Skizze einer auf nur zwei Abstraktionen beruhenden Betriebssystem-Architektur: Nester und Bausteine

Thomas Schöbel-Theuer
schoebel@informatik.uni-stuttgart.de

28. Oktober 2002

Zusammenfassung

Es wird gezeigt, wie sich ein vollständiges Betriebssystem aus nur zwei Abstraktionen aufbauen lässt. Die Behandlung nicht-funktionaler Eigenschaften wird dadurch vereinfacht.

1 Einführung

Dieses Arbeitspapier ist im wesentlichen eine Zusammenfassung einiger Teile der bisher unveröffentlichten Arbeit [ST02]. Die Grundidee dieser Arbeit besteht darin, ein Betriebssystem auf allen Ebenen und Schichten aus zwei Abstraktionen aufzubauen: *Nester* und *Bausteine*.

Zum Verständnis dieser Einführung genügt es, sich eine *Nest*-Instanz als eine Art großen (virtuellen) Adressraum vorzustellen, in dem sich Daten befinden können. In diesem Adressraum können beliebige Löcher vorhanden sein. In Erweiterung zu bisher verwendeten Adressraum-Abstraktionen gibt es auf Nestern eine Verschiebe-Operation `move`, mit der ein beliebig großer Speicherblock im virtuellen Adressraum auf virtuelle Weise verschoben werden kann. Der Inhalt des Speicherblocks erscheint anschließend unter verschobenen virtuellen Adressen; die Implementierung erfolgt jedoch so, dass keine tatsächlichen Kopieroperationen im Speicher ablaufen, sondern nur der virtuelle Eindruck einer Verschiebung entsteht. Das Vorhandensein einer billigen Verschiebeoperation ist für die hier vorgestellte Architektur essentiell; zur Unterscheidung gegenüber „statischen" virtuellen Adressräumen habe ich den Begriff „Nest" gewählt; dieser Begriff assoziiert im Englischen die Möglichkeit zum Ineinanderschachteln, von der nachfolgend Gebrauch gemacht wird.

Bausteine kann man sich als „Transformatoren" vorstellen, die ein oder mehrere Eingangs-Nester in ein oder mehrere andere Ausgangs-Nester transformieren. Bausteine lassen sich instantiieren, d.h. man kann beliebig viele Inkarnationen des gleichen Baustein-Typs herstellen, deren Eingänge sich mit den Ausgängen anderer Baustein-Instanzen verbinden oder „verdrahten" lassen. Als für Menschen leicht verständliche Darstellung benutze ich Zeichnungen, wie sie in der Elektro- und Digitaltechnik für Schaltbilder verwendet werden. Eine Baustein-Instanz wird als Kästchen mit linksseitigen Eingängen und rechtsseitigen Ausgängen gezeichnet. Die „Verdrahtungsregeln" sind gleich wie bei Schaltbildern in der Elektrotechnik.

In Abbildung 1 sehen wir Ausschnitte eines Szenarios, wie es in konventionellen Betriebssystemen auftritt. Am Ende der Baustein-Hierarchie steht eine Instanz von `mmu_i386`, die einen (nicht eingezeichneten) virtuellen Benutzer-Adressraum herstellt, indem sie als „Treiber" für die MMU-Hardware (Memory Management Unit) fungiert. Der Benutzer-Adressraum enthält genau das, was sich im Eingangs-Nest dieses Bausteins „befindet" bzw. was dort vom Vorgänger-Baustein (virtuell) zur Verfügung gestellt wird. Im Beispiel ist dies ein virtuelles Prozessabbild, das von der `union`-Instanz generiert wird und als wesentliche Grundelemente jeweils ein Code-, ein Daten-, ein Stack-„Segment" sowie eine `mmap`-„Datei" zu einem ausführbaren Prozessabbild zusammenstellt. Der Begriff „Segment" wird hier jedoch nicht verwendet, da er mit der Nest-Abstraktion zusammenfällt bzw. einen Spezialfall davon darstellt.

Der `union`-Baustein wird aus mehreren Quellen gespeist. Drei davon sind Nest-Instanzen, die man konventionell als „Dateien" bezeichnen würde: eine Quelle ist eine `device_ramdisk`-Instanz, die flüchtigen Hauptspeicher bereitstellt (in der Praxis ist es sinnvoll, hierfür eine gepufferte „temporäre Datei" zu verwenden, damit bei Speichermangel ein Auslagern auf Hintergrundspeicher möglich ist). Die anderen „Datei"-Nester stammen wiederum aus Quellen, die die Bezeichnung `dir_*` tragen. Die `dir_*`-Bausteine erfüllen ungefähr den gleichen Zweck wie Verzeichnisse in konventionellen Dateisystemen, jedoch implementieren sie keine Verzeichnis-Bäume, sondern nur flache Verzeichnisse. Durch Schachtelung von `dir_*`-Instanzen lassen sich sehr leicht konventionelle Verzeichnisbaum-Hierarchien nachbilden. Man kann sich eine `dir_*`-Instanz als eine Art „Container" vorstellen, der den Platz für seine Ausgangs-Nester in seinem Eingangs-Nest allokiert und verwaltet. Dabei wird die relativ billige Verschiebeoperation `move` des Eingangs-Nestes ausgenutzt, mit deren Hilfe das *Platzmanagement* leicht lösbar wird. Verschiebe-Operationen können beispielsweise benutzt werden, wenn neue Ausgangs-Nester hinzukommen oder wenn sich die Größe eines Ausgangs-Nestes ändert.

Die billige virtuelle Verschiebeoperation wird in der `map_simple`-Instanz implementiert, die im Beispiel an der „Wurzel" der Verzeichnis-Hierarchie steht. Dort werden diejenigen Probleme gelöst, die bei konventionellen Dateisystemen als *Fragmentierungs-Probleme* bekannt sind (Lokalitätsverhalten des Zugriffs).

Die vorgeschaltete `buffer`-Instanz sorgt für die Entkop-

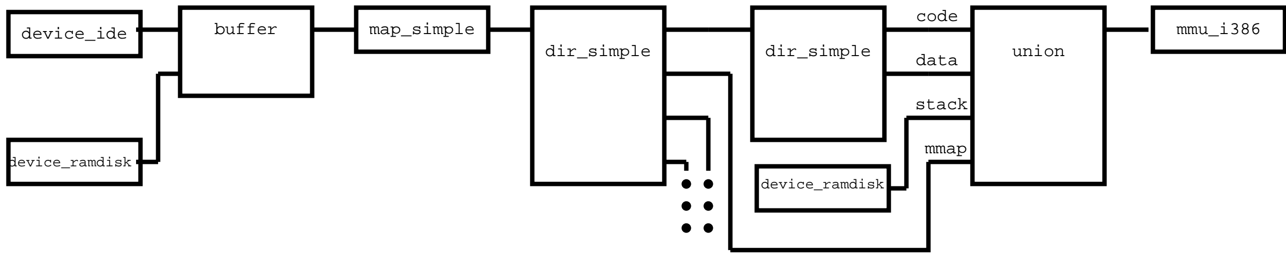


Abbildung 1: Erstes Beispiel

pelung des zeitlichen Zugriffsverhaltens auf langsame Peripheriegeräte wie z.B. Festplatten (`device_ide`), und läßt sich mit der Funktionalität konventioneller Buffer-Caches vergleichen.

Sämtliche Leitungen in der Zeichnung repräsentieren Nest-Instanzen, die jeweils immer die gleiche Schnittstelle besitzen. Die Bausteine sind daher in beinahe beliebiger Weise miteinander kombinierbar (die Frage ist nur, welche Kombinationen Sinn machen).

Das Beispiel in Abbildung 2 soll die Flexibilität dieses Systems andeutungsweise demonstrieren: im Vergleich zu Abbildung 1 kommen zwei `remote`-Instanzen hinzu, die das Client-Server-Paradigma auf Nester übertragen. Eine `remote`-Instanz macht ein Nest so auf einem anderen Rechner verfügbar, als wäre es dort lokal vorhanden. Die eine `remote`-Instanz sitzt im Beispiel am Ende der Hierarchie und macht ein komplettes Prozessabbild auf einem anderen Rechner verfügbar. Dort befindet sich eine weitere `mmu_i386`-Instanz, in der parallele Kontrollflüsse ablaufen können. Mit derartigen Konfigurationen läßt sich beispielsweise verteiltes Rechnen (number-crunching) oder Prozess-Migration betreiben. Die andere `remote`-Instanz stellt das andere Extrem von möglichen Einsatz-Szenarien dar: die bisherige „Wurzel" der Verzeichnis-Hierarchie wird auf einem anderen Rechner verfügbar gemacht; diese Funktionalität entspricht etwa derjenigen von konventionellen Netzwerk-Dateisystemen. Wie man an diesem Beispiel sieht, braucht hierfür kein neuer Baustein-Typ implementiert zu werden. Die anderen Bausteine müssen dazu die in [ST02] genauer untersuchte `multiuser`-Kompetenz besitzen. Nur wenn diese gegeben ist, dürfen Eingänge mehrerer Baustein-Instanzen parallel am gleichen Ausgang angeschlossen werden.

Weitere Beispiele, insbesondere zur nahtlosen Integration der Funktionalität von Datenbanken und von Transaktionen in Betriebssysteme, sind in [ST02] zu finden.

Der Rest dieses Papiers ist folgendermaßen aufgebaut: Abschnitt 2 behandelt die hinter dem vorliegenden Entwurf stehenden grundlegenden Prinzipien und Überlegungen. Abschnitt 3 zeigt einen Überblick über die Elementaroperationen auf Nestern. In Abschnitt 4 werden Baustein-Arten vorgestellt, die die Grundfunktionalität eines Betriebssystems abdecken. Abschnitt 5 behandelt generische Operationen und ihren Zusammenhang mit der Objektorientierung. Der Instantiierungs-Mechanismus und grundlegende Instantiierungs-Strategien werden in Abschnitt 6 vorgestellt. Schliesslich wird in Abschnitt 7 auf nicht-funktionale Eigenschaften von Betriebssystemen eingegangen, deren Realisierung und Aushandlung durch den vorliegenden Entwurf vereinfacht wird.

2 Entwurfs-Prinzipien

In diesem Abschnitt möchte ich die allgemeinen Prinzipien darstellen und begründen, auf denen die vorgestellte Betriebssystem-Architektur beruht.

2.1 Generizität: Wahl der angemessenen Abstraktionsebene

Ausgangbasis aller Überlegungen ist die *Funktionalität*, die ein Betriebssystem implementieren muss, wenn es bestimmte *Anforderungen* erfüllen soll.

Die Anforderungen werden als „freie Variable" betrachtet. Anforderungen können sich fortlaufend ändern; daher werden *Klassen von Anforderungen* auf informelle Weise betrachtet.

Gängige Methoden zur Lösung dieser Problematik werden vor allem im Software-Engineering, speziell bei den objektorientierten Entwurfs-Methoden (z.B. [Mey88]) gelehrt (Stichwort Anpassbarkeit an neue Aufgabenstellungen). Der objektorientierte Ansatz versucht dabei die *Wiederverwendbarkeit* von Software-Modulen zu verbessern, indem er die *Erweiterung* bestehender Module und Schnittstellen zum Zwecke der Anpassung an neue Aufgabenstellungen ermöglicht. In der Praxis eingesetzte objektorientierte Software mit einer längeren Entwicklungsgeschichte zeigt nicht selten eine riesige Ansammlung von Klassen mit einer ziemlich komplizierten Klassen-Hierarchie und schwer durchschaubaren Vererbungs- und Enthaltenseins-Beziehungen („is-a" und „has-a"-Beziehungen). Bei eingehender Betrachtung findet sich darin viel *Redundanz*, die (vielleicht) hätte vermieden werden können, wenn man die jetzigen Anforderungen von Anfang an gekannt hätte und bei sorgfältigem Systementwurf berücksichtigt hätte.

Die allgemeine Fragestellung lautet daher, wie man *unnötige Redundanz* in der Implementierung möglichst weitgehend vermeiden kann, selbst wenn alle Anforderungen (noch) gar nicht bekannt sind.

2.1.1 Parametrische Generizität

Ein bekanntes Mittel zur Vermeidung unnötiger Redundanzen ist der Einsatz von *Generizität*.

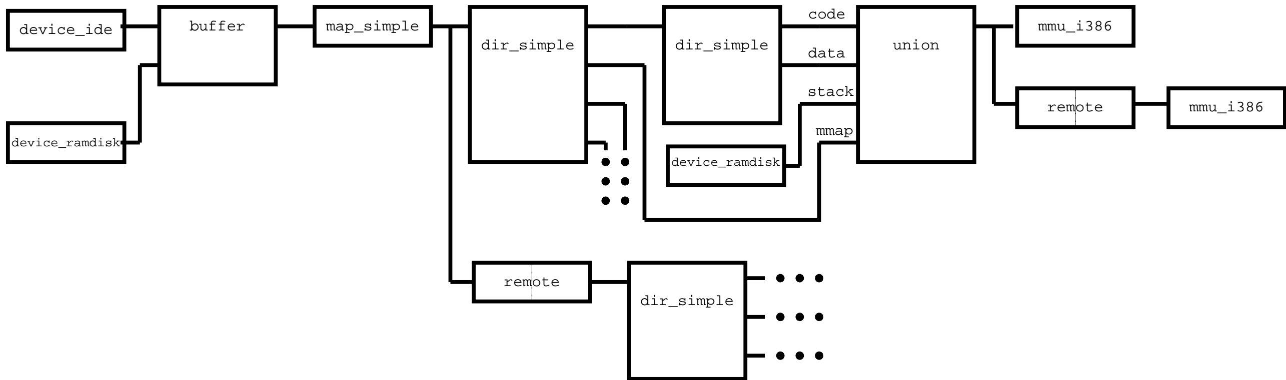


Abbildung 2: Zweites Beispiel

Als Generizität betrachte ich jedes Mittel, das es erlaubt, ähnliche Dinge nur einmal hinzuschreiben, obwohl sie mehrmals vorkommen (evtl. in Varianten).

Diese Definition ist wesentlich weit-reichender als die oftmals in der Literatur anzutreffende Verwendung des Begriffs Generizität (siehe z. B. [Mey88, Kapitel 6 und 19]). Ich verwende diesen Begriff nicht in der Reinform, sondern betrachte immer nur Spielarten von Generizität.

Eine besondere Spielart ist die *parametrische Generizität*. Beispiele hierfür sind Ersetzungmechanismen wie z.B. der Lambda-Kalkül (siehe z.B. [Fie88]) und seine Anwendung in funktionalen Sprachen wie Lisp oder ML oder Haskell (vgl. [Tho96]), Parameter-Substitution in Makro-Präprozessoren, Parameter von Template-Mechanismen in objektorienten Sprachen wie C++, oder Anwendungen des Schlüsselwortes `generic` in Ada. Parametrizität bedeutet, dass im Unterschied zu allgemeineren Ersetzungsmechanismen wie z.B. Chomsky-Typ-0-Grammatiken oder L-Systeme keine beliebigen Ersetzungen möglich sind, sondern syntaktisch eindeutig gekennzeichnete Platzhalter, die *formale Parameter* genannt werden, durch Zeichenketten oder Ausdrücke mit einer festgelegten Syntax (*aktuelle Parameter*) substituiert werden.

In der Literatur wird häufig der alleinstehende Begriff der Generizität in ungefähr der gleichen Bedeutung wie die *parametrische Polymorphie von Typen* (vgl. [CW85]) verwendet. Das Konzept der parametrischen Generizität setzt nicht das Vorhandensein von Typprüfungen voraus, jedoch sollten im Idealfall derartige Mechanismen vorhanden sein. Moderne funktionale Sprachen wie Haskell [Tho96] erlauben Typsicherheit durch automatisierte Typ-Inferenz sogar bei Typ-Parametern. Makro-Präprozessoren führen dagegen meist nur eine reine Text-Substitution durch, die zwar universell für beliebige programmiersprachliche Konstrukte anwendbar ist, aber keine separate Prä-Compilierung und keine gesonderte Typprüfung ermöglicht.

Parametrische Generizität erlaubt die Automatisierung immer wiederkehrender Programmierungs- und Formulierungs-Vorgänge. Formale Parameter können mehrmals auf verschiedene Weise durch verschiedene aktuelle Parameter substituiert werden. Dadurch ist eine kompakte Notation mit geringer Redundanz bei der Formulierung eines Algorithmus möglich. Dieser Effekt ist nicht nur bei der Programmierung im Kleinen, sondern auch bei der Programmierung im Großen nutzbar.

Im Folgenden werde ich weitere Arten von Generizität auf informelle Weise vorstellen, die sich durch parametrische Generizität darstellen und simulieren lassen (siehe [ST02]). Diese informell verwendeten Begriffe stellen kein eigenständiges theoretisches Konzept dar, sondern sind vielmehr als *Denk-Kategorien* oder besser *Denk-Hilfsmittel* für den Entwurf von komplexen Systemen durch Menschen gedacht.

Die Grundregel beim Entwurf lautet: verwende Generizität, wo immer es auf einfache Weise möglich ist und wo immer es Redundanz einsparen kann.

