Spring

- ❑ Überblick, Motivation & Ziele
- ❑ Anwendungssicht, Systemarchitektur
- ❑ Programmierung, Schnittstellen
- ❑ Namensraum
- ❑ Schutzmechanismen
- ❑ Kern-Architektur
  - Objektinteraktionen
  - Threads

Literatur

HaK93 Hamilton, G ; Kougiouris, P: The Spring Nucleus: A microkernel for objects. In: *Proceedings of the 1993 Summer USENIX Conf. (San Diego, CA 1993)*. Berkeley : USENIX Ass., 1993, S. 147 - 159

KhN93 Khalidi, Y. A.. ; Nelson, M. N.: An implementation of UNIX on an object-oriented operating system. In: *Proceedings of the 1993 Winter USENIX Conf. (Cincinnati, Ohio 1993)*. Berkeley : USENIX Ass., 1993, S. 469 - 480

MGH+94 Mitchell, J. ; Gibbons, J. ; Hamilton, G. et.al.: An Overview of the Spring System. In: *Proc. of the Compcon Spring 1994 (San Francisco)*. Los Alamitos : IEEE. 1994

Diese und weitere Literatur ist im Handapparat zu OOBP in der Gruppenbibliothek Informatik zu finden:
Publikationssammlung "A Spring Collection" von SunSoft
Ordner mit Schulungsunterlagen zu Spring, Auszügen aus WWW-Seiten zur Programmierung mit Spring und Reference Manuals zum Themenkomplex *contract* und *IDL*

Überblick

- ❑ Forschungs-Betriebssystem von SunSoft

  Spring war eine neue Betriebssystementwicklung. Eine Reihe von Architekturmerkmalen sind zwar von älteren Betriebssystemkonzepten beeinflußt — es wurde aber großer Wert darauf gelegt, nicht noch eine neue Implementierung traditioneller Betriebssystemkonzepte vorzunehmen, sondern Betriebssystemkonzepte für die Anforderungen moderner Architekturen zu entwickeln und zu implementieren.

  - verteilt

    Spring stellt Betriebssystemkonzepte für ein verteiltes System bereit. Während bislang Betriebssysteme für einzelne Rechner ausgelegt waren und "zusätzlich" auch Mechanismen zur Interaktion zwischen diesen Rechnern bereitstellten, wird hier das verteilte System (in der Praxis ein Cluster von Rechnern) zur Hardwareplattform des (wiederum verteilten) Betriebssystems. Im Bereich der Forschungsbetriebssysteme ist dieser Trend seit längerem zu sehen, z.B. im PEACE-Projekt der GMD (URL: http://www.first.gmd.de/peace/peace.html) oder in Clouds (siehe Abschnitt 5 dieser Vorlesung)

  - objektorientiert

    Spring ist ein objektorientiert aufgebautes Betriebssystem für objektorientierte Anwendungen. Sun hat viele Erfahrungen aus der Spring-Entwicklung in seine Aktivitäten in der Object-Management-Group (OMG) eingebracht (z.B. die IDL und Sicherheitskonzepte).

- ❑ Ziele
  - Flexibilität, Modularität
  - schnelle ortstransparente Objektinteraktionen
  - verbesserte Sicherheit
  - erhöhte Verfügbarkeit

  Diese Ziele reflektieren die derzeit größten Probleme bei der Weiterentwicklung, Wartung und Nutzung existierender Betriebssysteme.

- ❑ Entwicklung von 1989 bis 1995 bei Sun Microsystems Lab. und SunSoft (am Ende 30 Mitarbeiter)
- ❑ Produktpolitik

  Sun verfolgte zunächst nicht unmittelbar das Ziel, aus Spring das Produktionsbetriebssystem der Zukunft zu machen. Vielmehr sollen mit Spring Erfahrungen auf dem Gebiet verteilter, objektorientierter Betriebssysteme gesammelt werden und darüber die weitere Entwicklung der Produktlinie (Solaris) beeinflußt werden. 1995 ging man mit Spring an die Öffentlichkeit und vergab eine Reihe von Beta-Lizenzen . Die Resonanz war sehr gut - die Anzahl der Fehlermeldungen entsprechend hoch. Es stellte sich bald die Frage, ob man es sich bei SunSoft leisten kann, zwei Betriebssysteme zu pflegen - ohne professionelle Unterstützung wäre die Weiterverbreitung von Spring aber nicht sinnvoll gewesen. Zu diesem Zeitpunkt wurde außerdem absehbar, daß in die Weiterentwicklung von Java großer Personalaufwand zu investieren sein. Die Kompromißentscheidung, die Spring-Entwicklung in die Solaris Produktentwicklung zu integrieren mußte scheitern, da die Solaris-Entwickler nicht bereit waren, die mühsam erzielte Stabilität ihres Systems auf's Spiel zu setzen. Ein großer Teil der Spring-Entwickler wechselte im Herbst 1995 zu JavaSoft und arbeitete dort dann an der Entwicklung von JavaOS.

Motivation

bekannte Probleme mit traditionellen Betriebssystemen:

Die heute verbreiteten Betriebssysteme stammen alle aus einer Zeit, in der Betriebssysteme für einzelne Monoprozessor-Rechner Ziel der Entwicklung waren. Dadurch sind grundsätzliche Anforderungen, die sich durch moderne, parallele und verteilte Architekturen und darauf zugeschnittene Anwendungen ergeben, nicht berücksichtigt. Zwar wurden vielfach Anpassungen und Erweiterungen in diese Richtung vorgenommen, diese stoßen aber immer wieder an Grenzen, die durch die ursprüngliche Konzeption bedingt sind.

