Dynamische Adaption in Betriebssystemen – Das CHEOPS[1]-Projekt

Sven Graupner, Winfried Kalfa, Frank Schubert,
Ronny Vogel, Jörg Werner, Lutz Wohlrab

Technische Universität Chemnitz-Zwickau
Fakultät für Informatik, Professur Betriebssysteme,
09107 Chemnitz
Tel.: (+49)-371-531-1390 / Fax.: (+49)-371-531-1530
Email: {sgr,kal,fsc,rvo,jwer,lwo}@informatik.tu-chemnitz.de
http://www.tu-chemnitz.de/informatik/osg

Kurzfassung

Derzeit wird Anpaßbarkeit von Betriebssystemen breit untersucht. Damit wird der zunehmendenen Bedeutung maßgeschneiderter Systeme Rechnung getragen. Die in diesem Beitrag vorgestellten Arbeiten verfolgen daher das Ziel, Erweiterbarkeit und Konfigurierbarkeit auf allen Ebenen eines Betriebssystems zu erreichen. Zunächst wird ein Vorschlag für ein flexibles Architekturmodell unterbreitet, das durch eine rekursive Hierarchie von diensterbringenden Instanzen gekennzeichnet ist. Instanzen einer Ebene bilden dabei die Infrastruktur für Instanzen der nächsthöheren Ebene. Anschließend wird gezeigt, wie mit Hilfe objektorientierter Methoden und Mechanismen die dynamische Adaption von Instanzen realisiert werden kann. Grundgedanke hierbei ist die Beibehaltung identifizierbarer Einheiten aus der Systembeschreibung auch während der Laufzeit. Abschließend wird ein alternatives Konzept zur Verwaltung der Adaptionsvorgänge diskutiert, welches durch die feingranulare Struktur und die wachsende Komplexität des Systems erforderlich wird.

---

1. Einführung

Anpaßbarkeit wird sich immer mehr zu einer Schlüsseleigenschaft für Betriebssysteme entwickeln. Gründe dafür sind:

- zunehmende *Vielgestaltigkeit* von Hardware sowie Anwendungen (z. B. mobile Systeme mit knappen Ressourcen, Steuerungen für Geräte des alltäglichen Gebrauchs),
- fortschreitende *Spezialisierung* und Aufgabenteilung auch im Bereich traditioneller Universalrechner (z. B. Compute-Server, File-Server), forciert durch deren Vernetzung,
- *Effizienz* von Systemen als Wettbewerbsfaktor durch direkte Nutzung spezifischer Hardware und angepaßte Ressourcenverwaltung für individuelle Anwendungen.

[1] **CHE**mnitz **OP**erating **S**ystem, Teil des von der DFG geförderten Forschungsprojektes *„Untersuchung rekursiver und reflexiver Methoden zur Anpassung von objektorientierten verteilten Betriebssystemen an unterschiedliche Hardware und Software“*.

Die sich hieraus ergebenden Probleme lassen sich mit *statischer* Anpaßbarkeit lösen. Diese kann mit Frameworks (z. B. *Tigger* [Cahill94]) erreicht werden, welche Bausätze für Betriebssysteme darstellen. Als Ergebnis erhält man Betriebssystemfamilien, wobei jedes einzelne Betriebssystem auf genau bestimmte Anforderungen zugeschnitten ist.

Darüber hinaus ist *dynamische* Anpaßbarkeit wünschenswert bei Systemen:

- mit hoher Verfügbarkeit, um auf Veränderungen im System (z. B. Ressourcenverknappung) mit verändertem Verhalten reagieren zu können,
- bei denen durch wechselnde Anforderungen und zeitaufwendigen Neustart statisch zugeschnittene Betriebssysteme nicht in Frage kommen.

Über dynamische Anpaßbarkeit hinausgehende Selbstanpassung wird als *Adaptivität* bezeichnet. In diesem Fall reagiert ein System selbsttätig auf beobachtete Anforderungsprofile und Systemzustände.

Damit ein System überhaupt anpaßbar gestaltet werden kann, ist eine Reihe von Voraussetzungen zu erfüllen:

- klare, detailliert offengelegte, weitgehend homogene Systemstruktur,
- handhabbare Schnittstellen und Mechanismen, die zur Anpassung genutzt werden können,
- alternativ einsetzbare Module,
- Metainformationen über Module (Eigenschaften der implementierten Dienste, z. B. Kompatibilität zu einer älteren Version).

Erst wenn diese Voraussetzungen erfüllt sind, können ohne aufwendige Neuprogrammierung speziell zugeschnittene Betriebssysteme zusammengestellt und Anpaßbarkeit erreicht werden.

Im Bestreben, die genannten Voraussetzungen zu erfüllen und Adaptierbarkeit oder sogar Adaptivität zu erreichen, versuchen Betriebssystementwickler derzeit, Vorteile erfolgreicher Technologien aus anderen Informatikbereichen für ihre Ziele zu nutzen, beispielsweise Objektorientierung. Der effektive Bau von Frameworks (s. o.) ist heute ohne die Nutzung dieser Technologie kaum noch vorstellbar, da man selten auf ihre Vorteile (siehe u. a. [Coad91]) verzichten möchte:

- Datenkapselung und -abstraktion,
- Wiederverwendbarkeit von Code,
- Polymorphie und
- Erweiterbarkeit.

Um feingranulare Anpaßbarkeit zu erreichen, wäre das Objekt als kleinste Einheit für Adaptionsvorgänge besonders geeignet. Jedoch stehen objektorientierte Strukturierungsmechanismen i. allg. nur auf Quelltextebene zur Verfügung.