2.1.2 Erweiterungs-Generizität

Die in der Objektorientierung verwendete Erweiterung bzw. Spezialisierung von Schnittstellen durch *Vererbung* (gelegentlich auch die Einschränkung von Schnittstellen) läßt sich ebenfalls als eine besondere Art bzw. als Spezialfall von parametrischer Generizität charakterisieren: *Erweiterungs-Generizität*. Erweiterungs-Generizität läßt sich auch durch theoretische Begriffe wie Untertypen (subtyping, vgl. [CW85]) oder praxisnähere Ausprägungen wie abstrakte Basisklassen beschreiben.

In [ST02] wird durch ein Beispiel nachgewiesen, dass sich die Vererbung prinzipiell durch parametrische Generizität simulieren läßt[1]. Die Mächtigkeit parametrischer Generizität umfasst auch wesentliche Grundkonzepte der Objektorientierung. Daher betrachte ich Generizität schlechthin als Oberbegriff, unter den die Objektorientierung als Spezialfall untergeordnet wird.

2.1.3 Kompositorische Generizität

Eine weitere besondere Form von Generizität spielt bei uniformen IO-Schnittstellen wie [Che87], in stapelbaren

[1] Dieses Beispiel widerspricht nicht der scheinbar genau umgekehrten Aussage von Meyers Analyse in [Mey88, Abschnitt 19.3]. Meyer verwendet den Begriff der Generizität synonym zu einer Ausprägung von parametrischer Polymorphie, wie sie speziell in Ada auftritt. Im Gegensatz dazu verstehe ich unter parametrischer Generizität ein sehr viel allgemeineres Konzept, unter Generizität schlechthin sogar eine nochmals allgemeinere Denk-Kategorie.

Dateisystem-Layern wie [HP94][2] oder beim Port-Konzept von Gnu Hurd [Hur] eine Rolle, die man als Vorläufer der hier vorgestellten Architektur ansehen kann. Konzeptuell wird bei diesen Systemen die gleiche Art von Generizität wie bei der vorgestellten Architektur favorisiert, die ich *kompositorische Generizität* nennen möchte.

Bausteine implementieren Funktionen in einem weiteren Sinne, wobei die Verdrahung von Bausteinen zu komplexeren Netzwerken sich als eine Art der *Komposition von Funktionen* auffassen lässt. Kompositorische Generizität ist ebenfalls eine Variante von parametrischer Generizität, die sich allerdings von der Erweiterungs-Generizität in der Entwurfs-Philosophie (nicht notwendigerweise jedoch in den verwendeten Mechanismen) stark unterscheidet: während objektorientierte Paradigmen ebenso wie die in den 1970er Jahren populären Hierarchie-Konzepte [Dij68, Dij71, BS75, Lag75] ausdrücklich die Erweiterung von Schnittstellen bzw. von abstrakten Maschinen um neue Funktionalität durch Hinzunahme weiterer Operationen und Abstraktionen bezwecken und anstreben, wird bei kompositorischer Generizität das Gegenteil angestrebt: die Schnittstelle muss in den Grundzügen auf allen oder möglichst vielen Hierarchieebenen gleich oder zumindest weitgehend kompatibel bleiben, damit eine vielfältige Kombinierbarkeit möglich ist. Bei kompositorischer Generizität wird die Erweiterung der Funktionalität nicht durch Erweiterung der Schnittstellen, sondern durch Hinzunahme weiterer Funktionalität *innerhalb* der kombinierten Funktionen erzielt; neue Funktionalität ergibt sich mithin durch die *besondere Art der Verdrahtung zu einem Netzwerk.*

2.1.4 Universelle Generizität

Um kompositorische Generizität über das bisherige Anwendungsfeld von IO-Schnittstellen [Che87] hinaus auf das gesamte Betriebssystem ausdehnen zu können, müssen nicht nur die Urbild- und Bildbereiche der beteiligten Funktionen zueinander passen, sondern diese müssen darüber hinaus eine weitere Art von Generizität ermöglichen, die ich *universelle Generizität* nennen möchte. Beispiele für universelle Konstrukte sind Turingmaschinen oder Registermaschinen, die prinzipiell andere Turing- oder Registermaschinen *simulieren* können. Universelle Generizität ist analog dazu die Fähigkeit, alternative Funktionalitäten, Repräsentationen, Zugriffsverhalten, Granularitäten etc. *auf relativ einfache Weise* simulieren zu können; der Begriff ist daher kein absoluter Begriff, sondern gilt immer nur relativ zum Universum der jeweils simulierbaren alternativen Konzepte.

Ein Beispiel für universelle Generizität in Betriebssystemen ist die bekannte File-IO-Schnittstelle von Unix [RT74], die aufgrund ihrer Fähigkeit zu IO mit beliebiger dynamischer Blocklänge und beliebigen Zugriffsmustern ältere Datei-Konzepte wie z.B. Lochkartenstapel mit fester Satzlänge simulieren kann, darüber hinaus aber auch unterschiedliche Zugriffsmuster verschiedener Leser und Schreiber ermöglicht, die mit den vorher üblichen satzorientierten IO-Schnittstellen nicht auf einfache Weise realisierbar waren. Dieses inzwischen zur Selbstverständlichkeit gewordene historische Beispiel einer Vereinfachung durch universelle Generizität zeigt auf, dass „weniger oft mehr" ist: der Verzicht auf das Konzept eines „Datensatzes" (record) in Unix macht nicht nur die Schnittstellen und die Implementierung einfacher, sondern verbessert darüber hinaus sogar noch die Funktionalität. Universelle Generizität ist daher in hohem Maße erstrebenswert.

2.1.5 Fragen der Anwendung von Generizität

Die zentralen Fragen bei der Anwendung von kompositorischer und universeller Generizität lauten:

- Wie sollen die Bild- und Urbildbereiche aussehen?
- Welche Funktionalitäten sollen durch die Bausteine realisiert werden?

Diesen Fragen liegt das gemeinsame Problem der *Wahl der am besten geeigneten Abstraktionsebene* für die zu lösenden Aufgaben (bei teilweise unbekannten Anforderungen) zu Grunde.

Eine konkrete Wahl von Abstraktionsebenen könnte die berechtigte Kritik auf sich ziehen, daß präzise Aussagen über Vor- und Nachteile bestimmer Entwurfs-Entscheidungen nicht oder nicht ausreichend möglich sind. Im Forschungsgebiet der Betriebssysteme hat es bisher meines Wissens nach keine Evaluation verschiedener Entwurfs-Konzepte auf der Ebene ganzer Betriebssysteme gegeben, die diese Konzepte *als solche* auf einer wirklich *vergleichbaren* Grundlage (d.h. deren *isolierte* Auswirkung auf gut messbare Größen wie Performanz oder weniger gut messbare Größen wie die Änderungsfreundlichkeit) *ohne Einfluss weiterer versteckter Parameter* wie z.B. die Kenntnisse und Fähigkeiten der jeweiligen Implementierer untersucht hätten; da solche Evaluationen nach aktuellem Stand kaum möglich erscheinen, ist es stattdessen üblich, das Für und Wider verschiedener Entwurfs-Alternativen durch informelle Argumentation darzustellen.

Der Begriff der „Architektur" wird in diesem Papier in einem weiteren Sinn verwendet als vielerorts in der Literatur (vgl. z.B. [PC75, Jon80, MP81, $B^{+}95$, Lie95b, EKO95, Ass96])[3], da er weder feste Implementierungsparadigmen

[2] Die in [HP94] vorgestellte hierarchische Modularisierung von Dateisystemen verwendet einige der hier vorgestellten Konzepte, beschränkt sich jedoch nicht nur auf die Funktionalität von Dateisystemen und verwendet deren aufwendigere Schnittstellen (insbesondere zum Management von Dateisystem-Unterbäumen), sondern basiert auch inhärent auf konventionellen Dateisystem-Konzepten und -Abstraktionen wie `vnodes`, `vnode`-Operationen und `vfs`, die in der vorliegenden Arbeit keine Rolle spielen.

[3] In [Ant90] beschreibt Antonov eine „reguläre Architektur", die trotz anderer formeller Darstellung dem Kerngedanken der hier vorgestellten Architektur sehr nahe kommt, dabei jedoch hauptsächlich auf Erweiterungs-Generizität und weniger auf kombinatorische Generizität ausgelegt ist. Im Detail gibt es weitere gravierende Unterschiede, so etwa bei der enormen Anzahl von Schnittstellen-Funktionen, bei den Instantiierungs-Mechanismen, den fest eingebauten Schutzmechanismen, und möglicherweise auch bei der Anzahl der „Ausgänge"pro „Baustein" (in allen Beispielen kommt immer nur ein einziger Ausgang vor; einen grammatisch klaren expliziten Hinweis auf mehrere „Ausgänge"konnte ich nicht finden). Weiterhin fehlt bei Antonov der hier vorgestellte Ersatz von Dateisystemen durch dynamische hierarchische Instantiierung von Bausteinen; eine Unterscheidung verschiedener Betriebsarten oder Betriebsmodelle ist ebenfalls nicht zu erkennen.

wie „communicating sequential processes“ (siehe [Hoa78]) vorschreibt, noch die Frage nach Kern-Größen und -Umfängen (vgl. [Lie95b]) in festliegender Weise beantwortet, noch die Frage nach den Grenzen zwischen Benutzer-Adressräumen und Kern- oder Server-Adressräumen festlegt, sondern diese Fragen je nach Systemkonfiguration anders lösen kann, ohne den Quelltext des Systems ändern zu müssen (eine Ausnahme ist [Str78], wo Stroustrup die Unabhängigkeit und Austauschbarkeit der Implementierungsparadigmen von den Modulschnittstellen aufzeigt). Einige konventionelle Architekturfragen, die bisher als „festverdrahtet“ betrachtet worden sind und deren Diskussion in der Literatur große Beachtung gefunden hat, werden hier zu *dynamischen Parametern*, deren Variation einen direkten Vergleich dieser Paradigmen bei einer Evaluation erleichtert.

Zur Begriffsbildung: statt mit Architektur könnte man die vorliegende Entwurfs-Methodik auch als *Framework* oder als *Komponenten-Architektur* bezeichnen. Da diese Begriffe nach weit verbreiteter Auffassung zur Zeit sehr eng mit der Objektorientierung verknüpft sind, die ich im vorliegenden Papier zwar *ermögliche*, aber nicht zur *zwingenden Voraussetzung* der Architektur erhebe, habe ich von einer derartigen Begriffsbildung vorerst Abstand genommen. Im Unterschied zu Szyperskis Komponenten-Begriff [Szy98] könnte man die hier vorgeschlagenen Bausteine auch als *leichtgewichtige instantiierbare Komponenten* charakterisieren.

Bei Konflikten zwischen verschiedenen Arten von Generizität bei zu treffenden Entwurfs-Entscheidungen propagiere ich zumindest für die Anwendung in Betriebssystemen die folgende Priorisierung miteinander konkurrierender Arten von Generizität:

1. Universelle Generizität
2. Kompositorische Generizität
3. Erweiterungs-Generizität

Eine genaue Evaluation der Vor- und Nachteile dieser Priorisierung für die Anwendung in Betriebssystemen steht noch aus. Da die Erweiterungs-Generzität erst an dritter Stelle steht und beim hier skizzierten Entwurf gar nicht verwendet wird (obwohl sie zugelassen ist), braucht man die hier vorgeschlagene Architektur nicht unbedingt mit dem Etikett „objektorientiert“ zu versehen.

2.2 Trennung von Mechanismus / Strategie / Repräsentation

Im Betriebssystem-Bau ist das Prinzip der Trennung von *Mechanismus* (mechanism) und *Strategie* (policy) schon sehr lange bekannt und benutzt worden; eine Darstellung dieses Prinzips und ihrer Vorteile findet man z.B. in [LCC+75, CJ75].

Als unabhängig von Mechanismen und Strategien ist fernerhin die *Repräsentation* zu betrachten (vgl. [Par78]). Beispielsweise lässt sich ein Monitor [Han73, Hoa74] als programmiersprachliche Repräsentation eines Mechanismus verstehen, der prinzipiell wie ein binäres Semaphor funktioniert. Hieraus ist zu erkennen, dass die Auswahl einer einzigen Repräsentation ausreichend ist und Redundanz vermeiden hilft; diese Repräsentation sollte jedoch möglichst universell und einfach handhabbar / verstehbar sein.

Die Abgrenzung zwischen Mechanismus und Strategie ist nicht immer eindeutig möglich, da sie vom Kontext und der gewählten Abstraktionsebene abhängt. Eine Strategie kann beispielsweise ihrerseits wieder in Unter-Strategien und Unter-Mechanismen zerfallen. Auch Mechanismen können in ihrer Feinstruktur wiederum Strategien enthalten.

Auf relativ grober Abstraktionsebene korrespondieren Mechanismen und Strategien mit den hier propagierten Abstraktionen Nest und Baustein auf direkte Weise: ein Nest stellt einen Satz von abstrakten Mechanismen zur Verfügung, ein Baustein implementiert eine Strategie oder eine Menge von Strategien.

Die Nest-Schnittstelle stellt insofern *abstrakte* Mechanismen bereit, als dass die konkrete Implementierung der Mechanismen durch späte Bindung (late binding) zu Stande kommen kann.

Die Trennung zwischen Mechanismus und Strategie wird durch die hier vorgestellte Architektur nicht nur direkt unterstützt, sondern geradezu zum Prinzip erhoben. Da Bausteine wiederum andere Bausteine enthalten oder über ihre Eingänge benutzen können, lässt sich das Prinzip der *schrittweisen Verfeinerung* auf die Trennung in Mechanismen und Strategien anwenden: eine Baustein-Hierarchie fungiert gleichzeitig als hierarchisch-rekursive Zerlegung des zu lösenden Problems in Mechanismen und Strategien.

2.3 Das Verantwortungs-Prinzip

Verantwortung und ihre Aufteilung ist ein Prinzip, das in komplexeren menschlichen Gesellschaften und Systemen wie Firmen oder Behörden seit Jahrtausenden mit Erfolg praktiziert wird; als Ergebnis eines Aufteilungsvorgangs von Verantwortung entstehen *Zuständigkeiten*. Im Kontext von Betriebssystemen wurde Verantwortung ebenfalls als Strukturierungs-Leitlinie eingesetzt (ohne dass dies den jeweiligen Autoren völlig bewusst gewesen sein muss): die in den 1970er Jahren populären hierarchischen Strukturen [Dij68, Dij71] lassen sich beispielsweise durch Aufteilung von Verantwortung gewinnen, ebenso die Weiterführung dieser Idee in sogenannten „objektorientierten“ Betriebssystemen (beispielsweise [K+81, Y+90, B+95, Ass96] u.v.m.), aber auch die auf bestimmten festen Mechanismen aufgebauten Betriebssysteme (z.B. auf Capabilities basierend [NW77, T+90]). Auch das im Software-Engineering oft zitierte Lokalitäts-Prinzip beruht letztlich auf der Aufteilung von Verantwortung. Es gibt unzählige weitere Beispiele.

2.3.1 Kontrollflüsse

Ein *sequentieller Kontrollfluss* (thread, konzeptuelle Beschreibung bereits in [DH66]) lässt sich als Aktivitätsträger charakterisieren, der von außen beobachtbar nur eine

sequentielle Folge von (Maschinen-)Operationen ausführt. Kontrollflüsse sind bereits in [Dij68] zur Strukturierung von Verantwortung in Betriebssystemen benutzt worden. Kontrollflüsse *können* in der hier vorgestellten Architektur zwar zur Strukturierung von Verantwortung benutzt werden, jedoch gibt es keine *notwendigerweise feste Verbindung* mit Verantwortungsbereichen.

Kontrollflüsse werden als vollkommen unabhängig von konventionellen Konzepten wie *Prozesse* oder Mechanismen wie *Schutzbereiche* (spheres of protection [DH66], protection domains [NW74, CJ75], siehe auch [BPS81] und [C⁺94, Ros94]) angesehen. Ich verwende den Begriff des Prozesses nicht und beschränke mich auf die Begriffe Kontrollfluss und Schutzbereich.

Jeder Kontrollfluss hat zu jedem Zeitpunkt einen eindeutig bestimmten *Kontext*. Die Bedeutung dieses Kontexts hängt vom jeweils gewählten *Ausführungs-Modell* ab: bei der Simulation eines „Prozesses“ wäre dies beispielsweise das Prozessabbild, in dem sich der Kontrollfluss befindet und seine Operationen ausführt; im Fall von Single-Space- oder Hybridsystemen wäre dies der jeweils aktuelle Schutzbereich.

Kontexte können auf Anforderung *gewechselt* werden (vgl. [DH66]): ein Kontrollfluss wechselt damit in einen anderen „Prozess“ bzw. in einen anderen Schutzbereich, der sich ggf. auch auf einem anderen Rechner befinden kann. Hiervon sind zwei verschiedene Varianten denkbar: eine ohne Abspeichern des alten Kontextes im Stile von Wechseln zwischen Coroutinen, die andere mit Abspeichern aller durchlaufenen Kontexte in Form eines Stapelspeichers mit der Möglichkeit zur Rückkehr in einen früheren Kontext ähnlich einem synchronen RPC oder LRPC. Eine Konsequenz aus der Wechselmöglichkeit ist, dass die Anzahl der in einem „Prozess“ oder Schutzbereich tätigen Kontrollflüsse sich dynamisch ändern und auch zeitweise zu Null werden kann.

