben, woraufhin dann alle nichtaufgelösten Symbole der Ziel-Domain mittels der exportierten Symbole aus der Quell-Domain aufgelöst werden.
Hierzu nun ein kleines Beispiel: Sei **A** eine Domain, die Symbol **x** exportiert und Symbol **y** importiert. Wird **A** jetzt gegen Domain **B**, die Symbol **y** exportiert gelinkt, so enthält die Exportmenge von **A** anschließend immer nur noch **x** (und nicht auch noch **y**), aber die Referenz von **A** auf das Symbol **y** verweist nunmehr auf das von **B** exportierte Symbol **y**.

Daneben gibt es auch die Möglichkeit, bestehende Domains zu einer neuen Domain zusammenzufassen. Der *Combine*-Befehl erschafft eine neue Domain, die die Vereinigungsmenge der bereits existierenden Domains enthält. Kombinierte man in obigem Beispiel **A** und **B**, so entstünde eine neue Domain, die **x und y** exportiert.

Von dieser Möglichkeit macht der Kern großen Gebrauch, um bestimmte Interfaces in einer einzigen Protection Domain zusammenzufassen. So wird zur Bootzeit des Systems eine Domain **SpinPublic** erzeugt, in der alle Interfaces enthalten sind, die den Applikations-Erweiterungen zur Verfügung stehen. Die Interfaces darin entsprechen denen eines traditionellen MicroKernel-System-Call-Interfaces, deren Benutzung die Systemintegrität nicht gefährdet. Interfaces, bei denen das möglich ist, sind in der **SpinPrivate-Domain** enthalten, die aus diesem Grund nur einigen wenigen, priviligierten Clients zugänglich ist.

3.2 Events in SPIN

Das Herzstück zur Integration von Erweiterungen in das System, ist der Event-Mechanismus von SPIN. Jedwedes Systemvorkommnis wird explizit durch das Auslösen eines *Events* eines bestimmten Namens angekündigt. Jeder (System oder Anwendung), der sich für ein solches Vorkommnis interessiert, kann sich als *Event-Handler* registrieren lassen.

Dieser Mechanismus mag vielleicht für die *Erweiterung* eines Systems ausreichen, die *Geschwindigkeit* bliebe dabei aber auf der Strecke, stellte man sich ein Event als irgendwelche Strukturen à la X-Windows oder MacOS vor. Das ist jedoch in SPIN nicht der Fall.

Wie definiert man Events?

Programme in Modula-3 sind in Module aufgeteilt, die untereinander durch Prozeduraufrufe mittels Interfaces kommunizieren, so auch in SPIN, da sowohl das System als auch die Erweiterungen in Modula-3 geschrieben sind. Das **Aufrufen** einer Prozedur wird nun als das **Auslösen eines Events** betrachtet, das **Ausführen einer Prozedur** als **Handling eines Events**. Die Verteilung von Events (Prozeduraufrufe) an die entsprechenden Event-Handler übernimmt ein zentraler **Event-Dispatcher**.

Ein **Event-Handler** ist nun einfach eine Prozedur, die als Antwort auf ein entsprechendes Event ausgeführt wird. Diese Handler können dynamisch einem Event hinzugefügt oder davon entfernt werden, pro Event können auch mehrere Handler registriert werden.

Diejenige Prozedur, die denselben Namen wie das Event besitzt, bezeichnet man als *intrinsic handler*, und wird jedesmal dann aufgerufen, wenn das Ereignis ausgelöst wird. Hier setzt also auch die Erweiterbarkeit des Systems an. Ein Spindle kann zum Beispiel den intrinsic handler eines Events explizit deregistrieren und einen eigenen Handler für das Event beim Event-Dispatcher anmelden.