Einige Systeme bieten zur Laufzeit die Möglichkeit, Objekte zu verwalten (z. B. *BirliX* [Härtig90], *Clouds* [Dasgupta91]). Diese Objekte und ihre Struktur weichen von dem auf der Quelltextebene Üblichen jedoch erheblich ab. Außerdem stehen keine Informationen über die Klassenzugehörigkeit von Objekten, über die Klassen und deren Eigenschaften sowie über die Klassenhierarchie und die sich daraus ergebenden Verwandtschaftsbeziehungen von Objekten zur Verfügung.

Um die Vorteile objektorientierter Methoden im laufenden System zu nutzen, müssen

- Informationen über Klassen, Klasseneigenschaften, Zugehörigkeit von Objekten zu Klassen und Verwandtschaftsbeziehungen von Objekten im laufenden System verfügbar gemacht werden und
- identifizierbare Verantwortlichkeiten geschaffen werden, die mit Objekterzeugung, Objektzerstörung usw. die Aufgaben der sonst anonymen Laufzeitumgebung übernehmen.

Hiermit würde eine möglichst vollständige Abbildung der objektorientierten Entwicklungswelt in die reale Welt des laufenden Systems erreicht. Dies erleichtert, Systeme zu schaffen, die durch Hinzufügen, Austauschen und Entfernen von Klassen und Objekten an neue Erfordernisse angepaßt werden können.

Außer der Objektorientierung erlangen Technologien und Abstraktionen aus der Künstlichen Intelligenz zunehmende Bedeutung bei der Entwicklung von Betriebssystemen, beispielsweise Meta-Level-Architekturen und Reflexion (z. B. in *Apertos* [Yokote93]) sowie Expertensysteme zur Unterstützung von Systemadministratoren.

Betriebssysteme sind große Softwaresysteme. Deshalb würde ein System, das sich ausschließlich auf feingranulare Strukturen wie das Objekt stützt, der Forderung nach einer guten, überschaubaren Struktur nicht entsprechen. Daher sind in einem Betriebssystem größere Strukturierungseinheiten, z. B. Objektgruppen oder Schichtenstrukturen, unverzichtbar.

2. Ein rekursives Architekturmodell und seine prototypische Realisierung

Eine speziell auf ein Zielsystem zugeschnittene Architektur ist oft nicht auf andere Systeme übertragbar. Will man daher generell Anpaßbarkeit und Spezialisierbarkeit erreichen, setzt dies ein gewisses Maß an Universalität, d. h. Übertragbarkeit auch der Architektur, voraus. Ziel ist aber nicht eine universelle Architektur für jedes konkrete Zielsystem, sondern eine spezielle, zugeschnittene Architektur, die es ermöglicht, dessen Vorteile weitestgehend auszunutzen. Dieser Widerspruch läßt sich durch explizite Trennung beider Aspekte in einer *spezialisierbaren Systemarchitektur* auflösen.

Die Idee ist, übertragbare strukturelle und methodische Grundzüge in einer *generalisierten Architektur* festzuschreiben. Für ein konkretes Zielsystem wird diese dann spezialisiert, d. h. konkrete Ausprägungen von Elementen, Infrastrukturen und Umgebungen werden benannt.

Im folgenden Unterabschnitt wird die generalisierte Architektur vorgestellt, die dann beispielhaft für ein konkretes Zielsystem, den CHEOPS-μ-Kern, spezialisiert wird. Spezialisierung heißt, es werden Ausführungseigenschaften (Infrastruktureigenschaften) von Instanzen für jede Ebene hinzugefügt, beispielsweise die Beziehung zu Adreßräumen oder die Lebensdauer bestimmter Elemente.

Diese Trennung eröffnet die Chance, eine universelle Vorgehensweise für die Konstruktion von Systemen anzuwenden und diese dann für ein konkretes Zielsystem angepaßt umzusetzen, indem erst bei der Spezialisierung konkrete, zugeschnittene Eigenschaften von Elementen und Infrastrukturen festgelegt werden.

Traditionell sind beide Aspekte vermischt, und es werden mit der Architektur auch das Ausführungsmodell und notwendige Infrastruktureigenschaften festgelegt. Für *BirliX* [Härtig90] heißt das beispielsweise, daß Anwendungen aus Mengen interagierender ADT-Instanzen bestehen, die separate Adreßräume besitzen, multi-threaded sind, über RPC kommunizieren und persistent sind. Hier ist in der Architektur bereits alles festgelegt, eine Übertragung etwa auf den Apple-Newton wäre aus Ressourcengründen unmöglich. Universalität wird in *BirliX* angestrebt, indem der jeweils allgemeinste und oft teuerste Mechanismus für eine Eigenschaft umgesetzt wird (multi-threaded schließt single-threaded ein, lokale Kommunikation ist ein Spezialfall entfernter Kommunikation, Persistenz ist der allgemeine Fall für die Existenz einer ADT-Instanz).

2.1 Generalisierte Architektur

Informationsverarbeitende Systeme bestehen i. allg. aus Teilsystemen, die selbständig Teilaufgaben (Dienste) ausführen. Für das Zusammenwirken der Teilsysteme sind Steuerung und Informationsaustausch (Protokolle) erforderlich. Verarbeitung und auch Steuerung sind inhärent aktive (Verarbeitungs-) Prozesse, die ihrerseits einer Steuerung bedürfen. Daraus resultiert eine Hierarchie von Steuerung und Verarbeitung, d. h. Aufträge aus der Umgebung an das informationsverarbeitende System werden intern (rekursiv) in Teilaufgaben für Steuerung und Verarbeitung für Teilsysteme zerlegt. Die Modellierung dieser Abläufe kann man beschreiben:

- Komplexe informationsverarbeitende Systeme werden in abgrenzbare Teilsysteme für Teilverarbeitungen (*Instanzen*) zerlegt, die auf einer *Infrastruktur* aufsetzen. Informationsverarbeitung kann vollständig als Erbringung von *Diensten* durch Instanzen eines Systems modelliert werden.