Man sollte sich vor Augen halten, dass Schutzbereiche einen Mechanismus zur Durchführung und Überwachung von Sicherheits-Konzepten darstellen, wohingegen Verantwortungsbereiche ein davon prinzipiell unabhängiges *logisches Konzept* zur Strukturierung eines Systems darstellen. Die Abbildung von Verantwortungsbereichen auf Schutzbereiche ist eine Frage von Strategien und kann auf unterschiedliche Weise, ggf. auch dynamisch zur Laufzeit, gelöst werden.

Kontrollflüsse werden in konventionellen Architekturen als eigenständige Abstraktion betrachtet. Da ihre Implementierung in einem eigenen Baustein erfolgen kann, ist die Behandlung als separate Abstraktion nicht unbedingt notwendig. Wegen ihrer dynamischen Durchdringung der gesamten Infrastruktur kann man sie jedoch als *Hilfsabstraktion* bezeichnen.

Im Folgenden betrachten wir unser Betriebssystem zur Vereinfachung standardmäßig weder mit Hilfe von Prozess- oder Schutzbereichen (vgl. [HR73]), noch von Kontrollflüssen oder anderen Aktivitätsträgern, sondern ausschließlich auf Basis der jeweils zu implementierenden und zu verantwortenden Funktionalität.

2.3.2 Schnittstellen-Mechanismen

Die Gesamtverantwortung eines Betriebssystems kann auf sehr viele verschiedene Arten aufgeteilt werden; die von mir bevorzugten Aufteilungs-Strategien werden in [ST02] näher dargestellt. Jegliche Art von Aufteilung von Verantwortung führt dazu, dass *Modularisierungs-Grenzen* in ein System eingeführt werden, die bei formalisierter Darstellung auch als *Schnittstellen-Instanzen* bezeichnet werden.

Durch die Einführung von Schnittstellen-Instanzen zur Begrenzung von Modul-Instanzen entsteht eine Unterscheidung der durchzuführenden *Operationen* in *Aufrufer*- und *Bearbeiter*-Instanzen, die jeweils immer nur relativ zu einer festen Schnittstelle so bezeichnet werden; dieselbe Instanz kann bezüglich verschiedener Schnittstellen gleichzeitig als Aufrufer oder Bearbeiter fungieren. Bei verschiedenen unterscheidbaren Operationen können diese Rollen unterschiedlich verteilt sein, so kann etwa eine Modul-Instanz A gegenüber der Instanz B bei der Durchführung der Operation op_1 als Aufrufer, bei op_2 als Bearbeiter fungieren.

Als Mechanismen zur Weitergabe der Verantwortung einer Aufrufer- an eine Bearbeiter-Instanz kommen mehrere bekannte und bewährte Methoden in Betracht, beispielsweise in der Reihenfolge fallenden Overheads bzw. fallender Kosten:

- RPC (remote procedure call) [BN84, TA90]
- LRPC (local / lightweight RPC) [B⁺90]
- Indirekte Prozeduraufrufe
- Direkte Prozeduraufrufe
- Makro- oder Inline-Prozeduraufrufe

Diese Beispiel-Mechanismen eignen sich teilweise nur für die *synchrone* Weitergabe von Verantwortung, bei der die Aufrufer-Instanz bis zur Beendigung der Bearbeitung *warten* muss. Bei *asynchroner* Weitergabe der Verantwortung entsteht implizit ein weiterer *logischer Kontrollfluss*[4]. Die asynchrone Weitergabe von Verantwortung per Prozeduraufruf ist ebenfalls möglich, dazu ist jedoch das jeweils verwendete Kontrollfluss-Konzept einzubeziehen.

Man sollte sich vor Augen halten, dass die Frage nach einer parallel oder sequentiell übertragenen Verantwortung prinzipiell von der Aufteilungsstrategie in Module unabhängig ist und daher als eine Frage der konkreten Implementierungsstrategie der Module gesehen werden kann, auch wenn die zugehörigen Mechanismen nicht vollkommen (wenn auch weitgehend) voneinander unabhängig sind.

Universelle Generizität relativ zu den oben genannten synchronen Mechanismen läßt sich direkt durch einheitliche Verwendung von synchronem RPC auf allen Ebenen

[4] Es muß nicht unbedingt ein tatsächlich neuer Kontrollfluss im physischen Sinne entstehen: Bei der Implementierung von Bausteinen als Server-Prozesse (vgl. [Han70, Hoa78]) bleibt die Anzahl physischer Kontrollflüsse im Regelfall konstant. Die im Nachrichtensystem gepufferte Nachricht stellt im logischen Sinne einen logischen Kontrollfluss dar, der von einem physischen Kontrollfluss simuliert wird, sobald die Nachricht bearbeitet wird.

des Systems erzielen, da dieser die Semantik der anderen Mechanismen als Spezialfall mit umfasst. Dies würde jedoch die Performanz eines lokalen Rechners in unakzeptabler Weise verschlechtern. Zur Erzielung universeller Generizität schlage ich daher eine Erweiterung der bereits in [Str78] dargestellten Methode vor:

Man wähle eine einheitliche und bequeme syntaktische Repräsentation, beispielsweise die Prozeduraufruf-Syntax der verwendeten Programmiersprache. Die *Bindung* von Instanzen dieser Konstrukte an einen der obigen Mechanismen erfolgt dann entweder zur Übersetzungszeit des jeweiligen Moduls, oder falls möglich auch zur Laufzeit bei der Instantiierung eines Moduls.

2.3.3 Abgrenzung von Verantwortung durch Kapselung

Im Software-Engineering ist lange bekannt, dass die Prüfbarkeit, Wartbarkeit und viele andere Eigenschaften von Modulen verbessert werden, wenn die Schnittstellen möglichst „dünn" sind und möglichst wenig Information über die innere Struktur der Implementierung preisgeben (Prinzip der *Verbergung von Information*, vgl. [Par72]).

Dies bedeutet zum einen, dass Schnittstellen stets offengelegt werden müssen, wobei die Schnittstelle zwischen Bausteinen eine von der Architektur vorgegebene Form haben muss, an die sich die Implementierer von Bausteinen halten müssen. Alles andere ist als *lokale Variable* eines Bausteins zu betrachten. Die interne Realisierung von Bausteinen darf jedoch (bzw soll möglichst) andere Baustein-Instanzen als lokale Variablen benutzen, so dass sich insgesamt eine baumartige *Lokalitäts-Hierarchie* ergibt, die von der hierarchischen Struktur der äußerlichen Baustein-Verdrahtungen unabhängig ist. Die Verantwortung für den Einsatz und den Betrieb von lokalen Baustein-Instanzen liegt beim Implementierer eines Bausteins. Die Tatsache der Benutzung von lokalen Baustein-Instanzen ist dabei von außen nicht sichtbar.

2.4 Das Problem der „Eierlegenden Wollmilchsau"

Größere Softwaresysteme, die über längere Zeit benutzt und erweitert wurden oder die für sehr große Anwendungsfelder und -bandbreiten ausgelegt wurden, zeigen oft das salopp als „Featuritis" bezeichnete Phänomen auf. Alleskönner haben manchmal Schwierigkeiten, ganz einfache grundlegende Aufgaben auf einfache und effiziente Weise zu erledigen. Die Softwarestruktur ist im Vergleich zu einfachen Programmen oder im Vergleich zu Spezialisten meist deutlich aufgebläht (Overhead) und schwerer zu durchschauen.

Um dieses Problem handhabbar zu machen, schlage ich die Verwendung unterschiedlicher *Modelle* vor, zwischen denen möglichst eine hierarchische Inklusions-Beziehung in dem Sinne herrschen sollte, daß man ein Modell als Spezialfall eines anderen Modells betrachten kann; oft ist das weniger mächtige Modell auch einfacher. Ein Beispiel hierfür sind die Zugriffs-Modelle `singleuser` und `multiuser`. Letzteres Modell muss von einem Ausgang eines Bausteins implementiert werden, damit mehrere Konsumenten parallel an ihn angeschlossen werden dürfen, ohne sich gegenseitig zu stören (wechselseitiger Ausschluss durch Locking).

Eine Nest- oder Baustein-Implementierung braucht nicht unbedingt alle möglichen Modelle zu unterstützen. So kann man beispielsweise mit der Implementierung der einfacheren Modelle beginnen und diese erst bei Bedarf auf die komplizierteren Modelle erweitern oder für die kompizierteren Modelle alternative Implementierungen vornehmen (deren höheren Kosten müssen dann von den Verwendern der einfacheren Modell-Variante nicht bezahlt werden). Welche Modelle von einer Nest- oder Baustein-Implementierung im Einzelfall unterstützt werden, wird als *Kompetenz* (*competence*) dieser Implementierungs-Instanz bezeichnet. Demgegenüber steht das *Verhalten* (*habit*) einer konkreten Aufrufer-Instanz; damit werden die Anforderungen an die Kompetenzen der Implementierungs-Instanz bezeichnet.

Die Kompetenzen einer Implementierungs-Instanz müssen zum Verhalten einer Aufrufer-Instanz *kompatibel* sein. Die Kompetenzen und das Verhalten von Nest- und Baustein-Instanzen werden je Baustein-Art in Form von *Attributen* angegeben. Die Kompatibilität kann dann bei der Verdrahtung konkreter Baustein-Instanzen automatisch getestet werden, wobei inkompatible Verdrahtungen zurückgewiesen werden.

Attribute können entweder *statisch* oder *dynamisch* sein: statische Attribute eines Baustein-Typs ändern sich nie (sie hängen nur vom Baustein-Typ oder der gewählten Implementierung ab), dynamische Attribute hängen von der konkreten Instantiierung und/oder von der Verdrahtung mit anderen Bausteinen ab, ändern sich jedoch nicht während der Lebensdauer einer Baustein-Instanz. Werte, die sich während der Lebensdauer einer Baustein-Instanz ändern können, stellen keine Attribute dar, sondern gehören zum *Zustand* der Baustein-Instanz.

Attribute werden insbesondere zur Unterscheidung verschiedener Modelle eingesetzt und in `Schreibmaschinenschrift` gesetzt.

2.5 Automatismen

Die Grundidee der Automatisierung wird seit Jahrhunderten erfolgreich zur Reduktion von Aufwand und Kosten eingesetzt.

Automatisierung bedeutet hier, dass vorzugsweise von deskriptiven Methoden Gebrauch gemacht wird, um einen Mechanismus selbsttätig auszulösen, der immer wiederkehrende Vorgänge selbsttätig bearbeitet.

Die Implementierung vieler Automatismen gehört zu den konkreten Strategien, die in speziellen Bausteinen (`strategy_*`) lokalisiert werden sollten (vgl. Abschnitt 6.2).

Zur Erstellung von Bausteinen sind weitere Automatismen von großem Nutzen, die über den Funktionsumfang üblicher Werkzeuge wie Compiler oder Debugger hinaus gehen sollten. Viele mit den hier vorgestellen (Schnittstellen-)Mechanismen zusammenhängende Konstruktionsvorgänge lassen sich automatisieren.

Die in Abschnitt 2.3.2 vorgestelle späte Bindung an konkrete Aufrufmechanismen kann beispielsweise auf folgende Weise automatisiert werden:

Der Programmierer gibt ein statisches Baustein-Attribut an, mit dem er sein *Denk-Modell* deklariert, mit dessen Hilfe er den Programmcode entwickelt hat. Das Attribut kann folgende Werte annehmen:

`code_nolock` Der Programmierer tut so, als gäbe es nur einen einzigen Kontrollfluss, der den Baustein jemals betreten dürfte; er kümmert sich also überhaupt nicht um Parallelität. Dies hat zur Folge, dass aus Sicherheitsgründen niemals ein weiterer Kontrollfluss den Baustein betreten darf, selbst wenn der bereits eingetretene Kontrollfluss eine lange dauernde blockierende Operation aufruft.

`code_monitor` Wie vorher, nur ist sich der Programmierer immerhin der Tatsache bewusst, dass mehrere Kontrollflüsse vorkommen können. Jeder Aufruf der Blockierungs-Operation `wait` (siehe Abschnitt 3) führt automatisch zu einer Entblockierung des Zugriffsschutzes gegenüber anderen Kontrollflüssen. Der Programmierer ist für die Beachtung der damit verbundenen Effekte verantwortlich; insbesondere ist ihm bewusst, dass kritische Abschnitte genau an der Stelle einer `wait`-Operation aufgehoben werden (analog zum Monitor-Konzept).

`code_reentrant` Der Programmierer ist sich dessen bewusst, dass mehrere Kontrollflüsse den Baustein asynchron betreten können. Er ist selbst für die Sicherung kritischer Abschnitte und für das Setzen von Locks verantwortlich.

Ein zu syntaktischer Analyse fähiger und die Schnittstellen-Konventionen kennender Quelltext-Präprozessor extrahiert diesen Attribut-Wert aus dem Quelltext und generiert bei Bedarf (je nach eingestelltem Aufruf-Mechanismus) automatisch Lock-Operationen zur Klammerung kritischer Abschnitte, Fallunterscheidungs-Kontrollstrukturen zum Demultiplexen eingehender (L)RPC-Aufrufe, und so weiter. Auf diese Weise läßt sich jedes Programmiermodell mit jedem Schnittstellen-Mechanismus kombinieren; bei Verwendung von statischen oder Inline-Prozeduraufrufen werden durch den Präprozessor mehrere Quelltexte verschiedener Bausteine zu einem einzigen Kombi-Baustein fest zusammen geschweisst. Bei sorgfältiger Konstruktion und Verwendung gut optimierender Compiler läßt sich durch Inline-Prozeduraufrufe jeglicher Schnittstellen-Aufwand fast vollständig eliminieren; dies ist insbesondere zum Anschluss von Prüf- und Sicherheits-Bausteinen oder kleineren Trivial-Bausteinen nützlich.

2.6 Zugriffsrechte und Schutzmechanismen

Die Rechteverwaltung in Betriebssystemen basiert bei den meisten praktisch eingesetzten Modellen [Lan81] auf *Rechte-Relationen*, die *Subjekte* und *Objekte* miteinander in Beziehung setzen; typischerweise lassen sich derartige Relationen zwischen Subjekten und Objekten tabellarisch darstellen.

Ich möchte für die Rechteverwaltung einen Ansatz vorstellen, der sich bei geeigneter Interpretation ebenfalls als Subjekt-Objekt-Schema auffassen läßt, dabei jedoch beliebige Dinge als Subjekte bzw Objekte betrachten kann.

Der *Zweck* eines Betriebssystems ist die *Ausführung von Operationen*; dies geschieht in der hier vorgestellten Architektur (vgl. Abschnitt 5) durch *Bearbeiter-Instanzen* im Auftrag von *Aufrufer-Instanzen*. Da eine Aufrufer-Instanz wiederum im Auftrag mehrerer anderer Aufrufer-Instanzen handeln kann, ergeben sich zwei Fragestellungen:

1. Wer soll natürlicherweise das Subjekt darstellen, das einen bestimmten Auftrag veranlasst?

2. Wer soll natürlicherweise das Objekt darstellen, das den Auftrag ausführen soll?

Die erste Frage ist auf verschiedene Weisen beantwortbar. Deshalb schlage ich die Einführung eines anderen Begriffes statt „Subjekt“ vor, nämlich das *Mandat* (mandate). Operationen geschehen grundsätzlich aufgrund eines Mandates; im allgemeinen kann dabei eine Instanz (bzw ein Subjekt in bisheriger Terminologie) mehrere verschiedene Mandate *wahrnehmen*; auch kann es vorkommen, dass verschiedene Instanzen aufgrund des gleichen Mandats handeln. Beispiele hierfür finden sich in der realen Welt im Rechtswesen: ein und derselbe Rechtsanwalt kann gleichzeitig verschiedene Mandate für verschiedene Mandanten wahrnehmen. Ein Mandant kann verschiedene Mandate gleichzeitig an verschiedene Rechtsanwälte oder an den gleichen Rechtsanwalt vergeben. Er kann dasselbe Mandat aber auch an mehrere Rechtsanwälte gleichzeitig vergeben, beispielsweise bei Anwalts-Gemeinschaften oder Kanzleien. Mandate können *vertreten* oder *weitergereicht* werden. Im Rechtswesen können auch *Untermandate* vergeben und aufgeteilt werden; von der letzteren Möglichkeit habe ich hier wegen der damit verbundenen Komplexität und Folgeeffekte vorläufig Abstand genommen. Die Menge aller möglichen Mandate sollte durch einen abstrakten Datentyp mit ausreichend grossem Wertevorrat[5] dargestellt werden. Mandate haben keine festliegende Interpretation, sondern werden lediglich zwischen Baustein-Instanzen weitergereicht. Alle Operationen auf Mandaten wie z.B. ihre Erzeugung gehören somit zu den Strategien, die der Implementierer eines Bausteins selbstständig bestimmen kann. Mandate stellen somit eine weitere Hilfsabstraktion dar.