Guards

Jeder Event-Handler kann zusätzlich noch sogenannte **Guards** beim Event-Dispatcher registrieren lassen. Ein Guard definiert ein Prädikat, implementiert als eine Prozedur, dessen Wert vor dem Aufrufen des Handlers durch den Event-Dispatcher bestimmt wird. Ist das Prädikat wahr, so wird der Handler tatsächlich aufgerufen, andernfalls nicht. Pro Event können auch mehrere Guards registriert werden, sogar mit Forderungen an die Reihenfolge der Ausührung (z.B. als letzter). Guards müssen jedoch frei von irgendwelchen *Seiteneffekten* sein, dürfen z.B. nicht ihre Argumente speichern oder verändern. Der Dispatcher hat dadurch die Möglichkeit, Guards aus Performancegründen in ihrer Reihenfolge zu ändern, zudem spielen dabei Sicherheitsaspekte eine Rolle.

Event-Dispatcher

Die Verteilung der Events an die entsprechenden Handler übernimmt wie gesagt der Event-Dispatcher. Dies ist der natürlich der Flaschenhals des gesamten Systems, weswegen hier gewisse Optimierungsstrategien anwendet werden.

Im häufigen Fall, daß es für ein Event nur einen einzigen Handler gibt, wird dies als einfacher Prozeduraufruf implementiert. Sollte es mehrere Handler geben, so wird *dynamische Codegenerierung* eingesetzt, die dann den Code des *intrinsic-handlers* ersetzt. Hier werden auch Guards oder komplette Handler, die nur aus wenigen Befehlen bestehen, einfach ge-inlined. Dieser dynamisch erzeugte Code wird darüberhinaus auch diversen anderen Run-Time-Optimierungen unterzogen.

3.3 Weitere Sicherheitsaspekte

Die grundlegende Eigenschaft des Event-Modells in SPIN zeigt, daß jeder Prozeduraufruf potentiell an einen oder mehrere Handler umgeleitet werden kann, worauf Flexibilität und Erweiterbarkeit in SPIN beruhen. Natürlich darf dies nicht immer erlaubt sein, könnte doch jede Erweiterung das Verhalten eines Interfaces im Systems untergraben. Daher sind zusätzliche Sicherheitsmaßnahmen zu treffen, die jetzt beschrieben werden sollen.

Typprüfung

Während der Installation eines Handlers muß der Dispatcher die Aufruf- und Rückgabeargumente von Handlern und Guards auf Typengleichheit überprüfen. Die Argumenttypen des Handlers müssen mit denen des Events übereinstimmen. Ein Guard muß die gleichen Aufruftypen wie der Handler besitzen, als Rückgabetyp jedoch einen Boolean-Typ.

Agiert ein Handler als Filter, so muß er sein Argument *by reference* und nicht *by value* erhalten. Damit aber der Parameter des Event-Raisers nicht verändert wird, muß der Dispatcher vorher erst eine Kopie anfertigen, die dem Handler übergeben wird.

Zugriffskontrolle

Damit Erweiterungen Events nicht unerlaubt auslösen, gibt es sowohl statische als auch dynamische Zugriffskontrollen. So stellt sich die Frage, wer darf überhaupt ein Event auslösen, wer darf Handler eines bestimmten Events sein, wer kann den intrinsic-Handler verändern. Hier kommt der wichtige Begriff der *Event-Authority* ins Spiel.

Die *Event-Authority eines Events* ist immer das Modul, das den **intrinsic-handler** dieses Events implementiert. Existiert z.B. ein Modul M, das eine Prozedur M.P (bzw. Event M.P) implementiert, so ist M die Event-Authority von M.P.

Die Event-Authority spezifiziert **Autorisierungsfunktionen** für alle Operationen, die ein Event betreffen. Der Event-Dispatcher ruft diese Prozeduren immer dann auf, wenn sich bei den Handlern oder Guards dieses Events irgendetwas verändert.
Versucht also ein Modul einen Handler für ein Event zu installieren, wird die Event-Authority mit bestimmeten Informationen über das Modul kontaktiert. Sie kann jetzt die Installation einfach verbieten, oder erlauben, und dabei noch bestimmte Anforderungen stellen, so zum Beispiel, daß der neue Handler nur nach den bisher installierten Handlern ausgeführt werden darf.