- Instanzen sind zu selbständiger Verarbeitung in der Lage, was innere Aktivität und deren Steuerung voraussetzt. Eine Instanz kann wiederum Dienstausführungen in anderen Instanzen oder in der Infrastruktur initiieren. Die äußere Steuerung einer Instanz wird durch andere Instanzen oder Ereignisse veranlaßt, vermittelt über die Infrastruktur („was“, „wann“). Die innere Steuerung der Aktivität („wie“) bezieht sich auf die Funktion der Diensterbringung durch die Instanz.

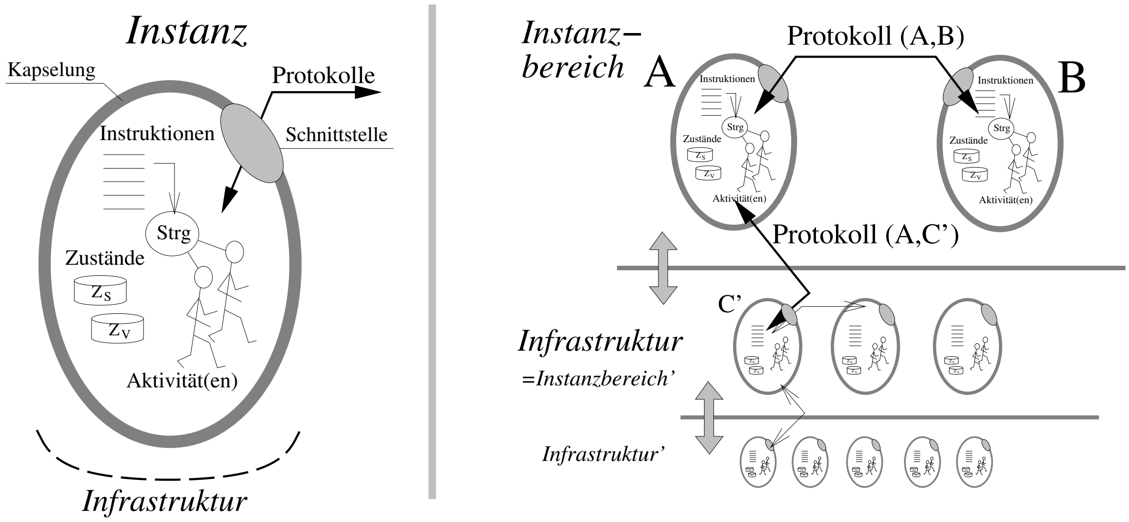


Abbildung 1: Modellierung von Instanzen auf mehreren Ebenen

- Zwischen Instanzen und zur Infrastruktur sind *Protokolle* für Steuerung und für den Austausch von Aufträgen und Ergebnissen vereinbart. Steuerung ist selbst ein gesteuerter Verarbeitungsprozeß anderer Instanzen.

- Infrastrukturen bestehen selbst wieder aus Instanzen einer elementareren Art, deren Eigenschaften durch eine untergeordnete Infrastruktur bestimmt werden (*Rekursion*). Die Rekursion endet mit den Hardware-Instanzen, die die unterste Infrastruktur bilden.

- (Software-) Instanzen weisen über alle Ebenen gemeinsame Merkmale auf:

  - Instanzen sind abgrenzbar und damit *identifizierbar*, bezogen auf:
    - ▷ innere Zustände,
    - ▷ Steuerungsinformationen (Methoden) und
    - ▷ innere Aktivitäten.

    Die technische Umsetzung der Kapselung ist Gegenstand der Spezialisierung[2].
  - Instanzen besitzen aufgrund selbständiger Verarbeitung mindestens eine innere *Aktivität*, deren Ausprägung ebenfalls Gegenstand der Spezialisierung ist.
  - (innere) Steuerung und Verarbeitung sind *zustandsbehaftet*.
  - Instanzen besitzen zur Abwicklung von Protokollen eine *Schnittstelle*.
  - Die Herstellung einer (Software-) Instanz erfolgt innerhalb des Systems anhand von Beschreibungsinformationen (*Typ-Informationen*), die ebenfalls herzustellen sind.

Instanzen sind als Ausführungseinheiten prinzipiell bekannt: *Unix-Prozeß*, *Mach-Task*, *BirliX-Team* u. a. Die Neuerung besteht darin, von konkreten Ausführungseigenschaften für die generalisierte Architektur zu abstrahieren und damit dieselbe, bekannte Grundabstraktion homogen über alle Ebenen des Systems legen zu können. In der unten beschriebenen Fallstudie wird gezeigt, wie eine Spezialisierung der generalisierten Architektur für einen μ-Kern aussehen kann.

[2]beispielsweise separate Adreßräume oder Restriktion der Adressierung bei der Steuerung innerer Aktivitäten

2.2 Fallstudie: CHEOPS-μ-Kern

Anhand der Implementation des CHEOPS-μ-Kerns (Prototyp) wird eine Spezialisierung der vorgestellten generalisierten Architektur angegeben und gezeigt, welche Eigenschaften sich für das System daraus ergeben. Die Grundeigenschaften des μ-Kerns sind an traditionelle μ-Kerne (*Mach*, *Chorus*) angelehnt, mit einigen Unterschieden:

- dynamische (Re-) Konfigurierbarkeit des Kerns durch Hinzufügen und Entfernen von Instanzen im Kern,
- individuelle Kernschnittstellen für verschiedene Anwendungen.