❑ Weiterentwicklung wird zunehmend schwieriger

- verteiltes Rechnen ursprünglich nicht berücksichtigt

  Zwar enthalten alle Betriebssysteme heute Mechanismen zur Rechnerkommunikation. Diese sind aber nachträglich hinzugefügt und nicht vollständig integriert. Beispielsweise können in UNIX zwar Prozesse über RPC über Rechnergrenzen miteinander kommunizieren und es ist möglich über NFS auf Dateien anderer Rechner zuzugreifen, es ist aber nicht ohne weiteres möglich, Signale an Prozesse auf einem anderen Rechner so wie an lokale Prozesse zuzustellen oder über NFS auch auf Gerätedateien wie auf lokale Gerätedateien zuzugreifen. Der Unterschied zwischen lokalem Rechnen und Operationen im verteilten System tritt immer wieder hervor.

- Parallelität (multithreading und multprocessing) ursprünglich nicht berücksichtigt

  Traditionelle Betriebssysteme bauen auf den Basisabstraktionen Prozeß und Datei auf. Der Prozeß ist dabei die Kombination eines Aktivitätsträgers mit den Ressourcen für dessen Ausführung (Adreßraum, Speichersegmente, Rechte, etc.). Die parallele Ausführung mehrerer Aktivitätsträger in einem Adreßraum ist dabei von vorneherein nicht vorgesehen. Die Aufspaltung des Prozeßkonzepts in Tasks/Domains und Threads ist vor allem seit MACH bekannt. Threadkonzepte werden mehr und mehr auch in traditionelle Betriebssysteme integriert. Solche Erweiterungen haben aber immer zur Folge, daß entweder große Teile des Betriebssystems überdacht werden müssen (vor allem das Scheduling) oder die Integration unvollständig bleibt und dadurch neue Probleme aufgeworfen werden (z.B. die Blockierung des gesamten Prozesses mit allen Threads, wenn ein Thread einen blockierenden Systemaufruf ausführt).

- Ausrichtung auf ursprüngliche Zielarchitektur setzt Grenzen

  Moderne Architekturen (RISC-Prozessoren mit umfangreichen Register-Files, komplexe Cache-Technologien, 64-Bit-Adreßräume, …) und neue Softwaretechnologien (objektorientierte Anwendungen, persistente Objekte, …) stellen viele neue Möglichkeiten zur Verfügung, die vom Betriebssystem möglichst gut unterstützt werden sollten. Da bei der Entwicklung älterer Betriebssysteme völlig andere Voraussetzungen vorlagen, ist es häufig nur möglich, Kompromisse zu realisieren — Anforderungen moderner Anwendungen müssen auf die Fähigkeiten des alten Betriebssystems abgebildet werden und die Fähigkeiten moderner Architekturen bleiben ungenutzt, weil sie das Betriebssystem nicht umsetzen kann.

❑ Schutzmechanismen sind vielfach ungenügend

Gutes Beispiel hierfür ist UNIX mit sehr eingeschränkten Möglichkeiten zur Formulierung von Zugriffsrechten.

❑ monolitisch, groß und schwer zu pflegen

Ziele

❑ Anforderungen an Betriebssysteme heute:

modular offen erweiterbar

Traditionelle Betriebssysteme sind in aller Regel als *general purpose* Betriebssysteme ausgelegt. Daneben gibt es natürlich auch für bestimmte Aufgaben spezialisierte Betriebssysteme — vor allem im Bereich Prozeßautomatisierung sind solche Systeme sehr weit verbreitet. Der Nachteil spezialisierter Betriebssysteme besteht vor allem darin, daß es relativ aufwendig ist, für eine dedizierte Aufgabe solch ein Betriebssystem zu erstellen und daß der damit ausgestattete Rechner dann nicht mehr parallel für andere Aufgaben genutzt werden kann. *General purpose* Betriebssysteme dagegen bieten als Betriebssystemdienste immer nur Kompromißlösungen an — für einige Anwendungen sind die Mechanismen unzureichend, für andere überdimensioniert und damit unnötig teuer.

Ein modularer Aufbau eines Betriebssystems vereinfacht die Anpassung an spezielle Anforderungen. Eine offene Architektur stellt Schnittstellen bereit, über die unterschiedliche Ausprägungen von Betriebssystemmechanismen (auch zur Laufzeit) für unterschiedliche Anwendungen bereitgestellt werden können. Erweiterbarkeit bedeutet in diesem Sinne, daß Betriebssystemmechanismen anwendungsspezifisch erweitert werden können. Solche Erweiterungen können dynamisch erfolgen, wenn sie benötigt werden, und sie können so erfolgen, daß sie nur für die tangierten Anwendungen wirksam sind, ohne andere Anwendungen zu beeinträchtigen.

❑ Einsatz moderner Softwaretechniken:

➔ höhere Qualität & bessere Modularität

Der Einsatz objektorientierter Techniken bei der Softwareentwicklung zusammen mit einer strengen Schnittstellenbildung fördert eine saubere Modularität und damit die Qualität.

❑ bessere Basis für Schutzmechanismen

Im Mittelpunkt der Spring-Schutzmechanismen steht ein geschützter Zugriff auf Objekte. Basis hierfür sind Access Control Listen und Software Capabilities. Diese Basismechanismen ermöglichen eine sehr große Flexibilität bei der Implementierung konkreter Schutzmechanismen (im Gegensatz beispielsweise zu der UNIX-Dateizugriffskontrolle). Es können unterschiedliche Implementierungen dieser Mechanismen in einem Spring-System existieren — im Sinne einer anwendungsspezifischen Anpaßbarkeit.

❑ Interoperabilität zu existierenden Systemen gewährleisten

Damit ein neues Betriebssystem überhaupt akzeptiert wird, muß eine volle Interoperabilität zu den existierenden Installationen (vor allem auf UNIX-Basis) gewährleistet sein. Da die Netzwerkschnittstellen traditioneller Systeme sehr eingeschränkt sind (TCP/IP als Basisprotokoll, Sun-RPC zur Programminteraktion und NFS für Dateisystemzugriff), ist dies allerdings relativ einfach zu gewährleisten.