Die zweite Frage ist im Kontext der hier vorgestellten Architektur relativ schnell zu beantworten: ein Objekt ist auf jeden Fall die Ausgangs-Nest-Instanz, an die der zu bearbeitende Auftrag gerichtet wird.

Zu schützen ist die *Ausführung* von Operationen. Die Menge aller ausführbaren Operationen wird *Operationenmenge* genannt; dies sind alle möglichen Kombinationen

[5] Nach heutigen Maßstäben sind hierzu mindestens 64 Bit reservierter Platz erforderlich.

von Operations-Bezeichnern mit ihren Eingabe-Parameter-Werten.

Eine *Schutzmenge* ist eine Teilmenge der Operationenmenge eines gegebenen Systems. Schutzmengen sind als Mengen aller (im Sinne irgendeines ausserhalb definierten Zulässigkeits-Begriffes) zulässigen Operationen zu interpretieren.

Der Begriff der Schutzmenge vereinfacht die Abstraktion irgendwelcher Schutz- und Zulässigkeitsbegriffe auf mathematisch sehr einfach fassbare und hochgradig flexible Weise. Da Schutzmengen i.a. zu groß für eine tabellarische Darstellung sind, braucht man Spezifikationsmechanismen, die sie beschreiben. Als Spezifikationsmechanismen kommen sehr viele konkrete Realisierungen (z.B. passend eingeschränkte Kalküle) in Betracht, deren Diskussion den Rahmen dieses Papieres sprengen würde. Es sind ohne weiteres auch militärische Schutzmodelle (vgl. [Lan81]) oder mehrere voneinander unabhängige Schutzmodelle gleichzeitig einsetzbar.

Die Prüfung von Schutzrechten kann in speziellen Prüf-Bausteinen `check_*` erfolgen, die sich prinzipiell an beliebigen Stellen einer Baustein-Hierarchie einfügen lassen. Dort weisen sie die im Sinne des jeweils implementierten Schutzmodells unzulässigen Operationen zurück. Falls man in speziellen Anwendungsbereichen wie z.B. Echtzeit-Steuerungen kein Schutzmodell benötigt, kann man auf die Instantiierung von `check_*`-Bausteinen vollkommen verzichten und damit jeglichen Overhead einsparen.

Die konkrete Realisierung eines Schutzmechanismus ist eine Frage der Strategie, an welchen Stellen welche Arten von Prüfungen auf Basis welcher Mandate durchgeführt werden sollen. Zu beachten ist, dass zur Laufzeit beliebige Baustein-Instanzen neu instantiiert und neue Verdrahtungen erzeugt werden können, mit denen ein festes Schutzkonzept u.U. umgangen werden kann. Beim Einsatz kompositorischer Generizität steckt ein Teil der im System vorhandenen Informationen in der Verdrahtung der Bausteine, die hochgradig flexibel und dynamisch änderbar ist. Zur Lösung dieses Problems müssen die Instantiierungs-Operationen von `control`-Instanzen (siehe Abschnitt 6.1) in das jeweilige Schutzkonzept mit einbezogen und überwacht werden.

Die Realisierung eines Schutzmodells ist von der Einteilung des Betriebssystems in Schutzbereiche abhängig. Im einen Extremfall kann das gesamte System in einem einzigen Schutzbereich ablaufen, im anderen Extremfall kann jede Baustein-Instanz in einem eigenen Schutzbereich ablaufen; es sind beliebige Zwischenstufen möglich. Sinnvollerweise sollten Zugriffsrechte vor allem an den Grenzen zwischen Schutzbereichen geprüft werden, sofern dieser Aufwand in Kauf genommen werden soll.

3 Nester

Die Abstraktion des *Nestes* stellt ein konkretes *Speichermodell* dar. Betriebssystem-Konstrukteure sind traditionell gewohnt, dass ihnen die Hardware eines Rechners oder eines Rechner-Typs ein bestimmtes Speichermodell vorschreibt oder zumindest in relativ engen Grenzen sehr nahe legt. Diesem Zwang versuche ich an einigen Stellen so zu entkommen, dass dadurch keine grundlegende Verschlechterung der Performanz-Eigenschaften der Hardware ausgelöst wird.

Eine *Nest-Instanz* ist ein *logischer Adressraum*, der von der Adresse 0 bis zu einer Maximalgröße (mit mindestens 64Bit) laufen kann. Mit logischen Adressen darf Adress-Arithmetik getrieben werden; es werden *Bytes* adressiert.

Ein Nest dient vor allem zur *Abbildung* von *logischen* Byte-Adressen auf *physische* Byte-Adressen von *Datenblöcken*; der Zugriff geschieht stets in Vielfachen eines dynamischen Attributs `transfer_size`. Damit soll u.a. das bekannte Konzept eines virtuellen Benutzer-Adressraums nachgebildet werden, jedoch unabhängig davon, ob MMU-Hardware vorhanden ist oder nicht. Ein physischer Datenblock kann innerhalb seiner begrenzten Länge durch Maschinenbefehle des Prozessors angesprochen werden; es darf Adress-Arithmetik auf Basis von physischen Byte-Adressen getrieben werden.

Es wird zwischen *statischem* und *dynamischem* Nest unterschieden: während ein statisches Nest sich ähnlich wie ein Unix-Device verhält und auch nur im Zusammenhang mit Peripheriegeräten o.ä. vorkommen sollte, basiert das gesamte restliche System auf dynamischen Nestern.

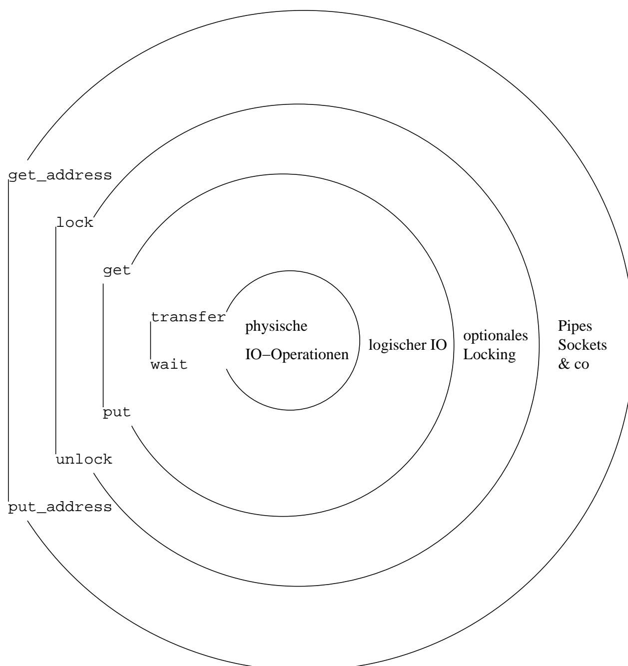
Ein dynamisches Nest kann im Unterschied zu einer konventionellen Datei abfragbare und veränderbare Löcher im Definitionsbereich der partiellen Abbildung von logischen Adressen auf physische Adressen von Datenblöcken enthalten, ähnlich wie ein Sparse File unter Unix (vgl. auch [Fot61, Lie95a]). Diese Eigenschaft wird vom gesamten Betriebssystem auf allen Ebenen außer Low-Level-Gerätetreibern intensiv genutzt. Als neuartige Elementaroperation kommt die *transparente Verschiebung* von logischen Adressbereichen hinzu. Eine Verschiebung bewirkt, dass ein Teil des Definitionsbereiches der partiellen Abbildung von logischen Adressen auf physische Datenblöcke so verschoben wird, dass dieselben Datenblöcke anschließend unter einem Adress-Offset (im Vergleich zu den vorigen Adressen) im logischen Adressbereich erreichbar sind. Im Allgemeinen können durch Verschiebe-Operationen Löcher im Definitionsbereich entstehen und/oder geschlossen werden, eventuell können am Ziel der Verschiebung auch Datenblöcke „verloren gehen“, d.h. sie werden nicht mehr vom Nest adressiert. In konventioneller Terminologie bedeutet eine Verschiebeoperation, dass je nach Vorzeichen des Verschiebe-Offsets und Länge des verschobenen Bereichs eine Insert- oder Delete- oder Move-Operation in einer Datei, in einer Datenbank, oder allgemein gesprochen in einem Nest durchgeführt wird.

Wichtig an der Verschiebe-Operation ist, dass sie *transparent* erfolgt. Damit ist gemeint, dass nur diejenigen Teile des Gesamtsystems von einer Verschiebung Kenntnis erhalten, die unbedingt davon betroffen sind; für die anderen Teile ergeben sich durch die Verschiebung keine Änderungen. Hierzu ein Beispiel: aus dem oberen Ende eines Original-Nestes sei ein Teil-Nest ausgeschnitten worden. Die Ausschneide-Operation bewirkt, dass die zugrunde liegenden Datenblöcke sowohl im Original-Nest als auch im ausgeschnittenen Teil-Nest erscheinen, dort jedoch unter

neuen Adressen (im Regelfall bei 0 beginnend). Nun finde eine Verschiebung im Original-Nest statt, die das gesamte ausgeschnittene Teil-Nest mit umfasse. Transparenz bedeutet in diesem Beispiel, dass sich an den Adressen des ausgeschnittenen Teil-Nestes nichts ändert. Benutzer des Teil-Nestes merken gar nichts davon, dass ihr Gastgeber-Adressraum „hinter ihrem Rücken“ verschoben wurde.

3.1 Elementaroperationen auf statischen Nestern

Ein konkreter Beispiel-Satz von Elementaroperationen auf Nestern wird in [ST02] ausführlich dargestellt. Eine Übersicht lässt sich dem folgenden Bild entnehmen:



Wie zu sehen ist, werden die Elementaroperationen immer paarweise mit einer entgegengesetzten oder aufhebenden Operation kombiniert. Verschiedene Arten von IO oder sonstige Zugriffs-Arten auf Nester ergeben sich durch ein Schichtenmodell, das hier kurz von innen nach außen skizziert werden soll:

Im einfachsten Fall wird physischer IO betrieben, wie er insbesondere bei Gerätetreibern vorkommt. Bei phyischem IO übergibt der Auftraggeber die physischen Adressen der zu transportierenden Datenblöcke der Elementaroperation `transfer`; die Funktionalität synchronenen IOs entsteht durch anschließendes Warten auf die Beendigung mittels `wait`. Bei asynchronem IO kann auch auf den Aufruf von `wait` verzichtet werden; weitere Details in [ST02].

Bei logischem IO werden logische und physische Adressen unterschieden; die Abbildung von logischen auf physische Adresse übernimmt hierbei die Operation `get`. Durch den paarweisen Aufruf von `put` wird die Ressourcen-Verwaltung (wer hat zu welchem Zeitpunkt Zugriff auf welche Ressourcen) auf ähnliche Weise wie bei einem konventionellen Buffer-Cache gelöst (z.B. interne Verwendung von Referenzzählern).

Damit ein Ausgang eines Bausteins mit mehreren Eingängen anderer Bausteine parallel verdrahtet werden darf (`multiuser`-Kompetenz), muss der *wechselseitige Ausschluss* (vgl. [Lag78]) bei parallelem Zugriff sichergestellt werden können. Hierzu sind die Operationen `lock` und `unlock` vorgesehen, die auf logischen Adressbereichen operieren. In [ST02] werden verschiedene Locking-Arten ausführlicher vorgestellt, die u.a. zur Konsistenzwahrung bei der Verschiebeoperation `move` eingesetzt werden können. Bei `singleuser`-Kompetenzen braucht kein Locking implementiert zu werden.

Zur Nachbildung von Pipes u.ä. sowie zur Speicherverwaltung existieren weitere Elementaroperationen `get_address` und `put_address`, mit denen das Problem der atomaren Reservierung von Adressbereichen insbesondere im Falle mehrerer paralleler Zugreifer lösbar ist.

Der hier vorgestellte Satz von Elementaroperationen auf Nestern ist als Beispiel zu verstehen, wie man die Schnittstelle zu Nestern gestalten kann.

3.2 Elementaroperationen auf dynamischen Nestern

Dynamische Nester enthalten alle Operationen eines statischen Nestes sowie zusätzlich die folgenden adressmodifizierenden Operationen. Sie haben mit der Lückenhaftigkeit des Definitionsbereich der Speicherabbildung und mit der Verschiebeoperation zu tun.

Die Operation `get_map` liefert Informationen über die definierten Bereiche eines dynamischen Nestes. Konzeptionell ist dies ein Array von Paaren, wobei jedes Paar die logische Startadresse eines definierten Bereichs und seine Länge repräsentiert. Die Paare werden nach den Startadressen aufsteigend sortiert gehalten. Zur effizienten Suche nach Adressen kann daher beispielsweise binäre Suche im Array verwendet werden. Da ein solches Array in Extremfällen sehr lang werden kann, wird es von `get_map` nicht direkt zurück geliefert, sondern statt dessen wird ein Verweis auf eine weitere Nest-Instanz zurück geliefert, die dieses Array enthält, und von der nur gelesen werden darf. Diese Nest-Instanz wird *adjungiertes Nest* genannt.

Adjungierte Nester haben ihrerseits auf jeden Fall keine Löcher, d.h. ihr adjungiertes Nest hat nur noch eine triviale Struktur mit einem einzigen Eintrag.

Die im adjungierten Nest enthaltene Liste hat eine besondere Bedeutung im Falle von Geräten mit dynamischen Block- oder Sektorgrößen (insbesondere Bandlaufwerke, bei denen jeder Datenblock eine individuelle Größe haben kann, oder sonstige zeichenorientierte Geräte, bei denen es auf die Länge von einzelnen Transfers ankommt), bei Pipes und bei Netzwerk-Sockets, bei denen die Paketgröße eines einzelnen Original-Eintrags vom Empfänger ermittelbar sein muss. In all diesen Fällen repräsentiert das adjungierte Nest dicht aneinander liegende Adressbereiche (ohne dazwischenliegende Lücken). Damit kann jeder Verwender selbst entscheiden, ob er einen Unix-typischen Zugriff auf das Nest unter Missachtung der ursprünglichen Paket- bzw. Datensatzgrenzen machen will, oder ob er zuvor die Paket- bzw. Datensatzgrenzen im adjungierten Nest nachsehen und

entsprechend berücksichtigen will.

Ein dichtes Aneinanderliegen von definierten Bereichen ist auch bei persistenten Nestern möglich. Die Benutzer werden jedoch gebeten, derlei in der Praxis möglichst zu meiden, da es zu Verständigungsproblemen mit den Datei-Paradigmen anderer aktueller Betriebssysteme kommen könnte.

Konzeptuell gesehen ist ein Nest somit weit mehr als eine Datei unter Unix, da es untrennbar mit seinem Definitionsbereich verknüpft ist, der bei der hier vorgestellten Architektur in einem adjungierten Nest repräsentiert wird.

Bei der Repräsentation einer Unix- oder Windows-Datei in einem Nest enthält das zugehörige adjungierte Nest nur einen einzigen Eintrag mit der Startadresse 0 und der Länge der Datei.

Die Operation `clear` erweitert den Definitionsbereich eines Nestes an einer wählbaren Stelle und Länge, soweit sich vorher dort eine Lücke befand. Falls anschließend in diesem Bereich des Adressraumes gelesen wird, erscheinen Null-Byte-Datenblöcke. Dies gilt auch dann, wenn vorher bereits Datenblöcke dem betroffenen Bereich ganz oder teilweise zugeordnet waren. In diesem Fall bleiben bereits mittels `get` ausgelieferte Datenblöcke weiterhin verwendbar, werden jedoch als *verwaiste* (*orphane*) Datenblöcke behandelt (analog zu einigen bekannten Buffer-Cache-Implementierungen).

Die Operation `delete` bewirkt, dass an der gewählten Stelle und Länge ein Loch im Definitionsbereich des Nestes entsteht bzw. bereits vorhandene Löcher ggf. erweitert werden; damit stellt es eine inverse Operation zu `clear` dar.

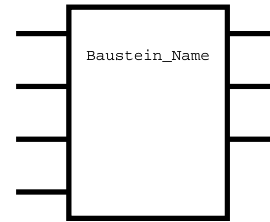
Die Operation `move` verschiebt einen wählbaren Bereich des Adressraums um einen Offset (positiv oder negativ). Der Inhalt der verschobenen Datenblöcke wird hierbei nicht verändert, lediglich die Zuordnung von logischen zu physischen Adressen wird verändert. Löcher werden ebenfalls mitverschoben; weitere Details in [ST02].

Die Operation `get_meta` liefert eine Referenz auf das sogenannte *Meta-Nest* aus, die einem Nest zugeordnet sein kann (die Existenz eines Meta-Nestes ist jedoch keine Pflicht).

Meta-Nester sind dazu gedacht, um Informationen *über* ein Nest bereitzustellen, analog zu Datei-Attributen oder Inode-Informationen, jedoch nicht ausschließlich auf diese Verwendungszwecke beschränkt. Meta-Nester spielen u.a. eine Rolle bei der Einführung von Typsystemen, ebenso bei der Auto-Instantiierung von Bausteinen (beispielsweise zur Herstellung von Netzwerktransparenz oder zum automatischen Zwischenschalten von Anpassungs-Bausteinen).