2.2.1 Spezialisierung der Architektur für einen μ-Kern

Für einen μ-Kern haben sich vier Ebenen als sinnvoll erwiesen (die Markierungen **(U)** und **(K)** geben die Plazierung im User- oder Kernbereich an):

$\boxed{4}$ **(U) aproc's:** Anwendungsinstanzen mit separaten Adreßräumen, preemptivem Scheduling, IPC (vgl. *Unix-Prozesse*, *Mach-Tasks*), Nachrichtenkommunikation usw.

$\boxed{3}$ **(K) kproc's:** Kerninstanzen für Gerätetreiber, Speicherverwaltung (VMM), die Verwaltung von aproc's. Diese Instanzen befinden sich im Kernadreßraum und sind vergleichbar mit den sogenannten in-kernel-Servern [Broner95]. Sie arbeiten zyklisch Dienstanforderungen ab. Interaktion erfolgt über Nachrichten bzw. Signale. kproc's sind dynamisch startbar (vergleichbar zu Prozessen auf Unix, hier aber im Kern), wodurch dem Kern neue Dienste hinzugefügt werden. kproc's können auch dynamisch wieder entfernt werden.

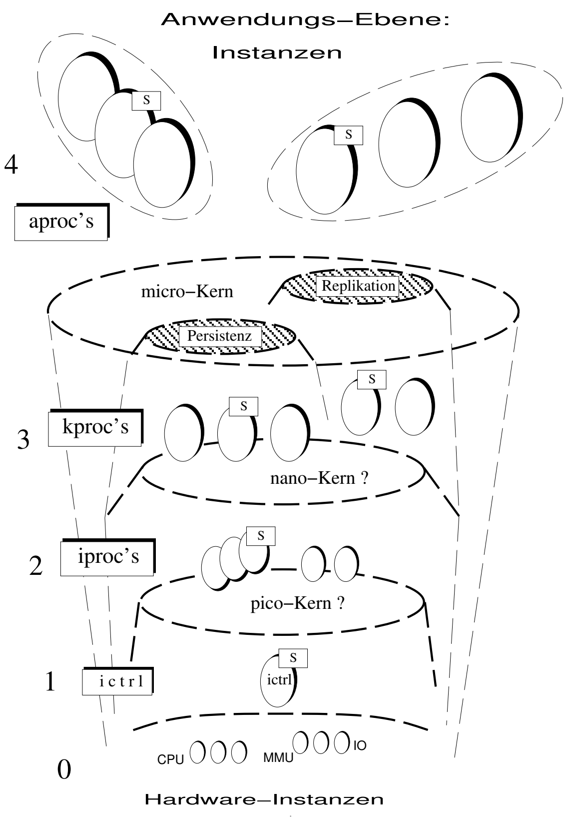


Abbildung 2: CHEOPS-μ-Kern

$\boxed{2}$ **(K) iproc's:** Jeder Interrupt wird von einer neu erzeugten iproc-Instanz behandelt. iproc's befinden sich wie kproc's im Kernadreßraum, ihre Verwaltung und Ablaufsemantik ist aber auf Interrupt-Behandlung spezialisiert. Es ist effektiver, für jeden Interrupt einen iproc neu zu erzeugen und nach der Behandlung wieder zu vernichten, als eine zyklische Instanz für die Behandlung eines Interrupts „aufzuwecken". Ein ähnliches Verfahren wird in *Solaris 2* verwendet [Kleiman95].

$\boxed{1}$ **(K) ictrl:** Es gibt im Kern genau eine Instanz, ictrl, bei der alle Interrupts eintreffen. Sie erzeugt für jeden eintreffenden Interrupt einen iproc und steuert dessen Abarbeitung.

$\boxed{0}$ **Instanzen der Hardware:** Abbruch der Rekursion von Ebenen für den μ-Kern.

2.2.2 Implementation

Der CHEOPS-μ-Kern wird als stand alone System auf zwei Maschinentypen implementiert: CADMUS 9700 Workstation und PC (protected mode). Die Ebenen ictrl, iproc und kproc-Verwaltung sind realisiert, diverse kproc-Instanzen werden derzeit implementiert (VMM, disk, aproc-Verwaltung usw.). Zur Herstellung der Typbeschreibungen wird C++ verwendet. Es wurde eine Shell für die Dialogfähigkeit des Systems auf der Ebene von kproc's zu Testzwecken hergestellt. Weiterhin wurden eine Bibliothek für C++ und ein Laufzeitsystem für C++ geschaffen.

3. Objektorientierung als Basis für feingranulare Adaption

Im zweiten Abschnitt wurde gezeigt, wie mit einer geeigneten Systemarchitektur eine Grundvoraussetzung für dynamische Adaptierbarkeit erfüllt werden kann. In diesem Abschnitt soll nun konkretisiert werden, wie darauf aufbauend Rekonfigurierbarkeit und Erweiterbarkeit in allen Ebenen des Betriebssystems auf der Basis objektorientierter Mechanismen und Methoden erreicht werden können. Dabei soll von dem eingangs formulierten Ziel ausgegangen werden, die objektorientierte Beschreibung des Systems möglichst vollständig im arbeitenden System zu erhalten.