❑ neue Anwendungsanforderungen:

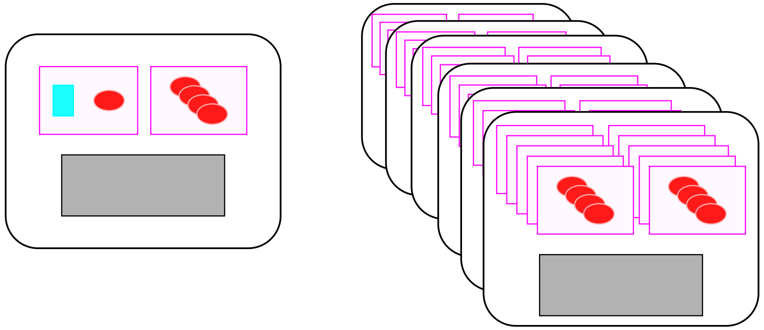
➔ nicht noch eine Neuimplementierung traditioneller Techniken!

Im Mittelpunkt der Spring-Entwicklung standen die Anforderungen verteilter, objektorientierter Anwendungen. Das System ist primär auf dieses Ziel ausgerichtet. Der Schwerpunkt der Basisabstraktionen liegt damit völlig anders als in traditionellen Betriebssystemen (effiziente Interprozeßkommunikation statt Dateien und Prozessen). Traditionelle Konzepte können auf diesen neuen Abstraktionen aufgebaut werden. Spring unterscheidet sich damit essentiell von Systemen wie MACH oder Chorus.

Anwendungssicht

Anything Can Be Anywhere

- kein Unterschied zwischen verteilter und nicht-verteilter Programmierung



Ein Grundprinzip von Spring ist, nicht nur Unterstützung für die Programmierung von verteilten, objektorientierten Anwendungen bereitszustellen, sondern das verteilte System völlig vor der Anwendung zu verbergen. Es soll kein Unterschied zwischen verteilter und nicht-verteilter Programmierung bestehen.

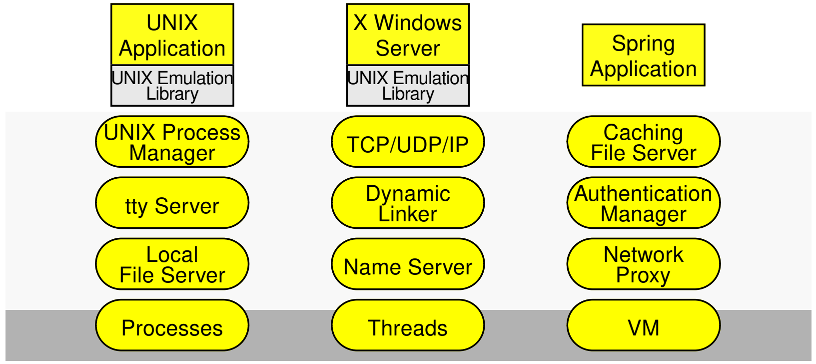
Für den Anwendungsprogrammierer besteht tatsächlich kein Unterschied, ob ein Objekt eine Methode eines anderen Objekts innerhalb des gleichen Adreßraums, in einem anderen Adreßraum auf dem gleichen Rechner oder auf einem anderen Rechner aufruft.

Systemarchitektur

- ❑ Minimalkern-basiert

  Wie bei Chorus und MACH baut Spring auf einem Minimalkern (die dunkelgraue Schicht in der Abbildung) auf.

- ❑ Entwicklungsschwerpunkt: BS-Module auf Anwendungsebene



Im Gegensatz zu Chorus und MACH wurde bei Spring der Schwerpunkt auf die Komponenten außerhalb des Minimalkerns gelegt. Bei Chorus und MACH ist das im wesentlichen ein UNIX-Emulator, der weitgehend aus einem existierenden UNIX-Kern abgeleitet ist. Es ist nicht das primäre Ziel von Spring, binärkompatibel zu existierenden Betriebssystemen zu sein, indem Server mit der Funktionalität dieser Systeme auf dem Minimalkern bereitgestellt werden.

Schwerpunkt der Spring-Entwicklung sind Komponenten außerhalb des Minimalkerns, die Mechanismen bereitstellen um die Spring-Ziele zu erreichen: Ablauf von verteilten, objektorientierten Anwendungen; flexible, erweiterbare Betriebssystemdienste für solch ein Umfeld. Beispiele:

- *Network Proxies* stellen die Mechanismen für rechnerübergreifende Methodenaufrufe zur Verfügung
- *Name Server* implementieren die Namensräume für Objekte und ggf. auch Kontrolle über Capability-Vergabe
- *Authentication Manager* überprüfen Access Control Listen beim Zugriff auf Objekte
- *File Server* implementieren Dateisysteme auf Disk

Die Anforderung von Diensten vom Minimalkern und solchen Server-Komponenten erfolgt über Methodenaufrufe. Damit sind solche Dienste im verteilten System ortsunabhängig erreichbar (da Methodenaufrufe ja ortstransparent sind).

Neben Komponenten, die neue Betriebssystemkonzepte realisieren ist es wichtig, Kompatibilität zu existierenden Systemen (vor allem UNIX) zu erhalten, um die Vielzahl der dort verfügbaren Werkzeuge und Anwendungen weiter nutzen zu können. In der Anfangsphase ist für Spring selbst ja nur wenig neue Software verfügbar. Durch eine (dynamisch hinzugebundene) Emulations-Bibliothek und einige Server-Komponenten (außerhalb des Minimalkerns!) werden daher UNIX(d.h. Solaris)-Anwendungen weitgehend unterstützt.