Auch kann ein **zusätzlicher Guard** installiert werden, der überprüft, ob der neue Handler auch bestimmte Zugriffsrechte besitzt. Will eine Anwendung zum Beispiel eine Erweiterung zum direkten Empfang von Netzwerkpaketen installieren, so wird vom Netzwerkcode des Systems dieser Erweiterung ein Guard vorgeschaltet, der überprüft, ob die Port-Adresse der Pakete auch mit einer Port-Nummer übereinstimmt, die der Anwendung vorher vom System zugeteilt wurde. Wenn nicht, so wird das Paket nicht an das Spindle weitergegeben.

Ob eine Erweiterung überhaupt ein Event eines anderen Moduls aufrufen kann, wird schon vorher beim dynamischen Linken bestimmt. Will sich eine Erweiterung nämlich an ein bestehendes Modul im System dazulinken, wird dessen Autorisationsprozedur aufgerufen, die verhindern kann, daß irgendwelche Symbole (und damit auch Events) an die neue Erweiterung exportiert werden.

Als ein weiteres Sicherheitsproblem erweisen sich Handler, die dem Betriebssystem einfach “davonlaufen”.

Runaway-Handlers

Ein Handler, der niemals von seiner Ausführung zurückkommt, verhindert das Weiterlaufen des Event-Raisers bzw. das anderer Handler, und wird als *Runaway-Handler* bezeichnet. Da der Kernel selbst präemptiv ist, können andere Threads natürlich trotzdem weiter ausgeührt werden, dennoch sollen solche Situationen vermieden werden. Zwei Arten von Lösungen werden angeboten.

Zum einen gibt es dafür eine aufwendige Präventivmaßnahme: Das Event selbst oder ein spezieller Handler laufen **asynchron** ab, so daß der Event-Raiser nicht blockiert wird. Es wird dazu ein neuer Thread erzeugt, in dem alle Handler abgearbeitet werden. Auch können nur einzelne Handler asynchron ablaufen. Im Virtual-Memory-System von SPIN werden zum Beispiel asynchrone Events für Page-In-Requests benutzt, weil dabei die Zeit zum Erzeugen eines neuen Threads nicht allzu sehr ins Gewicht fällt. In SPIN dürfen darüberhinaus nur solche Events asynchron aufgerufen werden, die keine Argumente zurückliefern, andernfalls wird eine Exception erzeugt.

Die zweite Möglichkeit wird immer dann angwendet, wenn es auf Performance ankommt: Ein Handler, der zu lange zur Ausführung benötigt, wird einfach **terminiert**. Die erlaubte Zeitspanne wird hierbei durch die Event-Authority bestimmt. Natürlich sind mit diesem Verfahren

gewisse Sicherheitsprobleme verbunden, die die Integrität des Systems verletzen könnten, wenn ein Handler gerade eine kritische Datenstruktur updated oder gerade mitten in der Speicherallokation steckt.
Deswegen ist die Terminierung nur bei solchen Handlern möglich, die dies explizit erlauben. was durch das Schlüsselwort EPHEMERAL im Source-Code des Spindles geschieht. Es liegt im Ermessen der Event-Authority, solche EPHEMERAL-Handler auch wirklich zu terminieren, genauso kann sie aber auch die Installation eines Nicht-EPHEMERAL-Handlers verhindern.
Als Beispiel lassen sich die Netzwerk-Interrupt-Handler nennen, die Pakete nur an solche Handler weiterleiten, die als EPHEMERAL deklariert sind. Die Terminierung eines derartigen Handlers führt im schlimmsten Fall nur zum Verlust eines Paketes.