3.1 Objekte und Aktivitäten

In einer Vielzahl von Systemen werden Objekte auf sehr unterschiedliche Weise definiert und benutzt. Oft werden Objekte im Umfeld der objektorientierten Methoden als Instanzen abstrakter Datentypen (Klassen) betrachtet, die Daten kapseln und Operationen (Dienste) über diesen Daten bereitstellen. Die Benutzung dieser Operationen abstrahiert dabei von der Struktur der gekapselten Daten. Existieren diese Eigenschaften nicht nur innerhalb der objektorientierten Beschreibung des Systems und werden Objekte innerhalb des arbeitenden Systems als identifizierbare Einheiten verwaltet, können diese als Grundelemente für

- Diensterbringung,
- Austausch (Rekonfiguration) und
- Migration

genutzt werden.

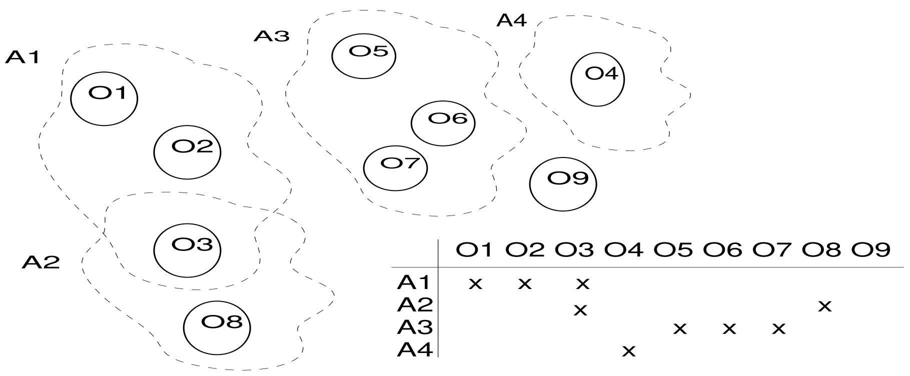
Diese allgemeine Objektdefinition liegt auch den hier angestellten Betrachtungen zugrunde. Objekte sind damit zunächst passiv. Für den *Benutzer* eines Objektes scheint dieses jedoch die diensterbringende Instanz im System zu sein. Um einen Dienst erbringen zu können, muß das die entsprechende Handlungsvorschrift enthaltende Objekt aktiviert werden. Die Art und Weise, auf die Objekte aktiviert werden, sollte von den Erfordernissen des konkreten Systems abhängen und kann unterschiedlich sein:

- In der einfachsten Form wird jedem Objekt bei seiner Erzeugung genau eine Aktivität zugeordnet.
- Für komplexe Objekte ist oft auch eine Parallelisierung der objektinternen Abläufe wünschenswert (Intraobjektparallelität).
- Sind von vornherein sequentielle Abläufe innerhalb kooperierender Objekte bekannt, ist es sinnvoll, dies durch die Einschränkung der möglichen Interobjektparallelität zu unterstützen. Auf diese Weise werden die Anzahl kritischer Abschnitte verringert und der Verwaltungsaufwand für Aktivitäten reduziert.

Deshalb werden Aktivitäten als eigenständige Entitäten im System betrachtet. Eine Aktivität ist ein identifizierbarer Handlungsablauf (Steuerfluß) im System. Aktivitäten können zueinander parallel verlaufen (impliziert Prozessorzuteilung).

Während Aktivitäten Handlungsabläufe im System darstellen, enthalten Objekte die dafür nötigen Handlungsvorschriften. Aktivitäten und Objekte stehen damit in Relation zueinander.

In einem konkreten System muß geklärt werden, welche Relationen (ggf. Kombinationen aus verschiedenen Varianten) existieren dürfen und auf welche Weise diese zu verwalten sind. Im Laufe der Arbeit eines Systems kann es außerdem erforderlich sein, bestehende Relationen zu modifizieren.



	O1	O2	O3	O4	O5	O6	O7	O8	O9
A1	x	x	x						
A2			x					x	
A3					x	x	x		
A4				x					

Abbildung 3: Varianten der möglichen Aktivitäts-Objekt-Relationen

Abbildung 3 zeigt eine mögliche Situation, in der sowohl Inter- als auch Intraobjektparallelität vorkommen. Die entsprechenden Relationen könnten auf einfache Weise in einer Matrix verwaltet werden. Diese Matrix kann für unterschiedliche Auswertungen genutzt werden. Alle möglichen Inter- und Intraobjektparallelitäten sind durch die aktuelle Relationsmatrix bestimmbar. Objekte, die nicht in Relation zu einer Aktivität stehen, erbringen zur Zeit keine Dienste. Damit ist deren Zustand allein durch die Zustände des Datenbereichs definiert und somit bestimmbar. Dies ist eine Vorraussetzung für den Austausch oder die Migration von Objekten [Schubert95].

Die einfachste Form einer Aktivitäts-Objekt-Relation ist die Zuordnung einer Menge von Objekten zu jeweils einer Aktivität. Dies entspricht einer aktiven Instanz aus dem zweiten Abschnitt. Ein Vorteil besteht darin, daß aus globaler Sicht relativ komplexe Instanzen existieren können und somit durch die begrenzte Anzahl dieser Instanzen das System überschaubar bleibt. Innerhalb einer Instanz existieren jedoch weiterhin Verwaltungsinformationen über Objekte. Diese Informationen entsprechen zudem weitgehend denen der Beschreibungswelt und unterstützen somit feingranulare Adaption und Rekonfiguration durch den Austausch von Objekten innerhalb der Instanzen.

3.2 Das Konzept der Klassenobjekte

Die Verwaltung von Objekten im laufenden System ist nicht ausreichend, um die in Abschnitt 1 gezeigten Voraussetzungen für die Nutzung objektorientierter Methoden im laufenden System zu erfüllen. Es ist notwendig, zusätzliche Informationen über Klassen, Klassenzugehörigkeit von Objekten, Klassenhierarchie usw. zu verwalten und bisher anonyme Dienste identifizierbaren Dienstleistern im System zuzuordnen, um die aus Klassen und Objekten bestehende Welt der Beschreibung in das laufende System abzubilden. Damit stehen gleiche Komponenten in beiden Welten zur Verfügung. Änderungen, die in der Beschreibungswelt vorgenommen wurden (Weiterentwicklung), können dann auch dynamisch im laufenden System vollzogen werden.