Programmierung & Schnittstellen

- ❑ Trennung der Konzepte von

  Typ ➔ ***Contract*** ➔ **IDL (CORBA)**

  und **Klasse** ➔ ***Implementation*** ➔ **C++ (o. a.)**

  Bei der Entwicklung von modularen, leicht erweiterbaren Systemen sind die Schnittstellen zwischen den Modulen von besonderer Bedeutung. Im Gegensatz zu vielen objektorientierten Sprachen wird in Spring die Schnittstelle eines Objekts (der Typ) getrennt von der Implementierung (der Klasse) beschrieben.

  Typbeschreibungen können von existierenden Typen erben (Subtyping), und es können mehrere verschiedene Implementierungen für einen Typ bereitgestellt werden.

  Die Implementierung kann in einer beliebigen objektorientierten Programmiersprache erfolgen. Aktuell wird allerdings nur C++ unterstützt.

- ❑ **Dynamische Typprüfung** durch Spring

  ➔ mehr Typanpassungsmöglichkeiten als auf Sprachebene

  Da das Betriebssystem (nicht der Kern, sondern BS-Komponenten außerhalb des Kerns!) die Methodenaufrufe abwickelt und hierbei Typinformationen vorliegen, ist eine dynamische Typprüfung möglich. Damit ist nicht nur Typanpassung im Sinne eines *inclusion polymorphism* (Verwendung eines Subtyps als Basistyp — *widening*) möglich, sondern es ist auch möglich, umgekehrt einen Basistyp in einen Subtyp umzuwandeln (*narrowing*), wenn zur Laufzeit erkannt wird, daß eine Basistyp-Referenz tatsächlich auf eine Subtyp-Instanz verweist (neuere C++-Laufzeitsysteme unterstützen dies ebenfalls!).

- ❑ IDL-Compiler erzeugt:
  - Client- und Server-Stubs
  - Objekt-Skelette

  Die Schnittstellenbeschreibung hat nicht nur den Vorteil, daß eine Trennung von Typ und Klasse erreicht wird. Sie wird außerdem benötigt, um den Stub-Code zu erzeugen, der für die Abwicklung der ortstransparenten Methodenaufrufe benötigt wird — es muß hierbei ja bekannt sein, welche Methoden es gibt und wie diese parametriert sind. Die Stub-Funktionen werden von einem IDL-Compiler erzeugt und müssen zu der Anwendung hinzugebunden werden.

  Da durch eine Schnittstellenbeschreibung alle Informationen für die Deklarationen der entsprechenden Klasse vorliegen, wird in Spring außerdem ein Werkzeug angeboten, das ein Skelett für die Implementierung einer entsprechenden Klasse erzeugt. Dieses Skelett enthält die komplette Klassendeklaration und die Methodenköpfe. Es muß lediglich mit der Methodenimplementierung gefüllt werden.

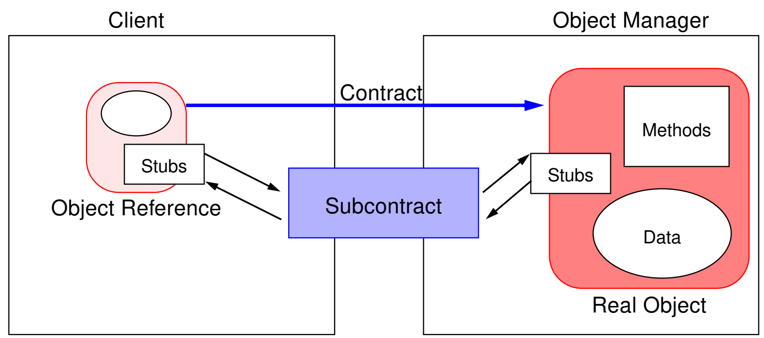
- ❑ **Subcontract**
  - Offene Implementierung der Methodenaufruf-Abwicklung

  Die Abwicklung eines Methodenaufrufs erfolgt im sog. *Subcontract*. Darunter ist der Code zu verstehen, der aus den Stub-Funktionen angesprungen wird. Im Subcontract wird zuerst das Marshalling der Parameter entsprechend einer Beschreibung, die vom IDL-Compiler erzeugt wurde, durchgeführt. Anschließend wird (bei Bedarf) mit Hilfe eines Systemaufrufs die Umschaltung in den Zieladreßraum und die Übertragung der verpackten Parameter dorthin vorgenommen.

  Es können — ohne Änderung des Anwendungsprogramms — andere Subcontracts eingesetzt werden, die z.B. die Aufrufe replizieren oder die Zielobjekte lokal in einem Cache halten und auf diese Weise Interaktionskosten sparen. Prinzipiell können speziell für eine Anwendung neue Subcontracts bereitgestellt werden (→ offene Architektur!)

Programmierung & Schnittstellen (2)

- ❑ Contract: Objekt-Schnittstelle
- ❑ Subcontract: Implementierung der Objektinteraktion
  Kommunikation mit Object-Mgr., Marshalling, etc.



- ❑ Transparente Objektreferenzen (fat pointer)
  - Objekte in anderen Adreßräumen bzw. auf anderen Rechnern werden über sog. *fat pointer* adressiert. Hinter dem fat pointer verbirgt sich der Client-Stub mit dem Subcontract-Mechanismus. Der fat pointer ist wie ein normaler Objektzeiger in C++ verwendbar.