Ein Meta-Nest enthält nur einen einzigen definierten Bereich ohne Löcher, der idealerweise nur relativ wenig Platz beanspruchen sollte (d.h. er sollte nicht zur Speicherung großer Datenmengen missbraucht werden). Bei einem persistent gehaltenen Original-Nest muss das zugehörige Meta-Nest, sofern es existiert, ebenfalls persistent gehalten werden. Falls absturzsichere atomare Transaktionen implementiert werden, muß das Meta-Nest bezüglich dieser Semantik wie ein Teil des Original-Nestes behandelt werden.

4 Bausteine



Eine Baustein-Instanz ist ein Objekt, das eine beliebige, eventuell auch zur Laufzeit sich ändernde Anzahl von Ein- und Ausgängen besitzt, die wiederum jeweils Instanzen von Nestern darstellen.

Bausteine werden ähnlich wie in der Elektro- und Digitaltechnik als Kästchen mit linksseitigen Eingängen und rechtsseitigen Ausgängen gezeichnet. Als Verdrahtungsregel gilt, dass Eingänge nur mit den Ausgängen anderer Bausteine verknüpft werden dürfen und umgekehrt, wobei ein Eingang nur mit einem einzigen Ausgang, ein Ausgang hingegen i.A. mit mehreren Eingängen verknüpft werden darf.

Ausgänge stellen damit Nester anderen Bausteinen zur Verfügung, wobei deren Eingänge als *Konsumenten* der von den Ausgängen angebotenen *Dienstleistungen* anzusehen sind; wir haben also ein System von *Produzenten* und von *Konsumenten* im *logischen* Sinne. Eine Verdrahtungs-Leitung repräsentiert eine *Hierarchie-Beziehung* zwischen einem *vorgeschalteten* Baustein (Produzent der Dienstleistung) und einem *nachgeschalteten* (Konsument der Dienstleistung). Per Konvention definieren wir als „Stromrichtung" einer Verdrahtungs-Leitung die Richtung vom Produzenten zum Konsumenten; dies hat jedoch nichts mit möglichen Datenfluss-Richtungen zu tun, denn die Stromrichtung ist zwar gleichzeitig die logische Datenfluss-Richtung beim Lesen, doch die logische Schreib-Datenflussrichtung läuft dem entgegen (was am Anfang zu Verwirrung führen kann, ebenso die baustein-interne Weitergabe von Operations-Aufrufen, die intern meistens von den Ausgängen her zu den Eingängen verläuft). Eine Leitung stellt alle Operationen, die am Ausgang eines Bausteins zur Verfügung gestellt werden, einem oder mehreren Konsumenten zur Verfügung. Da dies auch die Operationen `get_map` und `get_meta` umfasst, wird auf diese Weise das zugehörige adjungierte Nest und das Meta-Nest verfügbar gemacht.

Sinn der Bausteine ist, verschiedene Transformationen sowohl des Adressbereiches als auch eventuell des Inhaltes von Nestern, gelegentlich auch des zugehörigen Meta-Nestes oder des Operations-Nestes (Abschnitt 5), von Nest-Attributen oder von Zugriffs-Modellen durchzuführen.

Bausteine besitzen mindestens eine Instantiierungs- und Konstruktor-Operation, mit der sich neue Instanzen des jeweiligen Baustein-Typs erzeugen lassen, wobei die Parameter von Konstruktor-Operationen i.d.R. baustein-spezifisch sind. Die Destruktor-Operation hat hingegen eine einheitliche Schnittstelle. Einige Baustein-Typen haben darüber hinaus weitere spezifische Operationen, mit denen sich ihr Verhalten (etwa die Anzahl der Ein- und Ausgänge) zur Laufzeit steuern läßt.

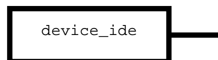
Eine automatisierte Überprüfung der Kompatibilität der Kompetenzen von Ausgängen mit dem Verhalten von Ein-

gängen findet bei der Verdrahtungs-Operation statt. Viele der nachfolgend vorgestellten Bausteine lassen sich in verschiedenen Ausbaustufen und mit verschiedenen Kompetenzen und Verhalten implementieren.

Beim Entwurf neuer Baustein-Arten sollte darauf geachtet werden, dass möglichst wenig Redundanz zur Funktionalität bereits vorhandener Baustein-Arten auftritt. Sinn der Zerlegung in Bausteine ist, die in einem Betriebssystem insgesamt zu lösenden Aufgaben in möglichst viele, kleine, voneinander möglichst unabhängige (orthogonale) Teile und Zuständigkeiten aufzuspalten (vgl. Abschnitt 2.1.3).

Ich habe mich bemüht, bei der hier als beispielhaft zu verstehenden Zerlegung möglichst nur solche Aufgaben zu stellen, die auch in anderen aktuellen Betriebssystem-Entwürfen (einschließlich Netzwerk-Betriebssysteme) auftauchen und dort mit teils hohem Aufwand ad hoc gelöst werden. Damit möchte ich zeigen, dass eine Komponenten-Zerlegung auf Basis der Abstraktionen Nest und Baustein den Gesamtaufwand der zu implementierenden Aufgaben mindert, da bereits die Kombination von wenigen wiederverwendbaren Baustein-Arten eine mächtige Funktionalität erzeugt.

4.1 `device_*`



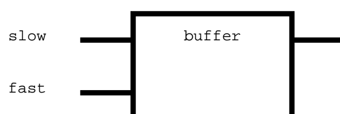
Diese Bausteine haben im Regelfall keinen Eingang und genau einen Ausgang, womit sie den Inhalt eines Gerätes (beispielsweise `device_ide` für IDE-Festplatten) als statisches Nest den etwaigen Konsumenten zur Verfügung stellen. Im Sinne der Verdrahtungslogik der Bausteine stellen sie Produzenten oder auch „Datenquellen“ dar.

Normalerweise dienen Geräte-Treiber zum Transfer von Datenblöcken auf Peripheriegeräte und brauchen daher nur die Operationen statischer Nester zu implementieren.

Eine Sonderform ist `device_mem`, die transienten Speicher ohne `transfer`-Operationen zur Verfügung stellt, indem sie die Operationen `get_address`- und `put_address` implementiert. Diese Baustein-Art erledigt damit die Funktionalität einer Speicherverwaltung.

Eine weitere `multiuser`-fähige Sonderform ist `device_ramdisk`, die eine limitierte Persistenz implementiert, die sich nicht über Stromausfälle hinweg erstrecken muss.

4.2 `buffer`



Ein `buffer`-Baustein sorgt für die Entkoppelung von Aktivitäten zwischen Eingang und Ausgang (in beide Richtungen). Er eignet sich zur Adaption des *zeitlichen Zugriffsverhaltens* zwischen langsamen und schnellen Baustein-Instanzen; im Idealfall sollte er die einzige Stelle im Gesamtsystem darstellen, die die Probleme der sogenannten „Speicherlücke“ zu lösen hat.

Ein häufiger Anwendungsfall ist die Nachschaltung hinter ein `device_*`. Dazu sind oft nur die Operationen statischer Nester erforderlich, und der Eingang braucht nur `singleuser`-Verhalten zu zeigen; auch der Ausgang braucht nur `singleuser`-Kompetenz bereitzustellen, da im häufigsten Anwendungsfall nur eine einzige `map_*`-Instanz nachgeschaltet wird. Es gibt aber auch Anwendungen, bei denen mindestens `multiuser`-Verhalten und -Kompetenz benötigt wird, insbesondere die Nachschaltung hinter einen `remote`-Baustein (siehe Abschnitt 4.11).

Bei der Implementierung des `multiuser`-Verhaltens tritt das in der Literatur bekannte Problem der *Cache-Kohärenz* (vgl. [AB86, LH89, HP95]) mit einer vorgeschalteten Instanz auf. Dieses Problem wird durch `notify_*`-Operationen gelöst, die in [ST02] näher beschrieben sind.

Wenn man `buffer`-Bausteine als die einzige Stelle im Gesamtsystem ansieht, die die Entkoppelungs-Problematik des *zeitlichen Zugriffsverhaltens* löst, und zwar *sehr effizient* löst, dann kann man auch noch einen Schritt weitergehen: man kann fast alle anderen Bausteine intern *statuslos* implementieren, oder zumindest weitgehend statuslos. Ein Baustein ist statuslos, wenn er zu jedem Zeitpunkt, in dem sich kein logischer Kontrollfluss in ihm aufhält, destruiert und anschließend erneut konstruiert werden kann, ohne daß dadurch ein geändertes Verhalten von außen sichtbar wird.

Statuslosigkeit ermöglicht eine deutliche Reduktion der inneren Komplexität vieler Bausteine im Vergleich zu konventionellen Implementierungen, weil die Verantwortung zur korrekten Aufbewahrung der internen Zustandsinformation an einen untergeordneten Baustein delegiert wird; daher ist Statuslosigkeit hochgradig erstrebenswert. Sie setzt allerdings voraus, dass Zugriffe über eine dermaßen entkoppelte Schnittstelle *fast nichts kosten*. Durch standardmäßige Benutzung von Prozeduraufrufen als Schnittstellen-Mechanismus wird dies ermöglicht.

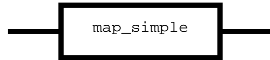
Der mit `slow` bezeichnete Eingang soll zur Entkoppelung der Zugriffs-Häufigkeiten und -Anzahlen dienen. Das mit `fast` bezeichnete Eingangs-Nest enthält den *gesamten Status* des Puffers; dazu gehört neben den Inhalten der zu puffernden Datenblöcke auch die transiente Zuordnung zwischen logischen und physischen Adressen und der Aktualitäts-Status. Der `buffer`-Baustein wird dadurch selbst vollkommen statuslos.

Damit dies zu guter Performanz führt, muss `fast` sehr schnellen Zugriff bieten. Die Idee besteht darin, an diesem Eingang wahlweise ein `device_ramdisk` anzuschließen, oder ein anderes relativ schnelles Gerät; bei der Pufferung von Zugriffen auf extrem langsame Bandlaufwerke kann dies beispielsweise auch ein Nest sein, dessen Inhalt auf Festplatte vorgehalten wird. Letztlich macht `buffer` nichts anderes, als das zeitliche Zugriffsverhalten des `fast`-Eingangs an den Ausgang weiter zu reichen. Die am `fast`-Eingang angeschlossene Instanz wird mit geringeren Datenmengen belastet als beim `slow`-Eingang vorhanden sind.

Aufgrund von absehbaren Fortschritten in der Hardware-Entwicklung ist zu erwarten, dass zukünftige Rechner ganz andere interne Speicher-Hierarchien besitzen werden, als

sie heute üblich sind. Das Baustein-Konzept ermöglicht eine flexible Anpassung an geänderte Rahmenbedingungen.

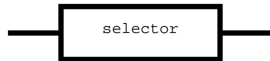
4.3 map_*



Aufgabe dieses Baustein-Typs ist, ein statisches Nest in ein dynamisches umzuwandeln. Es gibt daher nur einen Eingang (der meist nur singleuser-Verhalten zu implementieren braucht) und einen Ausgang, der im Falle von Netzwerk-Betriebssystemen mindestens multiuser-Kompetenz bereitstellen sollte; die Herstellung dieser Kompetenz kann aber auch an eine nachgeschaltete oder interne adaptor_*-Instanz (siehe Abschnitt 4.8) delegiert werden.

Es lassen sich verschiedene Arten von map_*-Bausteinen realisieren, die ihre Aufgabe mit jeweils anderen internen Realisierungsverfahren lösen und an spezielle Last- und Benutzungsmodelle angepasst sind. Beispiel-Realisierungen, insbesondere zur effizienten Herstellung der move-Funktionalität, sind in [ST02] beschrieben.

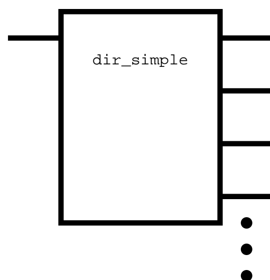
4.4 selector



Dieser Anpassungs-Baustein schneidet einen zusammenhängenden Teil-Adressraum aus seinem Eingangs-Adressraum aus und stellt ihn unverändert am Ausgang zur Verfügung, wobei die Adressen standardmäßig wieder bei 0 neu beginnen. Wird der ausgeschnittene Adressraum als Ganzes verschoben, dann wirkt sich das nicht auf den Gastgeber-Adressraum aus, ebenso umgekehrt bei Überstreichung einer move-Operation im Gastgeber-Adressraum über den gesamten ausgeschnittenen Bereich (Transparenz).

Die Implementierung ist relativ einfach, da außer einer uniformen Adressübersetzung und -Überprüfung nichts zu machen ist und alle Operationen vom Ausgang an den Eingang durch gereicht werden können.

4.5 dir_*



Ein Baustein dieser Art benutzt das Eingangs-Nest als eine Art Sammel-Container, um mehrere von einander unabhängige Ausgangs-Nester daraus zu bilden. Die Ausgangs-Nester werden bei Bedarf durch die baustein-spezifische Operation create neu erstellt; bereits früher erstellte werden durch lookup erneut instantiiert, sofern sie nicht bereits instantiiert sind. Unterschieden werden die verschiedenen möglichen Ausgangs-Instanzen durch einen Suchschlüssel. Zur Abfrage aller vorhandenen Schlüsselwerte dient ein spezieller Ausgang, der Verzeichnis-Nest genannt wird, von dem nur gelesen werden darf. Ein Verzeichnis-Nest stellt ein Nest mit lückenlos liegenden Datenpaketen dar, wobei jedes Paket genau einen der vorhandenen Suchschlüssel enthält. Die Schlüsselwerte können, müssen aber nicht nach irgendeinem internen Kriterium sortiert gehalten werden.

Damit ist ein Verzeichnis realisierbar, wie es in konventionellen Betriebssystem-Architekturen von *Dateisystemen* zur Verfügung gestellt wird. Ein dir_* stellt jedoch keine Verzeichnisbaum-Hierarchie zur Verfügung, sondern ähnelt eher dem flachen Index einer Datenbank. Dennoch lassen sich Verzeichnisbaum-Hierarchien sehr leicht herstellen: am Ausgang einer dir_*-Instanz braucht lediglich eine weitere dir_*-Instanz angeschlossen zu werden und so weiter. Auf diese Weise kann der Dateisystem-Baum (bzw. ein momentan instantiierter Teilbaum davon) direkt als Baumstruktur von Baustein-Instanzen mit der zugehörigen Verdrahtung dargestellt werden. Wenn man die in Abschnitt 6 vorgestellte Auto-Instantiierung von Bausteinen benutzt, dann braucht man sich als Benutzer nicht um die dynamische Herstellung dieser Baumstruktur zu kümmern, da dies automatisch geschieht.

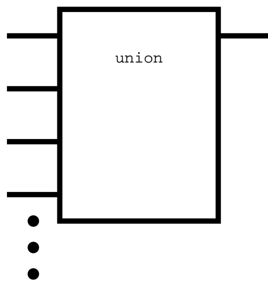
Im Vergleich zur Funktionalität klassischer Dateisysteme ermöglicht dieses Konzept eine wesentlich flexiblere Instantiierung und Verdrahtung, da man beispielsweise

- für jedes Verzeichnis individuell verschiedene Baustein-Typen einsetzen kann, beispielsweise dir_simple für kleine Verzeichnisse, dir_hash für solche mit besonders schneller lookup-Funktionalität, oder dir_btree für solche mit besonders guten Lokalitätseigenschaften trotz riesiger Ausdehnung
- Datenkomprimierungs-, Datenverschlüsselungs- und sonstige Anpassungs-Bausteine wie adaptor_* und andere auf beliebigen Hierarchieebenen (automatisch) dazwischen schalten kann
- auf Konzepte wie Mounts und Mount-Tabellen, Loopback-Devices etc. verzichten kann
- nahtlose Integration mit der Funktionalität von Datenbanken möglich ist: dazu zählt nicht nur der später vorgestellte transaction-Baustein, sondern auch die Möglichkeit, spezialisierte Bausteine wie beispielsweise dir_fixed_keysize einzusetzen, bei denen eine Uniformität der Schlüssellängen zugunsten besserer Platzausnutzung erzwungen wird
- eine *virtuelle* Herstellung von Verzeichnisinhalten durchführen kann, beispielsweise mit einem Baustein dir_proc für die Funktionalität von /proc-Dateisystemen, oder dir_join zur Herstellung der Join-Operation aus dem Daten-Inhalt mehrerer anderer

Nester. In diesem Fall können dir_*-Varianten entstehen, die gar keinen oder mehrere Eingänge besitzen, was sich u.U. mit dem althergebrachten Konzept eines Dateisystem-Baums nicht auf intuitive Weise modellieren läßt.

Die hier vorgestellte Baustein-Zerlegung verteilt die in Dateisystemen vorkommenden Problematiken und Funktionalitäten auf mehrere Baustein-Arten, isoliert sie voneinander, und ermöglicht vorher unbekannte Kombinationen. Einige der Möglichkeiten wie z.B. die Kombination mit dem mirror-Baustein (siehe Abschnitt 4.12) werden auf einfachere Weise als mit stapelbaren Dateisystemen (vgl. [HP94, HP95]) gelöst, da nicht mehr zwischen verschiedenen Ebenen wie „Dateien" versus „Dateisystem-Bäume" unterschieden wird.