Um dies zu erreichen, wird das Konzept der Klassenobjekte benutzt. Ein *Klassenobjekt* ist die Repräsentation einer Klasse zur Laufzeit. Es ist ein im System bekanntes und identifizierbares Objekt und erbringt klassenspezifische Dienste für die Objekte der jeweiligen Klasse. Die Aufgaben eines Klassenobjektes sind:

- Erzeugen und Zerstören von Objekten der Klasse,
- Verwaltung der Objekte der Klasse, Speichern klassenspezifischer Informationen,
- Dienstvermittlung zu Objekten der Klasse,
- Unterstützung des typsicheren Objektaustausches unter Ausnutzung der bekannten Verwandtschaftsbeziehungen (Polymorphie).

Die Klassenobjekte sind somit ein Teil der Infrastruktur von Objekten. Die Dienste der Klassenobjekte werden dabei entweder durch explizite Anforderung (z. B. Erzeugen und Zerstören von Objekten) oder implizit (z. B. Vermittlung ortstransparenter Dienste) von Objekten in Anspruch genommen.

Das Konzept der Klassenobjekte ist in ähnlicher Form bereits aus SmallTalk [Goldberg83] bekannt. Die in SmallTalk zugrundegelegte interpretative Arbeitsweise ist jedoch für die Anwendung im Betriebssystem zu ineffizient. Daher muß auf in der Systemsoftwareentwicklung bewährte Sprachen und Compiler zurückgegriffen werden. Die dort eventuell nicht verfügbaren Mechanismen für die Verwaltung von Klassenobjekten müssen nachgebildet werden.

Verwaltung der Klassenobjekte

Für die Verwaltung der Klassenobjekte und der sich dynamisch ändernden Klassenhierarchie erscheint es sinnvoll, einen zentralen Ankerpunkt, den Klassenobjektverwalter, einzuführen. Abbildung 4 zeigt die somit entstehende Hierarchie von Objekten im System. Ausgangspunkt dieser Hierarchie ist dabei der Klassenobjektverwalter. Dieser Verwalter ist ebenfalls ein Objekt. Er muß jedoch von Anfang an im System existieren und in Relation zu einer Aktivität stehen.

Die Aufgaben des Klassenobjektverwalters bestehen vor allem im

- dynamischen Laden von neuen Klassen, d. h. Erzeugen von neuen Klassenobjekten,
- Entfernen von Klassen aus der Hierarchie (mit Prüfung der Zulässigkeit),
- Verwalten der Klassenhierarchie.

In Analogie zur Bereitstellung infrastruktureller Dienste für Objekte durch deren Klassenobjekte stellt der Klassenobjektverwalter infrastrukturelle Dienste für Klassenobjekte zur Verfügung.

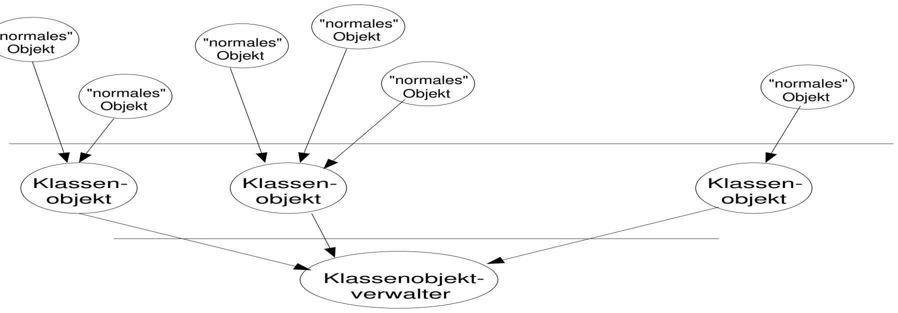


Abbildung 4: Objekthierarchie

3.3 Implementation

Die Umsetzung des in diesem Abschnitt erläuterten Konzeptes erfolgt zur Zeit auf Basis der Programmiersprache C++. Die Entscheidung für diese Programmiersprache wird dabei vor allem durch die hohe Verbreitung dieser Sprache und die Verfügbarkeit entsprechender Compiler motiviert. Zudem hat sich die Sprache bereits bei der Entwicklung von Betriebssystemen (z. B. Choices [Russo91]) und systemnaher Software bewährt. Basis der angestrebten Adaptierbarkeit ist somit der Austausch von C++-Objekten und Klassen, jeweils realisiert durch entsprechende Klassenobjekte bzw. den Klassenobjektverwalter.

Die durch eine Programmiersprache bereitgestellten Abstraktionen reichen im allgemeinen nicht aus, um die komplizierten Zusammenhänge in einem verteilten System zu beschreiben, beispielsweise unterschiedliche Arten der Interobjektkommunikation und Synchronisation paralleler Abläufe. Daraus folgten in den vergangenen Jahren eine Reihe von Entwicklungen, derartige Mechanismen in die Programmiersprache zu integrieren (z. B. C&&, ParC, ConcurrentC [Hahn94]). Im hier gewählten Ansatz soll jedoch auf Sprachmodifikationen verzichtet werden, um einerseits verfügbare Compiler auch weiterhin benutzen zu können und andererseits den Entwickler nicht mit einem weiteren C++-Dialekt zu konfrontieren. Stattdessen werden, gemäß dem zugrundeliegenden Architekturmodell, Mechanismen in Form von Diensten durch die Infrastruktur zur Verfügung gestellt.