Namensraum-Konzepte

❏ Uniformer Namensraum für Objekte (nicht nur für Dateien)

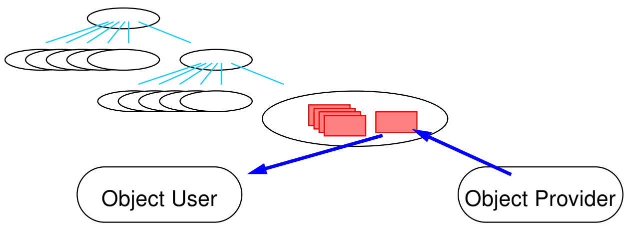
UNIX bietet primär einen Namensraum für Dateien an. Einige weitere Dinge werden ebenfalls darin untergebracht, indem es sie "gewaltsam" zu Pseudo-Dateien gemacht werden — Pipes, Sockets, Geräte, Prozesse. Dieses Vorgehen hat vor allem den Nachteil, daß ein Satz von Datei-Operationen (open, close, read, write, …) auf alle diese Dinge angewandt werden, unabhängig davon, ob alle diese Operationen sinnvoll sind oder nicht. Daneben existieren in UNIX weitere Namensräume, beispielsweise für shared Memory Segmente, Semaphore und Messages, das Prozeßenvironment, Rechneradressen und andere Verwaltungsdaten (NIS-Datenbasis), etc.

Spring geht dagegen von einem uniformen Namensraum für beliebige Objekte aus. Jedes Objekt hat dabei aber seine eigene, auf seine Funktionalität ausgerichtete, Schnittstelle

❏ UNIX-Directories verwalten Zuordnung <Name → Dateireferenz(=Inode)>

❏ ***Context***-Objekte verwalten Zuordnungen <Name → Objektreferenz>

Methoden: ***bind*** = Referenz unter einem Namen in Context ablegen
resolve = Referenz zu einem Namen liefern)



In einem Context-Objekt können Referenzen auf beliebige Objekte unter einem Namen eingetragen werden.

Namensraum-Konzepte (2)

❏ Hierarchie durch Referenzen auf andere Context-Objekte (→ Baum)

Da Contexte ebenfalls Objekte sind, kann in einem Context unter einem Namen natürlich auch eine Referenz auf ein anderes Context-Objekt eingetragen werden. Damit kann man problemlos Hierarchien aufbauen.

❏ **Namensraum**: Context-Baum, der durch einen *Nameserver* gebildet wird

Ein Namensraum wird durch einen Nameserver — eine Spring-Serverkomponente, die ein oder mehrere Context-Objekte anbietet — implementiert.

❏ Kombination von Namensräumen

Namensräume könne auf zweierlei Art miteinander verknüpft werden:

binding Einhängen der Wurzel eines Namensraums in einen anderen Namensraum

Dies ist in etwa mit dem *mounten* von Dateisystemen unter UNIX vergleichbar. Ein Unterschied ist allerdings, daß eine Referenz auf den root-Context eines Namensraums in beliebig viele andere Contexte eingehängt werden kann.

ordered merge geordnetes Übereinanderlegen mehrerer Namensräume

Hierbei werden Namensräume übereinandergelegt und nach außen die Sicht von "oben" angeboten. D.h. man sieht die Vereinigung aller Namensräume. Existieren Objekte in mehreren der beteiligten Namensräume, so sieht man das Objekt aus dem "obersten" Namensraum. Siehe auch die Abb. auf der nächsten Seite.

❏ **Prozeß-Namensraum**: Kombination von Namensräumen für einen Prozeß

- teilweise privat
- teilweise gemeinsam mit anderen Prozessen

Der Namensraum ist grundsätzlich für jeden Prozeß (= Domain) privat. Werden in diesem Namensraum Objekte gebunden, so sind sie nur innerhalb des Domains sichtbar. Wird aus einem Domain ein neuer Domain erzeugt, so erbt dieser den Namensraum von seinem Erzeuger.

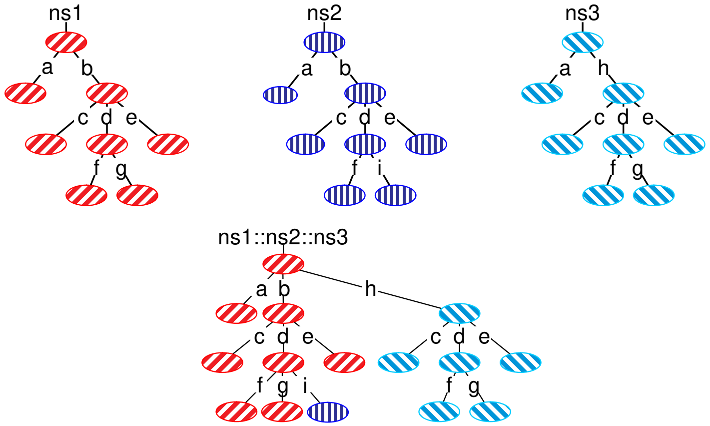
In den Namensraum eines Domains kann eine Referenz auf einen Nameserver eingebunden werden, der einen globalen Namensraum zur Verfügung stellt. Referenzen in dem globalen Namensraum sind auch aus allen anderen Domains sichtbar, wenn in deren privatem Namensraum eine Referenz auf den globalen Namensraumserver eingetragen ist.

Konvention in Spring ist, daß in jedem Domain unter dem Namen /machine ein rechner-lokaler Nameserver eingetragen ist. Unter /machine/services werden Objekte eingetragen, die allgemein erreichbar sein sollen. Außerdem ist unter /village ein Nameserver eingetragen, der einen gemeinsamen Namensraum für alle Rechner eines Clusters (= Village) bereitstellt.

Weitere domain-übergreifende Namensräume werden z. B. vom UNIX-Dateisystemserver sowie von dem NFS- und dem NIS-Modul gebildet.