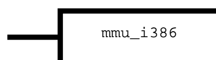
4.6 union



Dieser Baustein stellt in gewisser Hinsicht eine inverse Operation zu selector bzw dir_* dar: mehrere Eingangs-Nester erscheinen in einem Ausgangs-Nest und werden dabei adressmäßig neben einander aufgereiht. Über Parameter bzw. Meta-Informationen wird festgelegt, ob sich ein Eingangs-Adressraum lückenlos an seinen Vorgänger anschließen soll (so dass eventuelle Lücken am Ende des Vorgängers und am Anfang des eigenen Nestes zusammen geschoben werden und move-Operationen des Vorgängers mit vollzogen werden), oder ob er ggf. unter Lückenbildung an festen Adressen „festgenagelt" erscheinen soll (analog zu selector). Weitere Spielarten sind denkbar.

Ein wichtiges Anwendungsgebiet von union ist die Zusammenstellung verschiedener Regionen bzw. „Segmente" in virtuellen Adressräumen oder Schutzbereichen, beispielsweise die Einteilung in Code-, Stack- und Datensegmente, sowie in Mappings anderer Nester. Verschiedene Mapping-Arten lassen sich durch Vorschalten von Anpassungsbausteinen wie cow (Abschnitt 4.9) realisieren.

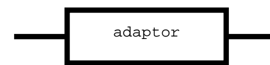
4.7 mmu_*



Dieser Baustein besitzt genau einen Eingang und keinen Ausgang, ist also im Sinne der Verdrahtungslogik ein reiner Konsument. Er stellt die Schnittstelle zur Memory-Management-Unit (MMU) der Rechner-Hardware dar. Das Nest wird hierbei 1:1 in einen virtuellen Adressraum umgewandelt, der sich über Kontrollflüsse direkt durch Maschinenbefehle des Prozessors adressieren lässt.

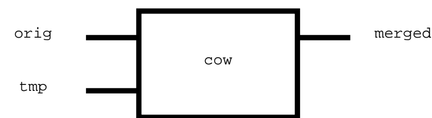
Die Realisierung von Schutzbereichen ist ebenfalls Aufgabe von mmu_*. Verschiedene Schutzbereiche lassen sich am einfachsten durch Zuordnen verschiedener Mandate (vgl. Abschnitt 2.6) und unterschiedlicher Behandlung in untergeordneten check_*-Instanzen implementieren. Damit werden mmu_*-Instanzen zu Verwaltern derjenigen Mandate, die mit den Schutzbereichen zu tun haben.

4.8 adaptor_*



Es handelt sich um einen Anpassungs-Baustein mit nur einem Eingang und einem Ausgang, der zwischen Nestern mit verschiedenen Kompetenzen und Verhalten wie beispielsweise verschiedenen transfer_size-Attributwerten vermittelt und übersetzt. Weitere Beispiele wie das nachträgliche Einführen von multiuser-Kompetenzen in eine Baustein-Hierachie sind in [ST02] näher ausgeführt.

4.9 cow

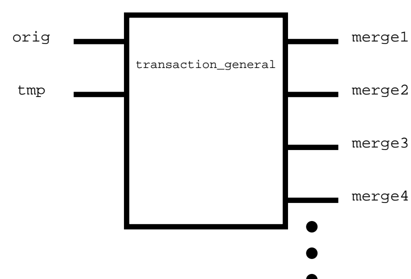


Dieser Baustein realisiert die von konventionellen privaten Mappings bekannte Copy-on-Write-Strategie.

Im initialen Zustand ist das mit tmp bezeichnete Eingangs-Nest leer, und am mit merged bezeichneten Ausgang erscheint exakt derselbe Nest-Zustand wie am orig-Eingang. Die Aufgabe besteht in der *Isolation* des orig-Eingangs von allen Änderungen, die vom Konsumenten hinten am merged-Ausgang in Auftrag gegeben werden. Jegliche Änderungen am Nest-Inhalt des Ausgangs oder an seinem Adressraum werden ausschließlich im tmp-Nest gepuffert, so dass sie keine tatsächlichen Änderungen am orig-Eingang bewirken.

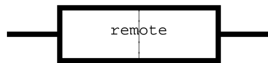
4.10 transaction

Im Unterschied zu Datenbanken, wo Transaktions-Identifier (TIDs) meist fest mit Prozessen verknüpft sind, verstehe ich unter einer Transaktion eine *logische Sicht* auf einen Datenbestand, die die bekannte ACID-Eigenschaft (oder Eigenschaften anderer Transaktions-Paradigmen) besitzt, und die von beliebig vielen Prozessen kooperativ (bzw. bei Verwendung zusätzlicher Sperren auch im multiuser-Modell) nutzbar ist.



Durch Kombination von Transaktionen mit Schutzbereichen lassen sich Funktionalitäten erzeugen, die in konventionellen Betriebssystem-Implementierungen einen *erheblichen* Aufwand verursachen würden. Ein Beispiel ist das Exeption-Handling in konventionellen Laufzeitumgebungen. Dieses kann auf einen Fehler nur noch *reagieren*, ihn aber meist nicht *reparieren*, da es beim Auftreten des Fehlers bereits „zu spät“ ist. Transaktionen bieten die Möglichkeit, auf frühere Zustände von Adressräumen zurückzusetzen und erneute Versuche zu starten (*spekulative Ausführung*).

4.11 `remote`

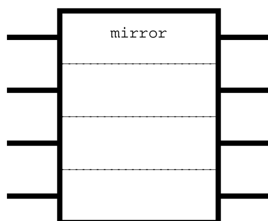


Dieser Baustein implementiert das Client-Server-Paradigma. Ein Nest wird auf einem anderen Rechner so verfügbar gemacht, dass seine Eigenschaften nicht von einem lokalen Nest unterscheidbar sind.

Die interne Realisierung des Netzwerk-Protokolls kann statuslos erfolgen, wenn man eine `buffer`-Instanz nachschaltet, die durch ihr Caching den Datenverkehr über die interne Netzwerk-Verbindung in vielen Fällen reduziert und nebenbei die Latenzzeit vieler Operationen senkt. Bei einer statuslosen Realisierung hält der Client-Teil keinerlei Status-Informationen über den Zustand des Nestes vor, sondern muss jede einzelne Operation an den Server durch reichen. Dies führt zu einer hohen Einfachheit, Robustheit, Unabhängigkeit und Absturzsicherheit.

Fragen der Einbruchssicherheit in Netzwerke, Verschlüsselung, Authentifizierung usw. sind eine interne Angelegenheit der Baustein-Implementierung; in diesem Papier wird darauf nicht weiter eingegangen.

4.12 `mirror`



Dieser Baustein realisiert das Konzept von Verteiltem Gemeinsamen Speicher (distributed shared memory).

Eine Baustein-Instanz von `mirror` darf sich über mehrere physikalisch getrennte Rechner erstrecken, die miteinander über interne (nicht von außen sichtbare) Kommunikationsmechanismen in Verbindung stehen (hierfür bietet sich insbesondere Gruppenkommunikation an). Die Verteilung der Baustein-Instanz über mehrere Rechner wird im Bild durch gestrichelte Linien angedeutet.

Ein- und Ausgänge sind im Grundmodell paarweise in der gleichen Anzahl vorhanden. Die Ausgänge stellen dieselbe Funktionalität dar, wie sie in einem nicht-verteilen System von der gemeinsamen Verdrahtung mehrerer Eingänge auf denselben Ausgang erfüllt wird: überall herrscht die gleiche logische Sicht, das Nest ist logisch betrachtet identisch und ermöglicht Kooperation im `multiuser`-Modell.

Diese knappe Funktionsbeschreibung läßt viele Möglichkeiten für die Realisierung zu, die im Wesentlichen den bekannten Techniken aus der Literatur folgen kann (siehe z.B. [TSF90, Doe96, Esk96]). Was die Eingänge enthalten, kann bei verschiedenen `mirror_*`-Typen stark voneinander abweichen. Ich werde zwei Extremfälle kurz erläutern:

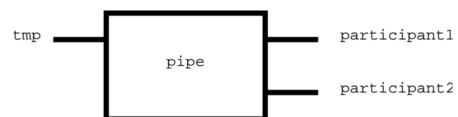
Bei einer Realisierung namens `mirror_replicate` enthalten alle Eingänge den gleichen Nest-Zustand wie die Ausgänge. Jede an irgendeinem Ausgang ankommende Änderung wird sofort auf alle Eingänge aller verteilten Teil-Instanzen durch gereicht. Bei parallelem Zugriff auf mehreren Rechnern erfordert dies im Regelfall verteilte interne Synchronisationsoperationen, um die *Konsistenz* jederzeit zu wahren.

Das dadurch verursachte Problem relativ hoher Latenzzeiten läßt sich durch rechner-lokales Nachschalten je einer `buffer`-Instanz hinter die Ausgänge in vielen Fällen abmildern; daher kann die interne Realisierung der Kommunikationsprotokolle weit gehend statuslos erfolgen.

Eine andere Realisierung namens `mirror_primarycopy` besitzt nur einen einzigen Eingang, der einem ausgezeichneten Rechner fest zugeordnet ist und stets dort verbleibt, auch wenn weitere Ausgänge auf anderen Rechnern dynamisch hinzu gefügt oder weg genommen werden.

Mischformen zwischen den beiden Extremen werden in [ST02] angesprochen. Die Verteilung von Nestern ist insbesondere zur Herstellung verteilter gemeinsamer Namensräume sehr nützlich; `mirror` ist somit die wesentliche Grundlage für ein verteiltes Betriebssystem.

4.13 `pipe`



Zum Verständnis des `pipe`-Entwurfs muss man sich nochmals in Erinnerung rufen, dass eine Leitung eine Hierarchie-Beziehung zwischen Baustein-Instanzen darstellt und nichts direkt mit Datenfluss-Richtungen zu tun hat. In abstrakter Sicht sind daher die Teilnehmer an einer Pipe prinzipiell gleichberechtigt, sie werden daher an verschiedenen Ausgängen angeschlossen. Der interne Status einer `pipe`-Instanz wird im `tmp`-Eingang gehalten, an den sich nicht nur `device_ramdisk` anschließen läßt, sondern z.B. auch Puffer-Nester, die nicht in den Hauptspeicher eines Kleinrechners passen würden.

5 Generische Operationen

Generische Operationen sind Operationen, die über eine *polymorphe Schnittstelle* aufrufbar sind.

Ein *Operations-Aufruf* ist die Übergabe von *Information* von einer *Aufrufer-Instanz* an eine *Bearbeiter-Instanz.*

Nach dieser Definition ist die Rückgabe von Ergebnissen, z.B. in Ergebnis-Parametern, ebenfalls ein Operations-

Aufruf, nur in umgekehrter Richtung. In diesem Fall spricht man von einem *Operations-Aufrufs-Paar*.

Operations-Aufrufe bzw -Paare lassen sich auf verschiedene Weisen realisieren. Bekannte Realisierungs-Arten sind z.B. Prozeduraufrufe mittels Übergabe von Parametern auf dem Stack eines Programmiersprachen-Laufzeitsystems, oder der RPC (Remote Procedure Call). Ersterer Fall läßt nur den *synchronen Aufruf*, letzterer Fall auch den *asynchronen Aufruf* zu. Weitere bekannte Realisierungs-Arten sind das Starten von Kontrollflüssen (asynchrones Verhalten), oder die Weitergabe der Flusskontrolle in einem Coroutinen-Modell durch Aufruf einer Transfer-Operation (dies bewirkt *synchrones* Verhalten aus Sicht des Gesamtsystems).

Ich möchte auf eine weitere bekannte Art der Realisierung von Operations-Aufrufen fokussieren: das so genannte *Nachrichten-Paradigma*, das zwar oft im Kontext von CSP (siehe [Hoa78]) in Erscheinung tritt, prinzipiell jedoch unabhängig von sequentiellen Kontrollflüssen ist. Das Nachrichten-Paradigma wird grundlegend in der Hardware verwendet und steckt z.B. hinter den weit verbreiteten read-write-Schnittstellen, bei denen ja auch Informationen von einer Instanz an eine andere weitergegeben werden.

Man sollte sich klar machen, dass zwischen einer reinen Nachrichten-Übertragung und einem Operations-Aufruf *überhaupt kein prinzipieller Unterschied* besteht. Ein solcher Unterschied entsteht erst durch die *Interpretation* der Nachricht durch den Empfänger, sowie ggf. durch seine *Reaktion* auf die interpretierte Nachricht.

Dies bedeutet für unsere Betriebssystem-Architektur, dass wir zur Realisierung von generischen Operationen keine neuen Konzepte einführen müssen, sondern die bereits ausführlich vorgestellten Abstraktionen Nest und Baustein verwenden können.

Konkret wird dies z.B. auf folgende Weise realisiert: Einer Nest-Instanz s wird ein so genanntes *Operations-Nest* als dynamisches Attribut zugeordnet, oder mit Hilfe einer einer eigenen Elementarfunktion $\texttt{get_ops}(s)$. Das Operations-Nest dient zum Austausch der Nachrichten für die generischen Operationen, die auf dem *zugeordneten Nest* s ausgeführt werden sollen. Zu einem Operations-Nest op kann man umgekehrt das zugeordnete Nest durch eine Elementarfunktion $\texttt{get_nest}(op)$ zugänglich machen. Es gilt dann $\texttt{get_nest}(\texttt{get_ops}(s)) = s$.

Zum Austausch der Nachrichten werden die Elementaroperationen `get_address`, `put_address`, `get`, `put`, `transfer` und `wait` verwendet. Das Operations-Nest darf zu einem beliebigen Zeitpunkt mehrere Operations-Aufrufe enthalten und ist daher prinzipiell geeignet, um die Funktionalität von Auftrags-Warteschlangen, Gerätetreiber-Warteschlangen u.ä. auszuführen.

Zum Übertragen der Nachrichten in einem Operations-Nest müssen auf jeden Fall Elementaroperationen verwendet werden, die an einen Schnittstellen-Mechanismus aus Abschnitt 2.3.2 gebunden sind, und mit deren Hilfe erst das Konzept der generischen Operationen auf einer höheren Abstraktionsstufe implementiert wird.

Nun zur Problematik der Polymorphie. Aufrufer und Bearbeiter müssen eine generische Operations-Nachricht auf zueinander passende Weise interpretieren, sonst entsteht eine Fehlfunktion des Systems.

Das Konzept des Meta-Nestes ist dafür vorgesehen, um Informationen über die Interpretation eines Nest-Inhalts aufzubewahren. Dies wird analog auf Operations-Nester übertragen. Das Operations-Meta-Nest soll einen *Konsens* zwischen dem Aufrufer und dem Bearbeiter über die zu verwendenden *Datenformate* bzw *Datentypen* vermitteln.

Wir brauchen also ein *Typsystem*, wie es prinzipiell bei Programmiersprachen schon lange eingesetzt wird.

Eine einfache Realisierung eines Minimal-Typsystems besteht darin, dass das Meta-Nest des Operations-Nestes eine Liste der implementierten zueinander disjunkten Operationsnamen enthält. Dieses minimale Typsystem ist bereits insofern *erweiterbar*, als dass der Wertevorrat für die Kodierung der Operationsnamen so groß gewählt werden kann, dass zukünftige Erweiterungen um neue Operationen praktisch unbeschränkt möglich sind. Es geht jetzt um die Frage von *Schnittstellen-Erweiterungen*, die man unter die Rubrik „Erweiterungs-Generizität“ (vgl. Abschnitt 2.1.2) einordnen kann. Vor dem Einsatz von Erweiterungs-Generizität ist gewissenhaft zu prüfen, ob der gleiche Effekt nicht auch durch kompositorische Generizität (vgl. Abschnitt 2.1.3) erreichbar ist. Kompositorische Generizität hat grundsätzlich Vorrang vor Erweiterungs-Generizität; dies ist ein wichtiges Unterscheidungsmerkmal des hier vorgestellten Ansatzes gegenüber den meisten konkurrierenden Ansätzen.

In der Literatur aus dem Gebiet der Programmiersprachen werden Typsysteme als *algebraische Struktur* aufgefasst, für die eine *abstrakte Syntax* angegeben werden kann und meist auch nur angegeben wird. Dies genügt i.A. für unsere Zwecke nicht: ein Betriebssystem muss auch die *konkreten Datenformate* (sog. Aufruf-Konventionen) festlegen, da die System-Schnittstellen übergreifend für unterschiedliche Aufrufer-Typen gelten sollen, insbesondere den Anschluss an unterschiedliche Programmiersprachen und Laufzeitsystem-Modelle ermöglichen sollen. Wir benötigen daher eine Spezifikation der algebraischen Struktur in *konkreter Syntax*.