3.4 Beispiel zum Erweiterungsprozeß

Zur Verdeutlichung des gesamten Erweiterungsprozesses folgt nun ein kleines, sehr vereinfachtes Beispiel: Eine Anwendung will informiert werden, wann immer sie blockiert werden soll (durch den präemptiven Kern).

Im Kern existiert ein Interface *Strand,* das unter anderem eine Prozedur *Block* enthält, die vom System immer dann aufgerufen wird, wenn ein Thread blockiert werden soll (enspricht dem Auslösen eines Events). Ein Spindle der Anwendung, das in Modula-3 geschrieben wird, muß sich also als Event-Handler dieses Events *Block* registrieren lassen, was durch einen Aufruf an den Event-Dispatcher mittels *Dispatcher.InstallHandler()* geschieht.

Die Anwendung wird gestartet und versucht, ihr Spindle in den Kern zu integrieren. Der dynamische Linker von SPIN soll das Spindle zum Beispiel gegen die *logical-protection-domain SpinPublic* linken. Dazu wird die Autorisierungsprozedur von *SpinPublic* gefragt, ob das neue Modul gegen *SpinPublic* gelinkt werden darf. Dies wird der Fall sein, da grundsätzlich jedem Spindle Zugriff auf *SpinPublic* gewährt wird.

Ist der Link-Prozeß erfolgreich verlaufen, ruft unser Spindle die oben erwähnte Installationsprozedur auf mittels *Dispatcher.InstallHandler(Strand.Block,myGuard,myHandler)*, die von *SpinPublic* exportiert wird. Der Event-Dispatcher kontaktiert nun die Event-Authority von Strand.Block, das ist das Modul Strand, mit der Information, daß unser Spindle *myHandler* als Handler für Block registrieren lassen will, mit *myGuard* als Guard.

Wenn dies erlaubt wird, so generiert der Event-Dispatcher noch zusätzlichen Run-Time-Code (z.B. kann unser Guard in der Dispatch-Routine aus Performancegründen ge-inlined werden), und bei entsprechendem Guard wird unser Spindle nun jedesmal aufgerufen, wenn die Anwendung blockiert werden soll.

4. Anwendungsgebiete von SPIN

Nachdem jetzt der Aufbau, die Architektur und die Grundkonzepte von SPIN beschrieben wurden, stellt sich die Frage, in wieweit SPIN seine Zielsetzung erreichen kann, für die Aufgaben der kommenden Generation von Anwendungen gerüstet zu sein.

Gerade Anwendungen wie Multimedia- oder Datenbanksysteme stellen an das Betriebssystem hohe Anforderungen, wenn auch zum Teil recht unterschiedliche. So können Seitenersetzungsstrategien oder Filesysteme, die für Multimedia geeignet sind, Datenbanksysteme fast zur Unerträglichkeit verlangsamen. Auch wäre es für manche Applikationen von Vorteil, könnten sie ihr eigenes Thread-Management implementieren. Es folgen einige Beispiele, wie solche Anforderungen in SPIN mittels anwendungsspezifischer Dienste realisiert werden können.

Protokollverarbeitung auf Anwendungsebene

Die Implementierung höherer Kommunikationsprotokolle wie zum Beispiel TCP/IP kann gänzlich auf Anwendungsebene in einer Library erfolgen. Ein Low-Level-Spindle, das im Kern direkt vom Gerätetreiber (z.B. Ethernet) mit Netzwerkpaketen versorgt wird, verteilt diese weiter an die Anwendung.

Anwendungsspezifisches Buffer-Cache-Managament und Filesystem

Ein Buffer-Cache-Manager für ein Filesystem kann als Library implementiert werden, die Anwendungen den Zugriff auf *Daten im Hauptspeicher* durch einen simplen Prozeduraufruf gestattet, ohne daß dabei Daten kopiert werden müssen. Die Applikation kann so auch ihre eigene Buffer-Cache-Strategien verwirklichen, um eine hohe Cache-Trefferquote zu erhalten.