Namensraum-Konzepte (3)

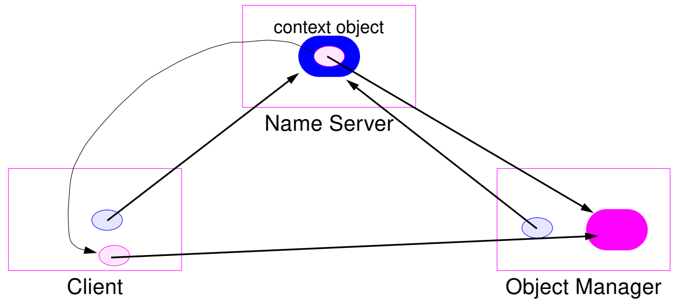
❑ Ordered Merges



Schutzmechanismen

❑ Capabilities

➜ Zugriffsschutz durch kontrollierte Referenzvergabe



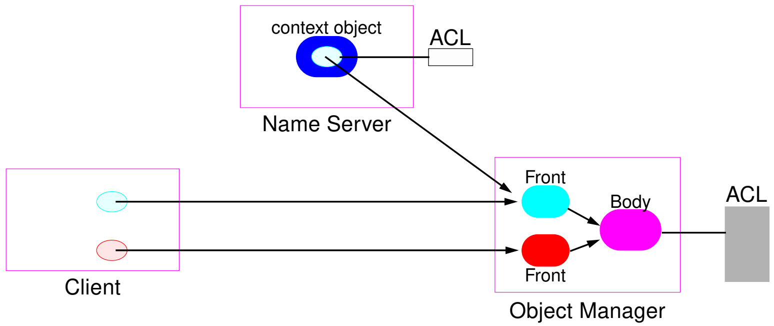
Objektreferenzen in Spring haben Capability-Charakter: wenn jemand eine Referenz auf ein Objekt besitzt, dann hat er Zugriff auf dieses Objekt. Wenn man also eine Referenz an einen anderen Domain übergibt, so hat dieser Domain vollen Zugriff auf das entsprechende Objekt.

Schutz für ein Objekt wird dadurch erreicht, daß man kritisch prüft, wem man eine Referenz auf dieses Objekt gibt. Wird solch eine Referenz in einem Nameserver abgelegt, gibt man damit entweder den Zugriff auf das Objekt völlig frei oder man vertraut darauf, daß der Nameserver die Referenz nur an Berechtigte weitergibt.

Da im Rahmen einer Anwendung eigene Nameserver bereitgestellt werden können, sind hierfür beliebig komplexe oder sehr einfache Lösungen denkbar.

Schutzmechanismen (2)

- ❑ Access Control Listen und Front-Objekte
  - ➜ Zugriffschutz bei Aufruf — aufruferspezifisch



Access-Control-Listen an context-Objekten

Eine weitere, genauere Möglichkeit den Zugriff auf Objekte zu kontrollieren, bieten *Access Control Listen* (ACLs). ACLs können entweder an Context-Objekte gebunden werden, so daß das Context-Objekt im einzelnen prüft, wer Objekte in dem Context binden und wer resolve-Aufrufe an den Context vornehmen darf.

Darüberhinaus kann man ACLs an einzelne Context-Einträge binden, um zu kontrollieren, wer die Referenz des Eintrags über einen resolve-Aufruf abfragen darf und wie er (unter welchem Typ) die Referenz nutzen darf.

Access-Control-Listen an Server-Objekten

Die Kontrolle der Capability-Vergabe alleine ermöglicht zwar Einfluß darauf, an wen eine Referenz abgegeben wird, aber keinen Einfluß darauf, an wen dieser die Referenz möglicherweise weitergibt und wie oft er sie nutzt. Man kann sie ihm vor allem auch nicht mehr entziehen.

Durch Front-Objekte und ACLs am Server-Objekt selbst ist eine Kontrolle über den Referenz-Nutzer selbst möglich. Hierzu wird im Nameserver nicht die Referenz auf das Server-Objekt selbst, sondern auf ein Front-Objekt abgelegt. Dieses Front-Objekt bietet eine *Default*-Schnittstelle an, die i. a. keinerlei Aufrufe zuläßt.

Greift nun ein Client auf dieses Front-Objekt zu, wird anhand der ACL überprüft, welche Rechte er hat, ein Front-Objekt erzeugt, das die entsprechenden Aufrufe durchläßt und seine Referenz automatisch auf das neue Front-Objekt erweitert. Da dieses Front-Objekt später auch wieder ersetzt werden kann, ist es auch möglich, dem Client das Zugriffsrecht wieder zu entziehen.

Authentication

Die bislang ungelöste Frage: wie erfährt ein Server, wer der Client ist. Hierzu werden von einem *Authentication manager* sog. Credential-Objekte verwaltet. Ein Client übergibt an das Front-Objekt des Servers seine Identität. Der Server überprüft diese Information und legt mit dieser Information die Referenz auf das neue Front-Objekt im Credential Obj. des Clients im Authentication Mgr. ab (agiert als Context). Den zugehörigen Namen übergibt er an den Client. Der Client hat eine Referenz auf sein Credential Obj. und kann die Referenz dort abholen.

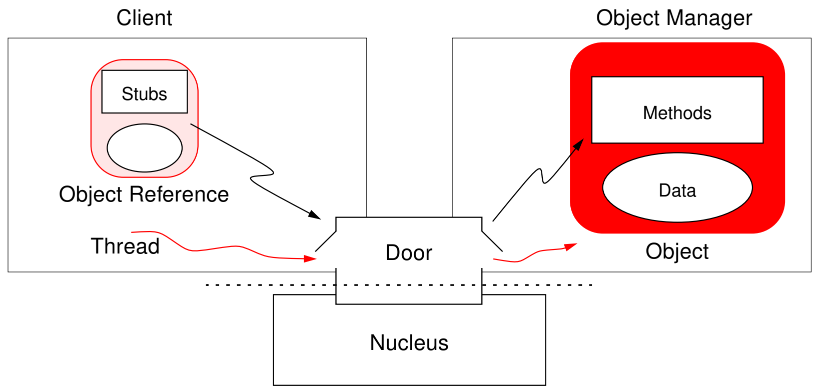
Kern-Architektur

- ❑ Minimalkern
- ❑ Unterstützt drei Basisabstraktionen

- ❑ Domains
  - virtueller Adreßraum und Ressourcen
    (= MACH-Task, UNIX-Prozeß ohne Aktivitätsträger)
- ❑ Threads
  - leichtgewichtiger Prozeß
  - wird in einem Domain ausgeführt
- ❑ Doors
  - Basismechanismus für Methodenaufrufe über Adreßraumgrenzen
  - steht für Einsprungpunkt in einen Domain