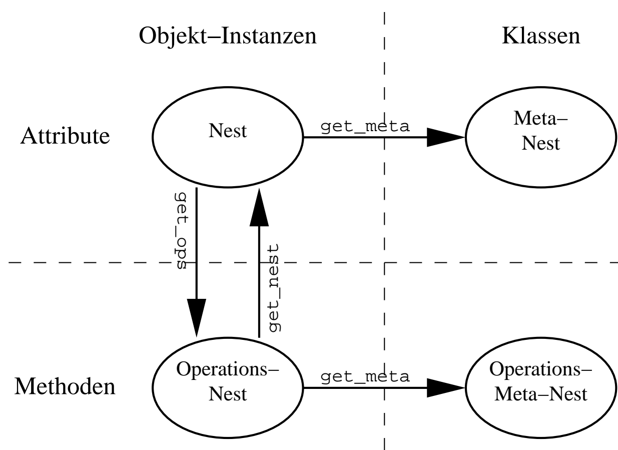
Zur Spezifikation einer konkreten Syntax eignen sich bekanntermaßen *kontextfreie Grammatiken*. Damit keine mehrdeutigen Interpretationen von Datenformaten möglich sind, sollten deterministische Unterklassen wie $LR(k)$ oder $LL(k)$ verwendet werden.

Für die Zwecke von Betriebssystemen dürfte die Klasse derjenigen $LL(1)$-Grammatiken ausreichend sein, die sich zusätzlich in Greibach-Normalform befinden. Das Wortproblem bzw. das Parsing-Problem ist bei diesen Grammatiken trivial lösbar, da man beispielsweise das bekannte Verfahren des *rekursiven Abstiegs* in *interpretativer Weise* direkt auf einer Kodierung der Grammatik ausführen kann, ohne die bei $LL(1)$ notwendigen FIRST-Mengen berechnen zu müssen, da diese trivialerweise mit dem ersten Terminalsymbol übereinstimmen, mit dem jede Grammatikregel wegen der Greibach-Normalform beginnen muss. Die beim rekursiven Abstieg notwendigen Parsing-Entscheidungen werden dadurch trivial. Ich nenne eine Greibach-Grammatik mit dieser Forderung eine *Typ-Grammatik*.

Eine Typ-Grammatik setzt die folgenden bekannten Grundprinzipien des Entwurfs von Datenstrukturen direkt in leicht zu handhabende Grammatik-Regeln um: die *Schachtelung*, die *Sequenz*, und die *Alternative*.

Im Gegensatz zu Programmiersprachen wird diese Methodik in Betriebssystemen vollkommen dynamisch eingesetzt; die Erkennung eines Datenformats kann i.A. nicht vom Typsystem eines Compilers vorweggenommen werden, sondern muss mindestens bei der Verdrahtungs-Operation zur Laufzeit durchführbar sein. Hierfür eignet sich die *Laufzeit-Interpretation* von Datenformaten mit Hilfe kontextfreier Grammatikregeln ganz besonders. Performanz-Steigerungen durch *Vorübersetzung* von Typ-Grammatiken in deterministische Kellerautomaten sind möglich.

5.1 Zusammenhang mit Objektorientierung



Das Bild zeigt den Zusammenhang der vier vorgestellten Nest-Arten mit den bekannten Konzepten der Objektorientierung. Die gestrichelten Trennlinien sollen eine Unterteilung des Feldes nach zwei verschiedenen Kriterien andeuten:

- Unterteilung in Objekt-Instanzen und Klassen: Nest-Instanz versus zugehöriges Meta-Nest
- Unterteilung in Attribute und Methoden: Daten-Nest versus Operations-Nest

Die Beziehungen von Instanzen der jeweiligen Nest-Art sind durch Pfeile mit Beschriftung der zugehörigen Zugriffsfunktionen gekennzeichnet. Eine Nest-Instanz steht mit ihrer zugehöriger Operations-Nest-Instanz in einer 1:1-Beziehung, genau wie beim Zusammenpacken von Attributen und Methoden in der Objektorientierung zu einer Objekt-Instanz. Der *Status* einer Objekt-Instanz wird dabei in der Nest-Instanz abgespeichert.

Der Zusammenhang mit der *Vererbung* läßt sich durch zwei Anpassungs-Bausteine darstellen, die entsprechende Transformationen auf den Nestern bzw. Meta-Nestern durchführen.

Beim `downcast` wird die Schnittstelle erweitert; zu den am Eingang `base_class` verfügbaren Attributen und Methoden können neue hinzukommen, die am Ausgang `son_class` zur Verfügung gestellt werden; ggf. kann dabei auch die *Implementierung* (bzw. das *Verhalten*) einiger Methoden abgeändert werden (*überschriebene Methoden* und *virtuelle Methoden*). Da sich dabei der Umfang des abgespeicherten Status vergrößern kann, ist ein Eingang `tmp` vorgesehen, um diesen aufzunehmen. Bei geeigneten Konventionen über die Platz-Allokation im `base_class`-Nest läßt sich der Status aber auch dort abspeichern, so dass `tmp` überflüssig wird.

Beim `upcast` wird lediglich die Schnittstelle verkleinert, so dass nicht mehr alle Attribute und Methoden von außen zugreifbar sind (Prinzip der *Verbergung*).

Aus dieser Beschreibung wird ersichtlich, dass die Konzepte der Objektorientierung einen *Spezialfall* der hier vorgestellten Methodik darstellen.

6 Instantiierung von Bausteinen

Zur Instantiierung der Bausteine wird irgendein Mechanismus benötigt. Dieser sollte außerhalb oder oberhalb der eigentlichen Baustein-Hierarchie liegen, da es sich um einen übergeordneten Kontroll-Mechanismus handelt. Nach dem Grundsatz der Trennung in Mechanismen und Strategien sind folgende Teilbereiche zu verwenden:

1. Der eigentliche Instantiierungs-Mechanismus (technische Durchführung der Instantiierung)

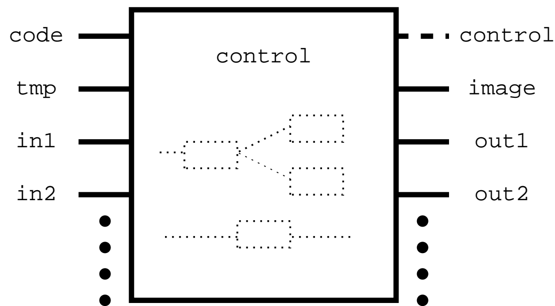
2. Die logische Kontrolle, welche Instantiierung zu welchem Zeitpunkt und aus welchem Anlass ausgeführt werden soll

Diese Trennung hat den Vorteil, dass insbesondere für Punkt 2 mehrere verschiedene Verfahren gleichzeitig oder konkurrierend eingesetzt werden können.

6.1 Der Mechanismus

Für die technische Durchführung der Instantiierung schlage ich vor, einen Baustein namens `control` einzusetzen, von dem es auf jedem Rechner beliebig viele Instanzen geben kann, wobei im Normalfall jedoch eine Instanz ausreicht, die ihrerseits beim Urstart auf eine Weise instantiiert worden sein muss, die ausserhalb des normalen Instantiierungs-Mechanismus liegt (und logischerweise auch liegen muss).

Der `control`-Baustein verwaltet alle von ihm erzeugten Baustein-Instanzen samt Verdrahtung. Er selbst benötigt ebenfalls Betriebsmittel, beispielsweise den Maschinencode von sich selbst.



Ich schlage vor, folgende Funktionsbereiche logisch zu trennen: Die von einer control-Instanz verwalteten Baustein-Instanzen können i.a. in einem anderen (virtuellen) Adressraum (der wiederum in Schutzbereiche aufgeteilt sein kann) liegen als die control-Instanz selbst; damit sind z.B. beliebige hierarchische Schachtelungen von Adressräumen oder ganzen virtuellen Maschinen möglich. Die steuernde control-Instanz bedient sich des tmp-Eingangs, um darin den gesamten notwendigen Status zu halten, und stellt am image-Ausgang ein „Prozessabbild" bzw. eine Menge von Schutzbereichen zur Verfügung, die in einer nachfolgenden (ggf. über Zwischenbausteine wie union angeschlossene) mmu_i386 oder mmu_dummy-Instanz zur eigentlichen Ausführung gebracht werden. Damit ist eine Separation zwischen dem Adressraum möglich, der die control-Instanz beherbergt, und dem Adressraum, in dem die verwalteten Instanzen laufen sollen. Durch diese Trennung ist es u.a. möglich, Teile eines Betriebssystems in „Benutzer"-Adressräumen ablaufen zu lassen. Auf diese Weise lassen sich voneinander unabhängige Subsysteme schaffen, die von der Funktionalität her virtuellen Betriebssystemen gleich kommen, diese jedoch an Flexibilität übertreffen. Damit verschwimmt die klassische Einteilung in Kern- und Benutzer-Zuständigkeiten; die Verteilung dieser Zuständigkeiten ist nur noch eine Frage der Konfiguration.

Eine Trennung zwischen der control-Instanz und ihrem image in separate Adressräume ist jedoch keine Pflicht[6]: wenn man den image-Ausgang in die gleiche mmu_*-Instanz einblendet, in der auch der control-Baustein beheimatet ist, kann man ohne weiteres Single-Address-Space-Strategien fahren. Als weiterer Sonderfall ist es prinzipiell auch möglich, dass ein control-Baustein sich selbst verwaltet[7]; in diesem Fall besteht kein Unterschied zwischen „innen" und „außen"; eine Destruktion der control-Instanz würde dann nicht nur zur Destruktion aller verwalteten Instanzen führen, sondern nie mehr eine erneute Re-Konstruktion ermöglichen (dies kann ausnahmsweise beim Herunterfahren des Rechners auch erwünscht sein).

Die Befehle zum Instantiieren / De-Instantiieren der verwalteten Instanzen sowie zum Verdrahten werden über den control-Ausgang gegeben. Eine Möglichkeit ist die Einführung weiterer Elementar-Operationen, oder die Benutzung der Schnittstelle von generischen Operationen. Da auf dieser Leitung nur Steuer-Operationen benutzt werden, ist sie gestrichelt gezeichnet. Über den control-Ausgang lassen sich ferner die Attribute der instantiierten Bausteine, Ein- und Ausgänge und die Verdrahtungs-Struktur abfragen. Auftraggeber für diese Operationen können beliebige Bausteine sein; i.A. dürfen diese auch in image liegen. Zu Zwecken der Kommunikation nach außen existiert eine dynamische Anzahl von Ein- und Ausgängen in* und out*, die als Schnittstellen für LRPC dienen und die zu beliebigen Zwecken einsetzbar sind.

Es ist sinnvoll, das Instantiieren / De-Instantiieren vom Konstruieren / Destruieren zu trennen: Beim Instantiieren wird lediglich Platz für die Baustein-Infrastruktur (z.B. statische Attribute) angelegt und die Verdrahtung ermöglicht. Die *Parametrierung* der Baustein-Instanz (z.B. Festlegen dynamischer Attribute) erfolgt später beim Konstruieren. Wenn man einen Baustein nur destruiert, aber nicht deinstantiiert, wird er im Endeffekt lediglich in den Zustand der Statuslosigkeit geschaltet (d.h. nach dem Abarbeiten aller evtl. noch vorliegenden Aufträge wird der evtl. noch vorhandene interne Status auf die Eingänge abgewälzt, danach werden keine Operationen mehr bearbeitet), und er kann anschließend durch Konstruieren wieder in Betrieb genommen werden.

6.2 Einige mögliche grundlegende Strategien

Bei den *Anlässen* für eine Instantiierung sind mehrere mögliche Szenarien zu unterscheiden:

1. Manche Baustein-Arten wie z.B. dir_* können nur an bestimmten vorgegebenen oder vorgebbaren Stellen einer Baustein-Hierachie instantiiert werden, weil sie einen *Interpreter* für ein bestimmtes *konkretes Datenformat* darstellen. Es macht i.a. keinen Sinn, andere Baustein-Arten als die für die jeweilige Interpretation geeigneten zu instantiieren.

2. Andere Baustein-Arten wie z.B. mmu_* oder geschachtelte control-Instanzen werden fast ausschließlich auf Veranlassung von Benutzer-Aktivitäten instantiiert.

3. Es gibt Mischformen zwischen beiden Extremen: ein Verzeichnis-Pfadname wird zwar vom Benutzer vorgegeben, die Art des jeweils zu instantiierenden Bausteins hängt jedoch vom Datenformat ab.

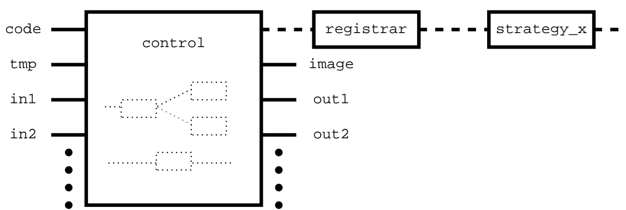
4. In manchen Fällen können Instantiierungen oder Re-Instantiierungen auch ohne *konkreten* äußeren Anlass

[6] Im Extremfall kann das gesamte Betriebssystem in einem einzigen Adressraum ablaufen, was z.B. bei Echtzeit-Steuerungen Performanz-Vorteile bringt. Falls Schutzbereiche genutzt werden, läßt sich trotzdem ein brauchbarer Zugriffsschutz realisieren (vgl. [C+94]). Im Unterschied zu bekannten Architekturen (vgl. [K+97]) erlaubt die hier vorgestellte Trennungsmöglichkeit beliebige Zwischenstufen zwischen reinen Single-Address-Space-Modellen und vollkommener Aufplitterung aller Funktionen in jeweils getrennte mmu_*-Instanzen.

[7] Die einfachste Lösung hierfür besteht darin, dass der tmp-Eingang bereits bei der „Ur-Instantiierung" das richtige Prozessabbild mit der Baustein-Konfi guration enthält, mit dem zu starten ist. Wenn dieses auch die steuernde control-Instanz enthält, entsteht automatisch die Selbstverwaltung. Bei der Systemgenerierung eines bootfähigen Systems wird auf einem (fremden) Rechner ein entsprechendes Prozessabbild mit allen zum Urstart notwendigen Instanzen erstellt; dieses kann dann mit konventionellen Ladern analog wie klassische Unix-Kerne geladen werden.

geschehen, z.B. zeitgesteuert in selbsttätig optimierenden Betriebssystemen (vgl. [BR76]) oder zur Erzielung von Fehlertoleranz und Ausfallsicherheit.

Der Fall 1 läßt sich entweder direkt in control abhandeln, oder durch einen eigenen Baustein-Typ registrar, der als *Registrar* für verschiedene Datenformate dient. Datenformate sollten vorzugsweise im zugehörigen Meta-Nest beschrieben werden; es gibt aber auch auch Sonderfälle wie Platten-Partitionen oder dir_*-Bausteine für konventionelle Archiv-Datei Formate (wie .tar oder .zip), bei denen die Format-Erkennung durch Inspektion des Inhaltes des betreffenden Nestes erfolgen muss. Ein Registrar verwaltet daher auch die Methoden für die automatische Format-Erkennung.



Um eine möglichst hohe Flexibilität bei verschiedenen Auslösern von Instantiierungen zu erreichen, schlage ich die Einrichtung von strategy_*-Bausteinen vor. Diese steuern den control- oder registrar-Baustein und erhalten von dort Informationen über vorhandene Datenformate und Kompetenzen; sie werden ihrerseits von anderen Steuer-Instanzen wie z.B. Benutzer-Operationen oder andere strategy_*-Instanzen gesteuert. Im Bild sind die Steuerleitungen gestrichelt gezeichnet.

Ein Beispiel wäre ein Baustein strategy_asciipath, der Pfadnamen akzeptiert und indirekt über registrar bzw. control auf einen Verzeichnisbaum von dir_*-Instanzen abbildet. Um ständig wiederholende Instantiierungen / De-Instantiierungen von Bausteinen zu verhindern, kann strategy_cache einen „Vorrat“ von häufig benutzten Instanzen anlegen, die bei Nichtbenutzen evtl. lediglich auf „statuslos“ geschaltet werden. Die Funktionalität des klassischen fork-Systemaufrufs von Unix lässt sich ebenfalls durch strategy_*-Bausteine, diejenige von /proc-Dateisytemen durch spezialisierte Arten von Registraren herstellen, die für automatische Default-Instantiierungen registrierter Komponenten sorgen.

Das Problem von inkompatiblen Kompetenzen und Verhalten von versuchten Baustein-Verdrahtungen bzw. Konstruktionen läßt sich durch einen zwischengeschalteten registrar_intermediate lösen. Dieser kennt alle möglichen adaptor_* und sonstigen Konversions-Baustein-Typen samt ihren Eigenschaften und veranlasst bei Bedarf die automatische Zwischenschaltung der geeigneten Komponenten. Auf ähnliche Weise läßt sich Netzwerk-Transparenz durch automatisierte Zwischenschaltung von remote erzielen.

Bei Implementierung geeigneter Strategien (vgl. z.B. [LS94]) lässt sich Fehlertoleranz durch automatisches Re-Instantiieren von ausgefallenen Bausteinen erzielen, oder in Kombination mit der Netzwerk-Transparenz zur dynamischen Lastbalancierung nutzen. Es sind unzählige weitere Anwendungen für strategy_* und registrar_* denkbar.