User-Level-Scheduling

In SPIN kann ein kompletter User-Level-Scheduler durch ein Spindle, das pro Anwendung installiert wird, implementiert werden. Diese Spindle wird bei jedem Kontext-Switch eines Kernel-Level-Threads aufgerufen, es wird also aktiviert, wenn die dazugehörige Anwendung blockiert bzw. deblockiert wird. Es kann dann eine Message, die den jeweiligen Zustand signalisiert, an die Anwendung weitergeben, woraufhin diese anwendungsspezifisches Thread-Management emulieren kann.

Anwendungsspezifisches Virtual-Memory

Dies kann so implementiert werden, daß eine Anwendung direkt mit physikalischen Speicherseiten versorgt wird. Dazu kann ein Spindle erzeugt werden, das das Low-Level-Mapping zwischen physikalischer und virtueller Adresse vornimmt, und ein weiteres, das Page-Faults an die Anwendung weitermeldet.

5. Performance und Benchmarks von SPIN

Nachdem SPIN ein komplettes, lauffähiges Betriebssystem ist, bietet es sich an, einmal die Performancewerte mit denen von etablierten Systemen zu vergleichen. Bei diesen handelt es sich einerseits um Mach 3.0, ein System mit einem Microkernel, und andererseits um DEC OSF/1 V2.1, ein monolithisches System. Alle drei Systeme wurden im Single-User-Mode auf den gleichen Maschinen getestet, DEC Alpha AXP 3000/400 Workstations mit 133 MHz.

5.1 Microbenchmarks

Mircobenchmarks sollen den *Overhead* von grundlegenden Betriebssystemdiensten, wie zum Beispiel eines geschützten Prozeduraufrufs, des Thread-Managements oder des Virtual-Memory-Managements, aufzeigen.

Geschützte Kommunikation

Hierbei unterscheidet man zwischen *System-Calls* und *Cross-Address-Space-Calls*. Letztere stellen in einem konventionellen System wegen ihres großen Overheads gerade den Hinderungsgrund für die Erweiterbarkeit des System dar. In SPIN gibt es jedoch noch einen weiteren Mechanismus, den *protected in-kernel-call*. Dieser resultiert aus SPINs Erweiterungsmodell und erlaubt, die Kommunikation zwischen dem Kernsystem und den Erweiterungen als einfachen Prozeduraufruf ablaufen zu lassen.

Tabelle 1 zeigt die jeweils gemessenen Zeiten, die Aufrufe sind dabei immer **ohne** Argumente, um nur die reine Kommunikationszeit zu erhalten. Man erkennt, daß in SPIN auch traditionelle

Mechanismen nicht-erweiterbarer Systeme implementiert werden können. Wenn möglich sollte man allerdings immer In-Kernel-Extensions vorziehen, da der Overhead dabei auf den eines einfachen Prozeduraufrufes zusammenschrumpft.

Operation	DEC OSF/1	Mach 3.0	**SPIN**
Protected in-kernel call	nicht verfügbar	nicht verfügbar	.13
System-call	5	7	4
Cross-address-space-call	845	104	89

Tabelle 1: Overhead von geschützter Kommunikation in Mikrosekunden

Thread-Management

In der folgenden Tabelle 2 werden die Zeiten für den Overhead typischer Thread-Operationen aufgezeigt, wobei sowohl Kernel-Level als auch User-Level-Threads betrachtet werden. *Fork-Join* mißt die Zeit, einen Thread zu erzeugen, zu schedulen und wieder zu beenden, letzteres in Synchronisation mit einem anderen Thread. *Ping-Pong* mißt den Overhead bei der Synchronisation eines Threadpaares. Ein Thread schickt dem anderen ein Signal und blockiert sich, der andere signalisiert dem ersten und blockiert sich dann auch.