Objektinteraktion über Adreßraumgrenzen

- **Domains** & **Threads**
- **Doors**: sehr effizienter Mechanismus zur Interadreßraumkommunikation



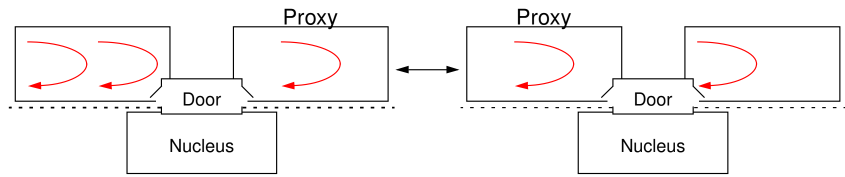
Doors realisieren einen geschützten Einsprungmechanismus in einen Domain (der einen Objekt-Server realisiert, in dem also Server-Objekte bereitstehen).

Ein Client-Thread ruft über einen Aufruf an der Client-Seite der Door eine Methode des Servers auf. Dadurch entsteht auf der Server-Seite der Door ein Server-Thread der die Methode ausführt. Am Ende der Methodenausführung kehrt der Thread in die Door zurück und der Client-Thread, seinerseits, kehrt aus der Client-Seite der Door zurück.

Die Door-Aufrufe erfolgen im Rahmen des Subcontracts — sind also im Normalfall für den Anwendungsprogrammierer nicht sichtbar.

Objektinteraktion über Rechnergrenzen

- Netzwerktransparenz durch **Network-Proxies**



Doors unterstützen nur Methodenaufrufe zwischen Domains auf einem Rechner. Durch sog. Proxy-Server können solche Methodenaufrufe in Netzwerkpakete verpackt werden, über Rechnernetze übertragen werden (z. B. über TCP/IP), im Proxy-Server der Gegenseite wieder ausgepackt werden und schließlich an den eigentlichen Server zugestellt werden.

Network-Proxies sind für Client und Server völlig transparent. Der Client erhält einfach statt einer Referenz auf das eigentliche Server-Objekt eine (identisch aussehende) Referenz auf den Network-Proxy.

Threads

Programmierschnittstelle

❑ Repräsentation als C++-Basisklasse

Threads werden als abstrakte C++-Klasse *thread* zur Verfügung gestellt.

❑ Nebenläufige Funktionen als Methoden von Subklassen

Nebenläufig auszuführende Aufgaben werden in der Methode *startup( )* einer konkreten Subklasse von *thread* implementiert.

❑ Instantiierung einer Thread-Klasse erzeugt Thread

Durch Instantiierung einer konkreten Subklasse von *thread* wird ein neuer Thread erzeugt (in suspendiertem Zustand). Durch Aufruf der Methode *make_runnable( )* (wird in der *thread*-Basisklasse bereitgestellt) wird der Thread schließlich in den Zustand "bereit" überführt.

Realisierung

❑ Uniformes Kernel-Thread-Modell

Spring-Threads werden generell durch den Minimalkern verwaltet. Es gibt nur einen Scheduler und nur eine Thread-Art. Unterscheidungen nach Anwendungen werden nicht getroffen.

Die derzeit vorhandenen Thread-Konzepte sind nur ein erster, einfacher Ansatz.

❑ Scheduling und Koordinierung nur durch Minimalkern

Koordinierungsmechanismen (z. B. durch mutex-Variablen) müssen zum Blockieren und Deblockieren von Threads immer Aufrufe an den Minimalkern durchführen. Auch Thread-Wechsel (z.B. im Rahmen von Zeitscheiben-Scheduling) sind nur durch den Minimalkern möglich.

➜ **unflexibel & teuer**

❑ Aufrufe über Domain-Grenzen

- Benutzersicht:
  - Thread wandert
- Realisierung:
  - im Zieladreßraum wird ein neuer Thread mit der Bearbeitung beauftragt, der Aufrufer wird blockiert
  - zusammengehörende Threads bilden ein *Shuttle* (Schedulingeinheit)

Ortstransparente Methodenaufrufe

❑ zentraler Interaktionsmechanismus in Spring

Im Gegensatz zu traditionellen Betriebssystemen gibt es in Spring mit dem Methodenaufruf einen uniformen Interaktionsmechanismus.

- Anwendung ↔ Anwendung (traditionell IPC)

  In herkömmlichen Betriebssystemen gibt es hierfür unterschiedliche IPC-Mechanismen (Dateien, Pipes, Messages, shared Memory, …)

- Anwendung → Systemkern (traditionell *System call*)

  Diese Interaktion erfolgt traditionell über einen Trap bzw. SVC und eine Verzweigung zur gewünschten Kern-Routine über eine Dispatcher-Funktion im Kern.

- Systemkern → Anwendung (in traditionellen Systemen nicht vorhanden)

  Diese Art der Interaktion wird. i. a. als *upcall* bezeichnet und ist — aus Sicherheitsgründen und weil es traditionell nicht üblich war, daß das Betriebssystem Dienste von Anwendungen nutzt — bislang in Betriebssystemen kaum zu finden.