4. Adaptionsmanagement

Durch die Bereitstellung zahlreicher Anpassungsmöglichkeiten wird ein System weniger beherrschbar. Dieser Trend ist bereits bei heutigen kommerziellen Betriebssystemen zu beobachten. Deshalb werden neue Verfahren und Werkzeuge benötigt, die den Nutzer oder Administrator bei der Optimierung und beim Ausbau des Systems unterstützen, um Fehler bei der Anpassung zu vermeiden und die Verwaltung überschaubar und komfortabel zu gestalten. Als Lösungsansatz wird der Einsatz eines Expertensystems verfolgt, das nicht nur zur Unterstützung eines Systemadministrators gedacht ist, sondern in regelmäßigen Abständen (z. B. nachts) vom System selbst angestoßen wird. Die Wissensbasis eines solchen Systems muß folgendes Wissen enthalten:

- Eigenschaften der einzelnen Objekte bzw. Komponenten des Betriebssystems: z. B. mögliche Parallelität, Echtzeit-Forderungen, Lebenszeit, benötigte Ressourcen

- Gegenseitige Abhängigkeiten von Objekten bzw. Komponenten des Betriebssystems, wie:

  - Nutzungsbeziehungen: „netscape 1.1 benötigt libc.so Version 4.5“,
  - Versionen: „Stimmt die major number überein, kann stets die Version mit einer höheren minor number statt der älteren Version benutzt werden“

- Wissen über Objekte bzw. Komponenten, mit denen das System erweitert werden kann oder die derzeit laufende Komponenten ersetzen können, z. B. andere Speichermanager, Scheduler und ihre Eigenschaften. Dies schließt Wissen über Kompatibilität und die beim Austausch notwendigen Zustandsüberführungen ein.

- Eigenschaften des Systems als Ganzes:
  z. B. Schwellwert der Systemlast, von welchem an die systeminternen Messungen und die Datenerfassung für die Selbstoptimierung einzustellen sind, die das System zu sehr belasten.

- Wissen über Basis und Umgebung des Systems:
  Hardware, auf welcher es arbeitet, oder ob es Teil eines verteilten Systems ist.

- Ziele für die Optimierung:
  Präferenzen des Nutzers bzw. Administrators, z. B. „Die Laufzeit sollte gering sein, statt Speicher zu sparen.“

Neben umfangreichem Wissen benötigt man leistungsfähige Mechanismen zur Gewinnung von aktuellen Fakten sowie zur automatischen Aktualisierung und Erweiterung der Wissensbasis. Zur Faktengewinnung ist es erforderlich, die entsprechenden Daten an den Stellen im System, an denen sie anfallen, aufzuzeichnen, ggf. zu bearbeiten und an die Faktenbasis weiterzureichen. Der Code hierfür wird im gesamten System verteilt sein und muß daher besonderen Anforderungen genügen. Beispielsweise sollten derartige Datenerfassungen abstellbar sein, ggf. der Code sogar, falls nicht mehr benötigt, aus dem System entfernt werden können. Besonders wichtig ist das bei Beobachtungen in zeitkritischen Systemteilen, z. B. bei der Messung der für die Ausführung bestimmter Codesequenzen benötigten CPU-Zeit.

In welchem Umfang die Aktualisierung und Erweiterung der Wissensbasis automatisiert werden kann, hängt stark von der Architektur des (restlichen) Betriebssystems ab. Moderne Meta-Level-Architekturen eignen sich gut für diese Automatisierung, da dort die für die Wissensbasis relevanten Informationen in genau definierten Meta-Objekten[3] zu finden ist. Deshalb kann die Wissensbasis hier zum Zeitpunkt des Ladens von neuen Objekten automatisch aktualisiert werden.

5. Verwandte Arbeiten

Das System *Apertos* gilt als erstes reflexives Betriebssystem mit Meta-Level-Architektur [Yokote89, Yokote93, Lea95]. Dynamische Anpassung wird in diesem System erreicht, indem ein Objekt von seinem nicht mehr optimalen Meta-Space zu einem kompatiblen, aber optimalen migrieren kann.

Auch der Objektserver OMOS des *Flex* (Mach4)-Projektes [Orr92] unterscheidet Objekte und zugehörige Metaobjekte, wobei diese vor allem Informationen darüber besitzen, wie das entsprechende Objekt erzeugt und wieder zerstört werden kann.

[3] z. B. im Klassenobjekt

Statische Anpassung an spezielle Anforderungen sowohl der Hardware als auch der Software sind Gegenstand des Projektes *Tigger* [Cahill94]. Im Blickwinkel dieses Projektes liegen vor allem verteilte Systeme aus spezialisierten, ggf. Ein-Applikations-Maschinen.

Der Fokus der Projekte *Spin* [Bershad95] und *Bridge* [Lucco94, Wahbe93] liegt im Kontext von Mikrokernsystemen vor allem auf der Einbringung von Code in den Kern und den damit verbundenen Schutzfragen. Darüber hinausgehend wird im *VINO*-Projekt [Seltzer] angestrebt, Applikationen wesentlich unbeschränkteren Zugriff auf im Kernel implementierte Primitive zu bieten.

Größere Übereinstimmung in den Zielvorstellungen gibt es zwischen dem vorgestellten Projekt und *Lipto* [Druschel92], wo versucht wird, die in Mikokernsystemen wie Mach [Acetta86] vorhandene zwangsläufige Übereinstimmung zwischen softwaretechnologisch motivierten Modulgrenzen und sicherheitstechnisch begründeten Schutzdomänengrenzen zu überwinden. Erreicht wird dies in Lipto durch alternativ einsetzbare Kommunikationsmechanismen zwischen den Modulen.