Es sei noch darauf hingeweisen, daß es in SPIN zwei Arten gibt, User-Level-Threads zu implementieren: Zum einen als User-Level-Library, die sich einer Erweiterung bedient, welche ein *Mach-Kernel-Thread-Interface* implementiert, zum anderen als applikationsspezifisches Spindle (siehe auch 4.).

<table>
<tr><th rowspan="3">Operation</th><th colspan="2">DEC OSF/1</th><th colspan="2">Mach</th><th colspan="3">SPIN</th></tr>
<tr><th rowspan="2">kernel</th><th rowspan="2">user</th><th rowspan="2">kernel</th><th rowspan="2">user</th><th rowspan="2">kernel</th><th colspan="2">user</th></tr>
<tr><th>layered</th><th>integrated</th></tr>
<tr><td>Fork-Join</td><td>198</td><td>1230</td><td>101</td><td>338</td><td>22</td><td>262</td><td>111</td></tr>
<tr><td>Ping-Pong</td><td>21</td><td>264</td><td>71</td><td>115</td><td>17</td><td>159</td><td>85</td></tr>
</table>

Tabelle 2: Overhead des Thread-Managements in Mikrosekunden

Für weitere Benchmarks, z.B. Virtuelle-Speicherverwaltung und Netzwerkkommunikation, sei auf die Literatur verwiesen.

5.2 Performance-Vorteile von SPIN

Die überdurchschnittlich guten Performancewerte von SPIN beruhen auf den folgenden Eigenschaften des Erweiterungsmechanismuses:

- Erweiterungen bieten spezialisierte Dienste an
  ➡ keine Ausführung von unnötigem Code
- Erweiterungen nah an den Kernel-Diensten
  ➡ kurze Verzögerungszeiten bei Faults und Interrupts
  ➡ Aufruf von anderen Diensten nicht teuer

6. Zusammenfassung und Ausblick

Am Beispiel von SPIN wird gezeigt, daß ein modernes, erweiterbares Betriebssystem sehr hohe Performance aufweisen kann, ohne dabei Sicherheitsaspekte zu vernachlässigen, wie dies so oft der Fall ist.

SPIN befindet sich noch in der Entwicklung an der Universität Washington und soll dort bald als eigenständiges Betriebssystem eingesetzt werden. Es wird in seiner ursprünglichen Form auch anderen Universitäten und Forschungseinrichtungen zur Verfügung gestellt werden.

Darüberhinaus besteht auch die Absicht, bestimmte Schlüsselmechanismen von SPIN auf andere kommerzielle Betriebssysteme, wie zum Beispiel OSF/1 und Windows-NT, zu übertragen. Mit zunehmender Verbreitung dieser Systeme geht auch die Notwendigkeit einher, den Anwendungsprogrammieren mehr Möglichkeiten zu feingliedrigem Ressourcenmanagement zu bieten.

7. Literatur

Die im folgenden aufgeführte Literatur zu SPIN läßt sich auch über die **SPIN-Homepage** unter *http://www-spin.cs.washington.edu* online abrufen.

- ❑ P. Pardyak and B. Bershad. Dynamic Binding in an Extensible System. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI) 1996*, pages 201-212 OSDI96 Paper.
- ❑ M. Fiuczynski and B. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the 1996 Winter USENIX Conference*, San Diego, CA., pages 55-64, January 1996.
- ❑ B. Bershad, S. Savage, P. Pardak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Copper Mountain, CO., pages 267-284. SOSP95 Paper
- ❑ J. Auslander, M. Philipose, C. Chambers, S. Eggers, B. Bershad. Fast Effective Compilation, *Conference on Programming Language Design and Implementation*, May 1996.
- ❑ E. Sirer, M. Fiuczynski, P. Pardyak, and B. Bershad. Safe Dynamic Linking in an Extensible Operating System. In *Proceedings of the First Workshop on Compiler Support for Systems Software*, pages 141-148, February 1996.