7 Behandlung nichtfunktionaler Eigenschaften

Durch die Verdrahtung von Bausteinen mittels Leitungen und die Einfachheit der Nest-Schnittstelle ist bereits eine uniforme Infrastruktur geschaffen worden, die das Aushandeln von dynamischen Eigenschaften und Parametern bei der Instantiierung von Bausteinen erheblich vereinfacht.

Die in Abschnitt 2.4 eingeführten Begriffe der Kompetenz, des Verhaltens und der Kompatibilität zwischen Kompetenz und Verhalten eignen sich prinzipiell nicht nur zur Sicherstellung funktionaler Verdrahtungen, sondern auch zum Aushandeln nicht-funktionaler Eigenschaften wie Dienst-Qualitäten etc. Attribute und Kompatibilitäts-Prüfungen stellen einen generischen Mechanismus zur Spezifikation jeglicher Art von Eigenschaften dar.

Eine möglichst universelle Attribut-Darstellung könnte beispielsweise folgendermaßen aussehen: Es werden Listen von Quadrupeln (name,typ,prio,wert) erzeugt. Alle vorkommenden Attribut-Namen name müssen paarweise disjunkt sein. Der typ läßt sich als Wert aus einem abstrakten Meta-Datentyp auffassen, dessen Bedeutung dem Verdrahtungs-Mechanismus nicht bekannt zu sein braucht (es ist jedoch zweckmässig, vor-reservierte Typ-Werte wie boolean oder integer mit bekannter Bedeutung zu verwenden). Die Priorität prio ist konzeptionell eine reelle Zahl zwischen 0.0 und 1.0 (oder eine angenäherte Ersatzdarstellung mittels Ganzzahlen), wobei 0.0 die „absolute Bedeutungslosigkeit“ und 1.0 die „unbedingte Notwendigkeit“ des Zusammenpassens zwischen Kompetenz-Attributen und Verhaltens-Attributen ausdrückt. Zwischenwerte zwischen 0.0 und 1.0 dienen zum Ausdruck wünschenswerter, aber nicht absolut notwendiger Eigenschaften; bei Zielkonflikten zwischen mehreren Parametrisierungs-Varianten lassen sich unterschiedlich gewichtete Prioritäten zur Bestimmung eines „Optimums“ verwenden (z.B. Simplex-Methode zur Lösung linearer Optimierungs-Aufgaben). Der Wert wert ist ein abstraktes Objekt, dessen Interpretation von einer Vergleichs-Funktion is_compatible übernommen werden kann, die die Kompatibilität zweier Quadrupel-Listen testet und als Ergebnis einen „Kompatibilitäts-Grad“ (bzw. eine „Gesamt-Qualität“) zwischen 0.0 und 1.0 zurückliefert (im Regelfall ist dies das Minimum der Qualität aller Quadrupel gleichen Namens und gleichen Typs). Im Falle des Typs integer sind beispielsweise nicht nur einfache Zahlen-Werte, sondern auch Intervalle als wert repräsentierbar, die auf Überschneidung getestet werden können.

Die Idee besteht nun darin, keine feste Vergleichsfunktion is_compatible zu verwenden, sondern jedem Eingang eines Bausteins eine eigene Implementierung dieser Funktion zuzuordnen. Damit kann jeder Baustein-Typ vor der Konstruktion selbst entscheiden, welche eigenen Verhaltens-Attribute er mit welchen Typen und mit welchen

Methoden gegenüber den fremden Kompetenz-Attributen testen möchte. Weiterhin kann `is_compatible` als Seiteneffekt ggf. die „optimalen“ Parameter zur Parametrisierung des eigenen Bausteins bestimmen.

Damit haben wir einen universellen generischen Mechanismus zur Spezifikation beliebiger Eigenschaften, dessen Semantik frei wählbar ist, wobei diese Wahl zu den zu implementierenden Strategien gehört. Eine detaillierte Diskussion der Wahlmöglichkeiten würde dieses Papier sprengen.

Man sollte sich vor Augen halten, dass dieser universelle Mechanismus eine Optimierung von Parametern nur in Richtung von vorgeschalteten Bausteinen zu nachgeschalteten ermöglicht (normalerweise werden Bausteine nur in dieser Reihenfolge konstruiert). Möchte man wechselseitiges Aushandeln von Parametern in beide Richtungen ermöglichen, kann dazu auch den Ausgängen der Lieferanten eine Version von `is_compatible` zugeordnet werden. Unter Umständen sind nachträgliche Re-Parametrisierungen bereits konstruierter Ausgänge wegen später hinzukommender Verdrahtungen zu parallel nachgeschalteten Bausteinen notwendig. Die nachträgliche Änderung bereits eingestellter Parameter würde die bislang während der Lebensdauer einer Baustein-Instanz als konstant betrachteten Attribute zu einem Teil ihres Zustandes machen. Da mir dies aus theoretischen Erwägungen nicht sonderlich glücklich erscheint, könnte eine passende Einschränkung darin bestehen, dass Re-Parametrisierungen nur nach dem (temporären) Schalten auf Statuslosigkeit erlaubt werden. Dies kann jedoch zu Betriebs-Unterbrechungen führen. Ein weiteres Problem besteht in der möglichen Verschlechterung bereits ausgehandelter Dienstqualitäten durch neu hinzugekommene „Konkurrenten“. Daher scheint es mir sinnvoll, einmal ausgehandelte Parameter nicht ohne Not anzurühren.

Das Anstreben lokaler Optima an jeder Verdrahtungs-Leitung muss i.a. nicht unbedingt zu einem globalen Optimum aller Parameter eines komplexen Systems führen. Um globale Optima anzustreben, sehe ich prinzipiell zwei Möglichkeiten, die ich hiermit zur Diskussion stellen möchte:

a) Verwendung einfacher leicht berechenbarer Modelle auch auf globaler Ebene, beispielsweise lineare Optimierung (hierzu ist eine zentrale Kontroll-Instanz notwendig)

b) Fixpunkt-Iteration auf dem Graphen der Verdrahtungs-Leitungen bis zu einem pragmatischen Abbruch-Kriterium. Das Erreichen eines globalen Optimums ist damit zwar i.a. nicht garantiert, dafür können auch komplizierte nicht-lineare Probleme mit vertretbarem Aufwand behandelt werden.

8 Abschlussbemerkungen

Es wurde eine einfache, aber trotzdem sehr mächtige Betriebssystem-Architektur vorgestellt, die mit sehr wenigen Abstraktionen und Operationen auskommt. Bausteine dienen als universelle Transformatoren zwischen verschiedenen Nest-Instanzen. Ein Teil der Funktionalität konventioneller Betriebssysteme wird durch die Kombination von Bausteinen zu komplexen Netzwerken erzeugt; dadurch ergeben sich Transformationsmöglichkeiten und Funktionalitäten, die mit herkömmlichen Betriebssystem-Architekturen nur schwer realisierbar sind. Die zugrundeliegenden Entwurfs-Prinzipien der universellen und kompositorischen Generizität wurden gegenüber der Objektorientierung abgegrenzt.

Die Integration einer universellen Transaktions-Funktionalität in die Nest-Schnittstelle ist in Form eines weiteren Zugriffs-Modells möglich. Dadurch wird Transaktionsfähigkeit zu einem alles durchdringenden Prinzip.

Nichtfunktionale Eigenschaften von Betriebssystemen sehe ich ähnlich dazu als etwas, das die gesamte Infrastruktur durchdringen können sollte. Die Behandlung sollte auf generische Weise erfolgen. Zur Realisierung von verschiedenen Generizitäts-Arten wurden Mechanismen vorgestellt.

Danksagung

Beim Korrekturlesen haben mir Klaus Lagally, Nicole Ondrusch und Jörgen Bertele geholfen. Für Verbesserungsvorschläge zur kompakteren Darstellung möchte ich mich bei Nicole Ondrusch besonders bedanken.

Literatur

[AB86] ARCHIBALD, JAMES und JEAN-LOUP BAER: *Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model.* Transactions on Computer Systems, 4(4):273–298, 1986.

[Ant90] ANTONOV, VADIM G.: *A Regular Architecture for Operating Systems.* Operating System Reviews, 24(3):22–39, 1990.

[Ass96] ASSENMACHER, HOLGER: *Ein Architekturkonzept zum Entwurf flexibler Betriebssysteme.* Dissertation Universität Kaiserslautern, 1996.

[B+90] BERSHAD, BRIAN N. und OTHERS: *Lightweight Remote Procedure Call.* Transactions on Computer Systems, 8(1):37–5, 1990.

[B+95] BERSHAD, BRIAN N. und OTHERS: *SPIN – An Extensible Microkernel for Application-specific Operating System Services.* Operating System Reviews, 29(1):74–77, 1995.

[BN84] BIRRELL, ANDREW D. und BRUCE JAY NELSON: *Implementing Remote Procedure Calls.* Transactions on Computer Systems, 2(1):39–56, 1984.

[BPS81] BELADY, L. A., R. P. PARMELEE und C. A. SCALZI: *The IBM History of Memory Management Technology.* IBM Journal of Research and Development, 25(5):491–503, 1981.

[BR76] BLEVINS, PARKER R. und C. V. RAMAMOORTHY: *Aspects of a Dynamically Adaptive Operating System.* Transactions on Computers, 25(7):713–725, 1976.

[BS75] BERNSTEIN, ARTHUR J. und PAUL SIEGEL: *A Computer Architecture for Level Structured Systems.* Transactions on Computers, 24(8):785–893, 1975.

[C+94] CHASE, JEFFREY S. und OTHERS: *Sharing and Protection in a Single-Address-Space Operating System.* Transactions on Computer Systems, 12(4):271–307, 1994.

[Che87] CHERITON, DAVID R.: *UIO: A Uniform I/O System Interface for Distributed Systems.* Transactions on Computer Systems, 5(1):12–46, 1987.

[CJ75] COHEN, ELLIS und DAVID JEFFERSON: *Protection in the Hydra Operating System.* Symposium on Operating System Principles, Seiten 141–160, 1975.

[CW85] CARDELLI, LUCA und PETER WEGNER: *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys, 17(4):471–522, 1985.

[DH66] DENNIS, JACK B. und EARL C. VAN HORN: *Programming Semantics for Multiprogrammed Computations*. CACM, 9(3):143–155, 1966.

[Dij68] DIJKSTRA, EDSGER W.: *The Structure of the “THE” Multiprogramming System*. CACM, 11(5):341–346, 1968.

[Dij71] DIJKSTRA, E. W.: *Hierarchical Ordering of Sequential Processes*. Acta Informatica, 1:115–138, 1971.

[Doe96] DOEPPNER, THOMAS W.: *Distributed File Systems and Distributed Memory*. Computing Surveys, 28(1):229–231, 1996.

[EKO95] ENGLER, DAWSON R., M. FRANS KAASHOEK und JAMES O'TOOLE: *Exokernel: An Operating System Architecture for Application-Level Resource Management*. Symposium on Operating System Principles, Seiten 251–266, 1995.

[Esk96] ESKICIOGLU, M. RASIT: *A Comprehensive Bibliography of Distributed Shared Memory*. Operating System Reviews, 30(1):71–96, 1996.

[Fie88] FIELD, ANTHONY J.: *Functional Programming*. Addison-Wesley, 1988.

[Fot61] FOTHERINGHAM, JOHN: *Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of Backing Store*. CACM, 4(10):435–436, 1961.

[Han70] HANSEN, PER BRINCH: *The Nucleus of a Multiprogramming System*. CACM, 13(4):238–250, 1970.

[Han73] HANSEN, PER BRINCH: *Concurrent Programming Concepts*. Computing Surveys, 5(4):223–245, 1973.

[Hoa74] HOARE, C. A. R.: *Monitors: An Operating System Structuring Concept*. CACM, 17(10):549–557, 1974.

[Hoa78] HOARE, C. A. R.: *Communicating Sequential Processes*. CACM, 21(8):666–677, 1978.

[HP94] HEIDEMANN, JOHN S. und GERALD J. POPEK: *File-System Development with Stackable Layers*. Transactions on Computer Systems, 12(1):58–89, 1994.

[HP95] HEIDEMANN, JOHN und GERALD POPEK: *Performance of Cache Coherence in Stackable Filing*. Symposium on Operating System Principles, Seiten 127–142, 1995.

[HR73] HORNING, J. J. und B. RANDELL: *Process Structuring*. Computing Surveys, 5(1):5–30, 1973.

[Hur] *The GNU Hurd – GNU Project – Free Software Foundation (FSF)*. `http://www.gnu.org/software/hurd/hurd.html`.

[Jon80] JONES, ANITA K.: *Capability Architecture Revisited*. Operating System Reviews, 14(3):33–35, 1980.

[K+81] KAHN, KEVIN C. und OTHERS: *iMAX: A Multiprocessor Operating System for an Object-Based Computer*. Symposium on Operating System Principles, Seiten 127–136, 1981.

[K+97] KAASHOEK, M. FRANS und OTHERS: *Application Performance and Flexibility on Exokernel Systems*. Symposium on Operating System Principles, Seiten 52–65, 1997.

[Lag75] LAGALLY, KLAUS: *Das Projekt Betriebssystem BSM*. Technischer Bericht LRZ-Bericht 7502/1, gleichzeitig TUM-Bericht 7509, Leibnitz-Rechenzentrum München, 1975.

[Lag78] LAGALLY, K.: *Synchronization in a Layered System*. In: FLYNN, M. J. und OTHERS (Herausgeber): *Operating Systems, An Advanced Course*, Band 60 der Reihe *Lecture Notes in Computer Science*, Seiten 253–281. Springer-Verlag, 1978.

[Lan81] LANDWEHR, CARL E.: *Formal Models for Computer Security*. Computing Surveys, 13(3):247–278, 1981.

[LCC+75] LEVIN, R., E. COHEN, W. CORWIN, F. POLLACK und W. WULF: *Policy/Mechanism Separation in Hydra*. Symposium on Operating System Principles, Seiten 132–140, 1975.

[LH89] LI, KAI und PAUL HUDAK: *Memory Coherence in Shared Virtual Memory Systems*. Transactions on Computer Systems, 7(4):321–359, 1989.

[Lie95a] LIEDTKE, JOCHEN: *Address Space Sparsity and Fine Granularity*. Operating System Reviews, 29(1):87–90, 1995.

[Lie95b] LIEDTKE, JOCHEN: *On μ-Kernel-Construction*. Symposium on Operating System Principles, Seiten 237–250, 1995.

[LS94] LIN, TEIN-HSIANG und KANG G. SHIN: *An Optimal Retry Policy Based on Fault Classification*. Transactions on Computers, 43(9):1014–1025, 1994.

[Mey88] MEYER, BETRAND: *Objektorientierte Software-Entwicklung*. Prentice Hall, 1988.

[MP81] MILLER, BARTON und DAVID PRESOTTO: *XOS: An Operating System for the X-Tree Architecture*. Operating System Reviews, 15(2):21–32, 1981.

[NW74] NEEDHAM, R. M. und M. V. WILKES: *Domains of protection and the management of processes*. Computer Journal, 17(2):117–120, 1974.

[NW77] NEEDHAM, R. M. und R. D. H. WALKER: *The Cambridge CAP Computer and its protection system*. Symposium on Operating System Principles, Seiten 1–10, 1977.

[Par72] PARNAS, D. L.: *On the Criteria To Be Used in Decomposing Systems into Modules*. CACM, 15(12):1053–1058, 1972.

[Par78] PARNAS, DAVID L.: *The Non-Problem of Nested Monitor Calls*. Operating System Reviews, 12(1):12–18, 1978.

[PC75] PRUITT, J. L. und W. W. CASE: *Architecture of a Real Time Operating System*. Symposium on Operating System Principles, Seiten 51–59, 1975.

[Ros94] ROSCOE, TIMOTHY: *Linkage in the Memesis Single Address Space Operating System*. Operating System Reviews, 28(4):48–55, 1994.

[RT74] RITCHIE, DENNIS M. und KEN THOMPSON: *The UNIX Time-Sharing System*. CACM, 17(7):365–375, 1974.

[ST02] SCHÖBEL-THEUER, THOMAS: *Eine neue Architektur für Betriebssysteme*. Manuskript, 2002.

[Str78] STROUSTRUP, BJARNE: *On Unifying Module Interfaces*. Operating System Reviews, 12(1):90–98, 1978.

[Szy98] SZYPERSKI, CLEMENS: *Component Software*. Addison-Wesley, 1998.

[T+90] TANENBAUM, ANDREW S. und OTHERS: *Experiences with the Amoeba Distributed Operating System*. CACM, 33(12):47–63, 1990.

[TA90] TAY, B. H. und A. L. ANANDA: *A Survey of Remote Procedure Calls*. Operating System Reviews, 24(3):68–79, 1990.

[Tho96] THOMPSON, SIMON: *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[TSF90] TAM, MING-CHIT, JONATHAN M. SMITH und DAVID J. FARBER: *A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems*. Operating System Reviews, 24(3):40–67, 1990.

[Y+90] YOKOTE, YASUHIKO und OTHERS: *The Muse Object Architecture: A New Operating Systems Structuring Concept*. Operating System Reviews, 25(2):22–46, 1990.