6. Ausblick

Im Verlauf eines Redesigns wird die zunächst auf einer Unix-Plattform erprobte Objektverwaltung in den vorhandenen CHEOPS-Prototypen eingearbeitet. Ebenso muß ein dynamischer Linker in das System integriert werden. Auf dieser Grundlage sollen dann Fragen des Hinzufügens, Austauschens und Entfernens von Objekten im laufenden System untersucht werden. Eine besonders wichtige Fragestellung beim Objektaustausch ist beispielsweise, wann er überhaupt möglich ist und wie Datenübergabe und -transformation zwischen altem und neuem Objekt realisiert und verallgemeinert werden können.

Der derzeit ebenfalls auf einer Unix-Plattform entstehende Prototyp für das die Adaption unterstützende Expertensystem wird vor allem durch die Erstellung der Wissensbasis und die Schaffung von Diagnosemodulen weiterentwickelt und später in das Experimentalsystem integriert.

Auf diese Weise soll ein flexibles, dynamisch anpaßbares Betriebssystem mit einer leistungsfähigen Verwaltung der variablen Systemstruktur entstehen.

Literatur

[**Acetta86**] Acetta, M.; Baron, R.; Bolosky, W.; Golub, D.; Rashid, R.; Tevanian, A.; Young, M.: *Mach: A New Kernel Foundation for UNIX Development.* Proc. of the Summer USENIX Conference, 1986.

[**Bershad95**] Bershad, B.; Savage, S.; Pardyak, P.; Sirer, E. G.; Fiuczynski, M.; Becker, D.; Eggers, S.; Chambers, C.: *Extensibility, Safety, and Performance in the SPIN Operating System.* University of Washington, Department of Computer Science and Engineering, Draft of March 30, 1995.

[**Broner95**] Broner, G.: *UNICOS/mk: A distributed operating system with Single System Image.* Presentation at Cray Users Group, 1995.

[**Cahill94**] Cahill, V.; Hogan, C.; Judge, A.; O'Grady, D.; Tangney, B.; Taylor, P.: *Extensible Systems – The Tigger Approach.* Proc. of the SIGOPS European Workshop 1994, or:

University of Dublin, Trinity College, Department of Computer Science, Distributed Systems Group: Technical Report TCD-CS-94.

[**Coad91**] Coad, P.; Yourdon, E.: *Object-oriented Design.* Prentice Hall, 1991.

[**Dasgupta91**] Dasgupta, P.; LeBlanc, R. J.; Ahamad, M.; Ramachandran, U.: *The Clouds Distributed Operating System.* IEEE Computer, pp. 34–44, November 1991.

[**Druschel92**] Druschel, P.; Peterson, L. L.; Hutchinson, N. C.: *Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto.* IEEE: Proc. of the 12th International Conference on Distributed Computing Systems, Yokohama, 1992.

[**Goldberg83**] Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[**Graupner94**] Graupner, S.; Kalfa, W.; Schubert, F.: *Multi-Level Architecture for object-oriented Operating Systems.* TR-94-056, ICSI, Berkeley, 1994.

[**Hahn94**] Hahn, J.: *Automatische Parallelisierung von C++-Programmen für Parallelrechner mit physikalisch verteiltem Speicher.* Diplomarbeit, Fakultät für Informatik, Technische Universität Chemnitz, 1994

[**Härtig90**] Härtig, H.; Kühnhauser, W. E.; Lux, W.; Reck, W.: *Architecture of the BirliX Operating System.* GMD, St.Augustin, March 1990.

[**Kleiman95**] Kleiman, S.; Eykholt, J.: *Interrupts as Threads.* Operating Systems Review, Vol.29, No.2, pp. 21–26, April 1995.

[**Lea95**] Lea, R.; Yokote, Y.; Itoh, J.: *Adaptive operating system design using reflection.* To be presented at HTOS 1995.

[**Lucco94**] Lucco, S.; Wahbe, R.; Graham, S. L.: *Adaptable Binary Programs.* Carnegie Mellon University, Technical Report CMU-CS-94-137, also Proc. of Winter USENIX 1994.

[**Orr92**] Orr, D.; Mecklenburg, R.: *OMOS – An Object Server for Program Execution.* Proc. of IWOOOS 1992.

[**Russo91**] Russo, V. F.: *An Object-Oriented Operating System.* PhD Thesis, University of Illinois, 1991.

[**Schubert95**] Schubert, F.: *Adaptierbarkeit in objetorientierten Betribssystemen – ein alternativer Ansatz.* Technische Universität Chemnitz, 1995
`http://www.tu-chemnitz.de/sgr/papers/papers.html`

[**Seltzer**] Seltzer, M.; Endo, Y.; Small, C.; Smith, K. A.: *An Introduction to the Architecture of the VINO Kernel.* Harvard University.
`http://www.eecs.harvard.edu/chris/papers/arch-intro.ps`

[**Wahbe93**] Wahbe, R.; Lucco, S.; Anderson, T. E.; Graham, S. L.: *Efficient Software-Based Fault Isolation.* Proc. of the 14th ACM Symposium on Operating System Principles, 1993.

[**Yokote89**] Yokote, Y.; Teraoka, F.; Tokoro, M.: *A Reflective Architecture for an Object-Oriented Distributed Operating System.* Sony CSL, Technical Report SCSL-TR-89-001, 1989.

[**Yokote93**] Yokote, Y.: *Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach.* Sony CSL, Technical Report SCSL-TR-93-014, 1993.