  Eine upcall-vergleichbare Konstruktion wird allerdings durch den NFS-Clienten im UNIX-Kern realisiert — er löst einen RPC über die Netzwerkschnittstelle des Systems an eine IP-Adresse/Port-Nummer aus. Hinter dem Ziel kann sich ein normaler Anwendungsprozeß verbergen (z. B. ein *automounter*), der dann eine Dienstleistung für eine Betriebssystemkomponente (das Dateisystem) bereitstellt.

❑ Grundlagen

Um einen Methodenaufruf an ein Objekt zu programmieren, muß im Client-Programm auf jeden Fall die Schnittstelle des Server-Objekts bekannt sein. Diese Schnittstellenbeschreibung muß als IDL-Datei vorliegen.

- Schnittstellenbeschreibung für Zielobjekt (Contract, IDL)
  - wird für Systemobjekte mitgeliefert (`/sys/interfaces`)

    Unter dem Directory /sys/interfaces befinden sich in Spring die Schnittstellenbeschreibungen aller für Anwendungen verfügbaren Systemobjekte (implementiert als Teil des Minimalkerns oder in eigenen Server-Domains).

  - für Anwendung ↔ Anwendung Interaktionen
Angelegenheit der Anwendungsentwickler

    In diesem Fall hat der Programmierer die Schnittstellenbeschreibungen entweder zusammen mit der Server-Software (selbständiger Server, z.B. ein neuer Nameserver oder eine Klassenbibliothek) erhalten oder selbst erstellt und muß sie zur Übersetzung seiner Client-Software zur Verfügung stellen.

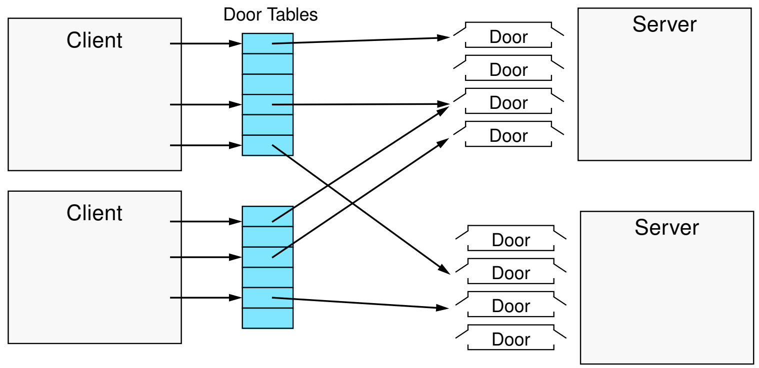
  - für Systemkern → Anwendung Interaktionen
Schnittstellen konform zu dem System bekannten Schnittstellen

    Systemobjekte (unabhängig davon, ob sie Teil des Minimalkerns sind oder in einem Server ablaufen) können natürlich nur mit Objekten interagieren, deren Schnittstelle sie kennen. Will ein Anwender einen eigenen Systemdienst im Rahmen seiner Anwendung bereitstellen (z. B. einen eigenen Nameserver mit besonderen Zugriffsschutzkonzepten), so muß dieser Dienst schnittstellenkonform zu den entsprechenden Standarddiensten sein (ein Subtyp also).

- Subcontract-Konfiguration

Doors

❑ Strukturen



❑ Door enthält PC und spez. Daten (Data-Cookie) für Aufruf auf Server-Seite

Die Door Tables werden vom Minimalkern verwaltet, damit es für eine Anwendung nicht möglich ist, sich unberechtigt Zugriff auf eine Door zu verschaffen.

Wird im Rahmen eines Methodenaufrufs eine Objektreferenz (fat pointer) übergeben, wird beim Marshalling dieser fat pointer durch den Door-Descriptor (Index in die Door Table) ersetzt. Bei der Übergabe eines Door-Descriptors an einen anderen Domain, wird in dessen Door Table eine Referenz auf die entsprechende Door eingetragen und dieser Index als Door-Descriptor herausgegeben. Beim Unmarshalling im Ziel-Domain wird dann aus diesem Door-Descriptor wieder ein fat pointer erzeugt.

Doors(2)

❑ Fast Path

- Parameter werden in Registern übergeben; max. 40 Bytes, max. 1 fat pointer
- häufigster Fall (ca. 80 - 90 %)
- Ablauf:
  - Door-Id, Methoden-Nr., Parameter-Zahl und Parameter aus Marshal-Buffer werden in Register geladen
  - Trap in Systemkern
  - Wenn Reg. `g2` Door-Id (fat pointer Parameter) enthält, Referenz auf diese Door in Door-Table des Ziel-Domain eintragen
  - Thread im Ziel-Domain besorgen (aus Thread-Pool) und Shuttle zuordnen
  - Startpunkt des neuen Threads aus Door entnehmen
  - Zugriff auf "alte" Registerfenster des Aufrufers sperren
  - Domain-Context umschalten und auf neuen Thread wechseln
- Kosten:

Action	call	return	total
Fiddling with register windows	10	9	19
Switching thread/process	12	11	23
Getting target info	7	0	23
Reference-counting target door	3	4	7
Saving/restoring return info	6	4	10
Switching VM context	7	7	14
Miscellany	13	9	22
Total	58	44	102

❑ Vanilla Path

- Parameter werden kopiert: Sender → Kern → Empfänger
- bis ca. 9 KByte

❑ Bulk Path

- Daten sind auf Seitengrenzen ausgerichtet
- Seiten werden in den Empfänger-Adreßraum abgebildet

❑ Kostenvergleich

- Inter-Adreßraumaufrufe in verschiedenen Betriebssystemen (Werte auf Sparcstation2-Performance umgerechnet - daher nur grob vergleichbar)

	µs
Berkeley socket	850
Taos Leightweight RPC	18
MACH 3.0 Message	44
Windows NT event pair	48
Spring cross-domain call	11